Designing and Proving Robust Safety of Efficient Capability Machine Programs

Aïna Linn Georges

PhD Dissertation

SOLIDUM PETIT IN PROFUNDIS · UNIVERSITAS ARHUSIENSIS ·

Department of Computer Science
Aarhus University
Denmark

Designing and Proving Robust Safety of Efficient Capability Machine Programs

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Aïna Linn Georges
October 26, 2023

Abstract

Memory safety vulnerabilities have plagued the computer security field for decades. High level languages such as Rust enforce memory safety through type systems and abstract representations of memory pointers. Unfortunately, these languages are compiled to low-level languages that are unable to enforce the same guarantees. In order to guarantee the security properties of the source language, it's thus important to target a machine with the necessary primitives to enforce them.

Capability machines are a kind of architecture that allow fine-grained privilege separation using *hardware capabilities*. Hardware capabilities grant authority over segments of memory, and can thus be used as primitives to enforce memory safety. However, maintaining safety can quickly become complex in systems with multiple domains, each with their own distinct authority, and with multiple switches between them. Such implementations must ensure that no authority is leaked from one domain to another, and that control flow is not tampered with. Domain switches are typically implemented using a *call-stack*, a shared structure with special spatial and temporal properties. However, enforcing these properties can lead to undesired overhead. In this dissertation, we propose new capability designs to efficiently enforce *stack safety*.

To establish high levels of confidence, and create a convincing proposal, each design is fully and mechanically formalized in models that capture and prove the desired security properties. These properties can be subtle, and complex to prove in low level machines. It is thus important to create mechanized frameworks to prove that the intended goals are met.

In Chapter 1, we outline the concept of capabilities and capability safety, provide relevant background, and present the general motivations behind this dissertation. In Chapter 2, we outline Cerise, the first of three frameworks created within this dissertation. Cerise consists of a program logic for reasoning about known capability machine programs, and a model that formally captures capability safety. Cerise can be used to prove functional correctness of capability machine programs, even whey they interact with unknown code. In Chapter 3, we introduce *uninitialized capabilities*, a new design that can be used to efficiently enforce spatial stack safety properties. In Chapter 4, we introduce *directed capabilities*, a new design that can be used to additionally enforce temporal stack safety properties. Finally, in Chapter 5, we lay the foundation for the exploration of a secure compiler that targets capability machines, by formalizing the security properties of WebAssembly, an interesting source language for a secure compiler.

Resumé

Sikkerhed relateret til hukommelse har i årtier været et betydeligt problem indenfor computersikkerhedsområdet. Højniveausprog som Rust opretholder hukommelsessikkerhed gennem typesystemer og abstrakte repræsentationer af hukommelsesmarkører (en slags værdi der peger på et hukommelsesfelt). Desværre er programmer skrevet i disse sprog ikke i stand til at sikre de samme garantier når de oversættes til lavniveau-sprog.For at kunne garantere kildesprogets sikkerhedsegenskaber, er det således vigtigt at vælge et lavniveau-sprog med de nødvendige primitiver til at bevare sikkerhedsgarantierne.

Kapacitetsmaskiner, også kendt som *capability machines*, er en slags arkitektur, der tillader adskillelse af privilegier ved hjælp af såkaldte *hardware kapaciteter*. Hardware kapaciteter giver autoritet over hukommelsessegmenter, og kan således bruges som primitiver til at opretholde hukommelsessikkerhed. At opretholde sikkerhed bliver dog hurtigt en kompleks process i systemer med flere domæner, hvor kontrol flere gange skiftes fra et domæne til det andet. I disse situationer er det vigtigt at sikre, at ingen kapaciteter lækker fra et domæne til et andet, og at domæneskifterne ikke manipuleres.

Domæneskift implementeres typisk ved hjælp af en såkaldt *call-stack*, en delt struktur med særlige egenskaber. Desværre kan det føre til uønsket tidsspilde at opretholde en sikker *call-stack*. I denne afhandling foreslår vi nye kapaciteter til effektivt at opretholde en sikker *call-stack*.

For at etablere tillid og skabe et overbevisende forslag, er hvert design mekanisk formaliseret i modeller der opfanger og beviser de ønskede sikkerhedsegenskaber. Disse egenskaber kan være subtile og komplekse at bevise. Det er derfor vigtigt at skabe mekaniserede rammer for at bevise, at de tilsigtede mål er opfyldt. I denne afhandling præsenterer vi i alt fire mekaniserede formaliseringer, tre som formaliserer kapacitetsmaskiner, og en som formaliserer WebAssembly, og bruger dem til at etablere sikkerhedsegenskaber fra programmer skrevet i lavniveau sprog.

Acknowledgments

... but when pain is over, the remembrance of it often becomes a pleasure.

Jane Austen, Persuasion

I want to begin by thanking my supervisor Lars Birkedal. Lars offered guidance when needed, and encouraged me to seek out my independence when most important. He has helped me overcome my insecurities without me realizing it. Most importantly, Lars has always been kind and patient, encouraging my creativity and enthusiasm.

I also want to extend my gratitude to the many amazing collaborators I've had the pleasure to work with. Collaboration and community building will always be the biggest strength of research and academia. I want to especially thank to Alix and Armaël, whose collaboration and friendship was integral to my PhD, as well as my mentees Bastien and Maxime, who I look forward to work with in the future.

I want to thank the many mentors and role models who lead me towards research. Brigitte Pientka introduced me to programming languages research, and showed me that I could belong in computer science. Lau Skorstengaard was an incredible mentor in the beginning of my PhD, and I am grateful that I got the privilege to build on his research.

I could never have done this PhD without the friendship and support from the LogSem group here at Aarhus University. This is a truly unique group, and it might sound cliché, but it has become like family to me. I want to especially thank my office mate Léon, as well as Jean, Jonas and Alejandro for being my dear friends, with whom I can always count on for research brainstorming, serious Friday evening discussions, and silly weekend marathons. The same goes to all my friends. I especially thank my boardgame friends Hart, Kevin, Per and Peter, for enriching my life beyond research, and allowing me to win once in a while.

Finally, I want to thank the most important people in my life. I could not have done this without the support of my partner João, who was there for me when I needed it the most. To my mom, I want to thank her for everything. She has given me unconditional love and support all my life. This dissertation is for her.

Aïna Linn Georges,, Aarhus, October 26, 2023.

Contents

Part I

Overview

Chapter 1

Introduction

1.1 Memory Safety Vulnerabilities in Hardware Architectures

While software systems continue to increase in scope, ubiquity and complexity, the underlying hardware architecture has, in its fundamentals, largely remained unchanged. To this day, the most prevalent computer architecture systems are, in essence, Von Neumann machines. Industry efforts have centered around efficiency, often at the cost of security. However, as computer systems are becoming an integral part of society, it is important to create systems that are secure at every level of abstraction. In the advent of Spectre, Meltdown and Heartbleed, we have witnessed increased concern about the security flaws of modern microprocessors, despite the impact mitigations have on efficiency [114].

Over the years, an extensive body of work has identified many vulnerabilities in the languages of hardware architectures, namely assembly languages and C. Conventional instruction set architectures, such as ARM, x86 or RISC-V, all depend on variations of the same model for pointers, in which memory is accessed via integer indices. Likewise, rather than crashing, C allows for undefined behavior when integers are used as pointers. As a result, all these languages are vulnerable against memory corruption attacks, which have plagued the computer security field for decades. In recent surveys, it was observed that 70% of all issues in Microsoft products [141] and in the Google Chrome browser [32] are memory safety related.

To make matters worse, unsafe pointers can lead to control flow hijacking. Compilers typically handle control flow by implementing a call-stack, which is a piece of memory that stores return pointers and local variables. As a result, unsafe stack pointers can be manipulated to access not just the local state of another function, but the pointers that directly determine control flow. Without proper mitigation techniques, assembly languages and C are thus vulnerable to control flow hijacking attacks.

Many mitigation techniques have been proposed, at different levels of the software stack, ranging from address space layout randomization [113], stack canaries [146], data execution prevention [96] to control flow integrity [2] (see [139] for a survey). Unfortunately, while many of these techniques mitigate the risks, they do not entirely eliminate them [22, 25, 28, 127]. In many ways, they seek to address the symptom, rather than the cause: that modern hardware architectures lack the necessary abstraction to enforce memory safety, and prevent memory corruption attacks.

Language-based techniques have for a long time provided promising approaches to security [124]. From certifying compilers to type systems, language-based approaches to security offers a variety of efficient enforcement mechanisms, all following the principle of least privilege [120]. For example, the systems programming language Rust offers an alternative to C that is both type-safe and memory-safe. Rust promises to offer low-level control, while maintaining strong safety guarantees. Moreover, these safety guarantees have been rigorously investigated and formally verified [76, 78]. Unfortunately, code written in these languages is compiled to low-level target assembly languages, which do not robustly preserve the source level safety guarantees. As a result, as soon as it is linked with native target level code, the code that actually gets executed inherits all the vulnerabilities of the target language.

The solution investigated in this dissertation is to target a *safe* assembly language instead. In the following paragraphs, we first review some related approaches, then describe the one investigated in this dissertation.

Proof carrying code (PCC) [106] is a mechanism that guarantees safety by requiring that untrusted code comes with a formal proof of safety, which the machine then validates before executing the associated code. In such systems, the onus is on the untrusted agent to establish safety of their code, whereas the proof checker simply validates its correctness. Proof validation is in many ways similar to a type checker, in that both statically check that a proof, or a type, is correct according to an established set of rules.

While type systems may only express a fixed set of safety properties, as determined by the richness of the types, they still display advantages over proof carrying code. Since type-checkers automatically establish whether a program is well-typed, types present a significantly smaller barrier of entry compared to proofs of safety, which must be established manually. Typed Assembly Language (TAL) [102] is a RISC-like assembly language with a type system that enforces high level abstractions such as data abstraction and encapsulation. More recently, WebAssembly [65], a language initially designed for the web, is now an increasingly popular portable bytecode language featuring a simple type system, with a formally defined operational semantics and proof of soundness.

PCC, TAL and WebAssembly take a static approach to establish safety. Alternatively, dynamic approaches can guarantee safety *a priori*, imposing no static restrictions on the executing code. However, dynamic approaches require bespoke languages with baked in dynamic checks.

Capability machines, the focus of this dissertation, are a kind of architecture that enable dynamic fine-grained memory protection via *hardware capabilities* [27, 36, 91]. A capability is a concept that exists at multiple levels of abstraction, from software to hardware. Conceptually, capabilities represent *unforgeable tokens of authority* [128]; they are tokens, meaning they are tangible primitives distinguished from other primitives such as integers, they are unforgeable, which means no unprivileged operation can result in a new capability, and finally, each capability carries with it a specified amount of authority, which can never be increased.

At the machine level, capabilities grants authority over segments of memory, and precisely delineate authority over memory regions. In order to access a memory region, one must demonstrate the ownership of a capability with sufficient authority. Capabilities can thus act as the building blocks for implementing memory safety properties. However, these implementations can quickly become complex. Consider a scenario with multiple processes, each with their own distinct authority. Secure implementations of such a scenario must ensure that one process never gains access to capabilities outside its intended authority, a goal that is particularly subtle when transferring from one process to another, since reachable objects must change to match the new process' authority, without leaking objects owned by the previous process.

In this dissertation, we propose new capability designs, and leverage them to implement efficient transfers from one domain to another through secure calling conventions. To establish high levels of confidence, and create a convincing proposal, each design is fully and mechanically formalized in models that capture and prove the security properties of the proposed calling conventions. These properties can be subtle, and complex to prove in low level machines, that do not enjoy the abstractions of high level languages. It is thus imperative to create mechanized frameworks to not just prove the properties in question, but lay the groundwork for rigorous investigations of new designs. Before detailing the precise contributions of this dissertation, we provide context behind capability machines, the kind of security properties that can be achieved using capabilities, how they have been formalized and reasoned about, and how low-level machines have been mechanized in proof assistants.

1.2 Capabilities as Security Primitives

The use of a capability-based access control for dynamic protection mechanisms has been intended since the inception of capabilities. Dennis and Van Horn [36] introduce the concept of a segment capability within the context of *multiprogrammed computer systems* (MCS); a system running multiple concurrent processes, for more than one user. These computations may share resources, pointed to by references, and may require different sets of computing resources, varying throughout their execution. These resources should be protected by an access control mechanism. Crucially, the access control system may need to adapt to an ever changing environment.

Dennis and Van Horn [36] introduce a range of concepts and terminology. A *segment* stores a list of words of information, and may be referenced by a *word name* $[i,a]$, consisting of an index i, abbreviating the name of a segment, and an address a. An MCS implements memory protection on a segment basis, by keeping a *list of capabilities*, specifying the *sphere of protection* of a *computation* (a set of processes), where each capability points to some computing object, and specifies the means of accessing that object. In particular, a segment capability contains a reference pointer (word name) and a permission, an indicator of the possible actions permitted by that capability: X, R, XR, RW and XRW, where X stands for executable, R for readable and W for writable.

During the execution of a computation, capabilities are dynamically added and deleted from its associated list of capabilities. Multiple processes may execute within that computation, thus sharing access to the resources granted by the capability list, but processes of another computation may not have access to the same capabilities. As such, capabilities allow for a dynamic access control protection mechanism, on systems with multiple computations, and protected resources.

Lampson [87] generalizes the protection mechanism of capabilities into a model for *dynamic protection structures*. As in [36], the model assumes a computation C, running on a multi-access system, executing multiple programs that at various times require access to system resources. The primary concern of a dynamic protection structure is to manage the resources, also denoted as objects, that a process (the currently executing program) has access to. Lampson [87] identifies three fundamental ideas behind the protection structure. First, objects are named by capability tokens, which themselves must be unforgeable. Second, capabilities are grouped into domains (previously denoted as spheres of influence), and each domain may have access to a different set of capabilities. As such, "when control passes from one domain to another (in a suitably restricted fashion) the capabilities of the process will change" ([, §1]). Third, capability ownership is implemented via so-called access keys, which specifically serve as indicators of authority.

According to Lampson, a capability is the "protected name of an object". To protect the integrity and unforgeable nature of a capability, Lampson [87] proposes the following structure for hardware dependent capabilities.

TAG	TYPE	VALUE

In this scheme, the underlying hardware must allow for words to be *tagged*, with a bit that can only be modified by the machine itself. The TYPE indicates what kind of object the capability grants authority over, and the VALUE is the pointer to the object in question.

During the transfer of control from one domain to another, the protection structure must carefully handle the ensuing acquisition of new capabilities. Lampson [87] thus introduces the notion of a protected entry point which he calls a *gate*; a new kind of capability specifically for changing domains. Invoking a gate is essentially a protected function call. As such, the subroutine in charge of implementing the domain transfer must also create a means of returning. However, the latter cannot take the form of another gate; "it is not satisfactory to create another gate which the called process may return through, since he might save it away and use it to return at some later and unexpected time" ([87, §5.1]). Instead, the protection system must implement some kind of call-stack to handle domain transfers.

It is interesting to note, that while our setting will be adapted to modern capability machines, the difficulties raised by Lampson are closely related to the difficulties tackled in this dissertation. In the following subsections, we present various existing capabilities, with a particular focus on CHERI (a modern capability machine that is at the heart of this dissertation), after which we delve into these exact challenges, and how they relate to this dissertation.

1.2.1 Capabilities at Different Levels of Abstraction

Capabilities in Language Design Building on the ideas presented by Dennis and Van Horn and Lampson, Morris Jr. [100] describes how programming language features can be used to protect one subprogram from another. In object oriented languages, where procedures are regarded as objects, an object is *local* when it is accessible only to part of a program. By restricting the accessibility of local objects, and by extension its associated procedures, objects become capabilities. More precisely, an object-capability is an encapsulated object, that communicates to other objects by invoking their procedures via accessible references, and may restrict access to its own procedures simply by not sharing references of itself to other objects. Examples of object-capability languages include E [97] and joe-E [94], two Java-like object-capability languages, Emily [134], a restricted version of OCaml, and Caja (as well as Cajita) [98], a capability-safe subset of JavaScript.

Capabilities in Operating Systems Operating systems must enforce an increasingly wide range of security policies. Among their responsibilities is the enforcement of an access control system in charge of authorizing and restricting a subject's access to certain protocols as well as its ability to perform operations on objects such as files or sockets. Conventional operating systems implement a Discretionary Access Control system (DAC), in which users specify permissions for an object, which are subsequently checked by the OS whenever it is accessed. On the other hand, a Mandatory Access Control system (MAC) implements a privilege-based approach, in which users have system defined access permissions, that are checked at runtime whenever a user attempts to perform a protected task. Unfortunately, MAC approaches are insufficient to protect a system running many processes from different sources on behalf of one user. As a result, web browsers such as Chromium cannot take full advantage of MAC, and must sandbox different components into several processes, where untrusted sources have restricted access to user data. Implementing Mandatory Access Control systems using capabilities offers an alternative approach, which prevents confused deputy issues [66].

Using capabilities as security primitives in operating systems is an old idea. In their provably secure operating system (PSOS), Feiertag and Neumann [50] propose the use of capabilities as primitives to enforce an access control system. The microkernel-based operating system FLASK [133] offers flexible support for security policies via capability-based Mandatory Access Control. More recently, Capsicum [156] extends UNIX with capability support, enabling lightweight sandboxing, in accordance with the intention behind Mandatory Access Control.

Another notable capability-based system is seL4, a microkernel based on the L4 family of microkernels, with support for capabilities to authorize inter-process communication [46]. All memory accesses are protected and managed using capabilities, which are themselves stored in special capability address spaces enclosed within containers called *CNodes*. Functional correctness of the seL4 kernel has been formally verified in Isabelle/HOL [126]. The formalization proves a refinement property, establishing a correspondence between the high-level abstract specification of the kernel against its low-level C implementation. Building on this refinement, Klein et al. [80] prove that seL4 enforces two high-level security properties: integrity of relevant state, in accordance to a given access policy, and authority confinement, which limits the propagation of authority between subjects.

Hardware Capabilities Capability-based hardware designs have been implemented since the inception of capabilities, dating back to the 60s (the Chicago Magic Number Computer [47–49], or the System 250 built by the Plessey Company [33, 34]). These machines implement capabilities by storing access permissions in tables. More recent capability machines, such as the M-Machine [26], implement capabilities via so-called guarded pointers. Rather than maintaining tables of access permissions, the M-Machine encodes the authority of a capability within the pointer object itself, and maintains a tag to distinguish pointers from integers. Guarded pointers enable fast context switching, since the change of protection domain no longer needs to manage the different tables of access permission.

CHERI [158], a modern and developed family of capability machine architectures, also features guarded pointers (referred to as fat pointers) as the capability representation. Over the last decade CHERI has matured into an extensive design featuring, among other, CheriBSD [154], a full UNIX-style operating system. Ideas from CHERI have recently been adopted by ARM in their Morello project [12], which is aimed at developing concrete CPU designs and prototypes that could be implemented in future hardware. The Morello board provides a platform to explore new security features that may realistically be implemented in hardware.

While early capability machines implement capabilities indirectly through capability lists and ownership tables, modern capability machines, including CHERI, implement capabilities as direct replacement for pointers. On such machines, the set of reachable capabilities, starting from the register state, exactly defines the authority of the currently running process.

Concretely, a CHERI capability is a fat pointer with metadata describing its authority. A memory capability's authority is determined by a *permission* (such as readable, writable, etc.) and by a *bounds interval* (the physical boundaries of a memory region). Below we represent a capability as a tuple $(perm, len, b, a)$, describing a capability pointing to a, with the ability to access addresses within $[b, b + len)$ via operations permitted by $perm$.

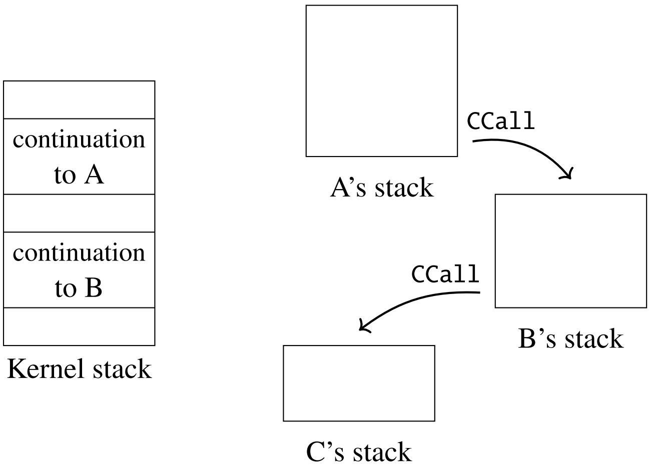
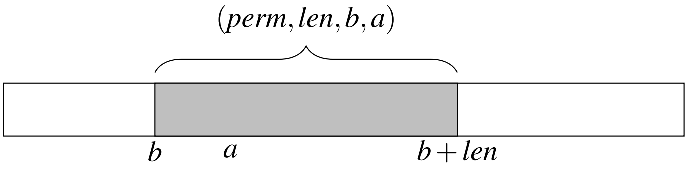


Figure 1.1: The trusted intra-component call-stack is managed by the kernel, while each component possess its own intra-component stack



CHERI allows capabilities to be *out-of-bounds*, meaning a may be outside $[b, b + len)$. Rather than maintaining some kind of capability invariant, bounds and permissions are enforced on dereference, by triggering a dynamic check that validates the capability before executing the operation. If the capability does not carry sufficient authority, the operation fails. By means of sophisticated optimizations and compression techniques, CHERI efficient implements capabilities, including native bounds and permission checks to dereference operations, with little runtime overhead.

In addition to memory capabilities, capability machines usually offer a capability for closures that represent the authority to invoke a component without exposing its implementation details and its private capabilities (recall Lampson's *gate*). Invoking such a capability passes control to the other component and makes available its private capabilities and thus, its authority. As such, they offer a cheap form of domain transfer. These capabilities are often referred to as *object capabilities*, and are closely related to their high-level counterparts. The M-Machine implements object capabilities by using sentry-capabilities, which are capabilities with a special *enter* permission E that only permits jump instructions. On CHERI[1], object capabilities take the form of pairs of code and data capabilities, tied together by being sealed with a common seal [154, 155]. Sealing is a primitive CHERI operation that renders capabilities opaque and unusable, except that the pair can be invoked with a special instruction `CCall`.

[1] Note that CHERI still has sentry capabilities

An object may then return by invoking the special instruction `CReturn`.

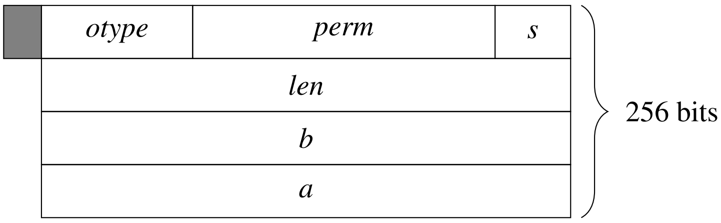
Both sentry capabilities and sealed capabilities offer lightweight domain transitions between capability machine processes. More precisely, a domain transition on a capability machine is triggered by invoking one of these object capabilities. Authority can be transferred from one process to another by keeping specific capabilities in registers when transferring control. Meanwhile, the authority of capabilities that are kept out of reach is guaranteed to be isolated from the new process. In this dissertation, we will use sentry capabilities to implement secure domain transfers, but note that many of the ideas can likewise be applied to seals.

While domain transfers on CHERI are relatively lightweight, they must still pass through the kernel, which manages a trusted call-stack to maintain well-bracketed inter-component calls and returns. Upon invocation via `CCall`, the kernel pushes the caller component's continuation onto the stack, which is subsequently popped upon return via `CReturn`. Each component then manages its own personal stack for intra-component calls. Figure 1.1, inspired by a similar figure in [154], illustrates a sequence of component invocations, and the resulting chain of per-component stacks, implicitly linked through the trusted call-stack.

Local variables defined within a function in C have a lifetime limited to the function's execution and are stored in the function's stack frame during runtime. To distinguish between capabilities that are local to a function, and those that are allocated on the heap and are not limited to the lifetime of a function, CHERI introduces the notion of *local capabilities* [154]. Conceptually, local capabilities grant temporary authority, and are meant to be used during the current execution, but lost upon return. This is enforced by allowing local capabilities to be stored in registers, but not on the heap. However, local capabilities should still be able to live on the stack. After all, local capabilities are exactly meant to point to stack variables, which may be passed as parameters on the stack in case of register spilling. This is achieved with a special "write-local" permission, specifically intended for the stack capability, which (unlike capabilities pointing to the heap) will allow storing local capabilities. Later in this introduction, we will detail what happens when multiple domains share a stack, pointed to by a shared write-local stack capability.

Above, we showed an abstract representation of a capability as a tuple. Concretely, CHERI capabilities are native machine values of fixed size and layout. Capabilities are distinguished from integers by virtue of a tag, which user-level instructions can only read from, but not write to. The tag is maintained via *tagged memory*, in which physical memory addresses are associated with a 1-bit tag, indicating whether that address stores a capability or an immediate. Subsequent store operations may alter the state of a memory address tag, but no user-level instruction can directly manipulate it, thus ensuring that no capability can appear out of thin air. Operations that manipulate capabilities are dynamically validated, such that the authority of a capability never increases. As such, tagged memory and dynamically validated manipulations render capabilities unforgeable. More precisely, new capabilities can only be derived from existing ones, and the authority of the currently running process is never increased.

Early versions of the CHERI instruction-set architecture implement a 256-bit capability format [154, 161], where *otype* determines the object type (distinguishing closures from memory capabilities, and linking an object's sealed code and data capabilities), *perm* determines its permission and locality, *s* is a bit indicating whether the capability is sealed, *len* is the size of the region, *b* is its base, and *a* its pointer value.



While the above format is simple to read and understand, it incurs a significant spatial overhead. More recent versions of CHERI thus implement rigorous compression schemes to achieve capabilities of 64 bits for 32-bit architectures, and 128 bits for 64-bit architectures [162]. For the purposes of this dissertation, we abstract away the details of the concrete implementation of a capability, and represent them more abstractly as simple tuples.

1.3 Enforcing and Characterizing Capability-enabled Security Properties

While each kind of capability greatly varies in implementation and abstraction level, they all share common principles and purpose. Capabilities are primitives that dynamically facilitate the compartmentalization of multiple sandboxed components. In early capability machine designs, a component (referred to as a computation) owns a list of capabilities determining its sphere of influence, and compartmentalization dictates that this sphere of influence is encapsulated and protected throughout execution.

On modern capability machine processors such as CHERI, a component is defined in terms of an object capability (either a sentry capability, or a sealed code and data capability pair), and a component's sphere of influence corresponds to all the reachable capabilities once the object capability is invoked. Compartmentalization is thus achieved by creating closures via sealing, or via sentry capabilities, by the absence of ambient authority (such as static objects), and the inherent prevention of privilege escalation, also called authority amplification.

As a result, capabilities enable compositional reasoning of individual components. In other words, safety of a component is guaranteed to be robust against the surrounding context, assuming that the component does not self-sabotage by exposing its safety-critical capabilities to the context. This restriction, however, is quite flexible, and does not prevent fine-grained resource sharing. By using the ability to copy and restrict the authority of a capability, a component can share parts of its resources with another component, while preserving the integrity and confidentiality of its remaining safety-critical resources.

These notions have been formalized in multiple characterizations of capability-enabled security properties. In the context of object-capability languages, many of such works characterize capability safety in terms of *reference graphs* [93, 99]. Maffeis et al. [93] develop a language-based foundation for reasoning about object-capability languages, by defining a more general notion of *authority safety* as the combination of two principles: any access must derive from the set of reachable authority established at the beginning of the current object's execution, and authority is either derived from an object's initial authority, achieved through interaction, or created over new allocated resources.

The approach uses reference graph dynamics to determine the authority of an object, and as such is based on the syntactic structure of an object capability program. However, as a result, the approach lacks the ability to capture the more refined dynamic behaviors of object capabilities. In response, Devriese et al. [37] develop a Kripke logical relation to reason about typical object capability patterns, fully capturing the dynamic behavior of capabilities.

Building on this model, Swasey et al. [138] introduce a logic for object capability patterns called **OCPL**, the first formal system for compositional reasoning about the security properties enabled by object capability patterns. **OCPL** can be used for modular reasoning about libraries that implement object capability patterns, as well as the clients that depend on them. Furthermore, the logic is fully mechanized in Iris [77], a higher order separation logic framework, making **OCPL** a framework for mechanically verifying robust safety properties of object capability patterns.

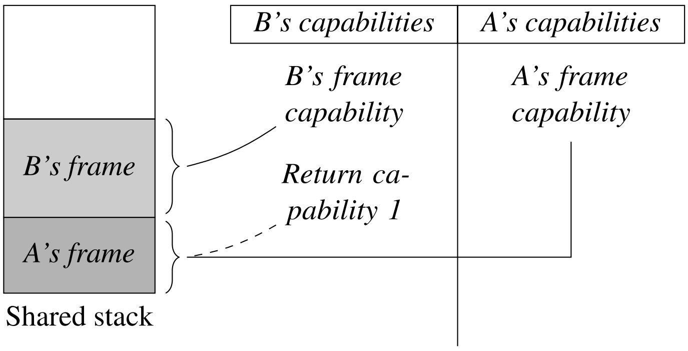
The core principles of capability safety remain the same across different levels of abstraction, but the sheer scale and complexity of low-level architectures can make characterizing capability safety on capability machines a challenge. By using Sail [14], a language for defining ISA semantics with the ability to generate definitions in various proof assistants, Nienhuis et al. [109] formalize the CHERI instruction set architecture, and define the key security properties behind CHERI's design, and prove in Isabelle that they hold. One key property, *reachable capability monotonicity*, states that, until execution switches to a new domain, the set of reachable capabilities does not increase. The set of reachable capabilities includes those accessible from the current register state, those that can be accessed via a series of load operations, and those reachable from a sealed capability that the current domain has the ability to unseal. Another key property characterizes the isolation properties granted by a setup that uses capabilities to implement compartmentalization. *Compartment isolation* states that the enclosed and isolated state of a compartment remains isolated during the execution of another compartment. Similar characterizations of capability safety are applied and proved for the Morello system [18], establishing very high confidence on the overall architecture design of ARM's capability machine.

Nienhuis et al. [109] prove an architectural property, and do not consider proofs of safety of individual programs. As such, the proven security properties do not attempt to characterize the intricacies of multiple interwoven domain changes, which is a key focus of this dissertation. Instead, reachable capability monotonicity describes monotonicity of intra-compartment executions, and compartment isolation is limited to the coarse-grained isolation guarantees of resources that domains do not share with the context. However, many applications depend on the specific behavior of shared resources across multiple domain changes. One particularly interesting application is a compiler.

Compilers typically implement domain changes through a calling convention by creating and managing a call-stack. However, CheriBSD implements domain changes as system call operations that manage a trusted call-stack in kernel space, with each compartment owning a per-compartment stack for intra-compartment calls. It is unclear whether this approach can scale to a large number of compartments, in which every cross-compartment call must pass through the kernel, and whether it robustly enforces the expected call/return disciplines. On the other hand, a cross-compartment shared call-stack, managed in user space by a compiler's calling convention, presents its own set of challenges for enforcing secure domain changes.

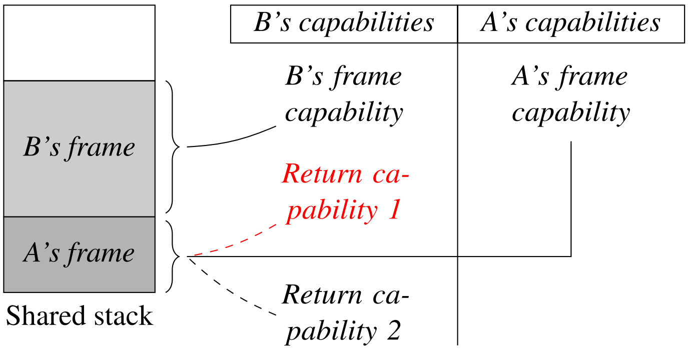
In his dissertation, Skorstengaard [128] identifies the challenges of enforcing two key stack related security properties, local state encapsulation and well-bracketed control flow, and proposes two calling convention implementations, the latter of which depend on a proposed kind of hardware capability, that robustly enforce them. A core difficulty of enforcing stack safety comes back to Lampson [87], who observed that a return must be distinguished from an entry, as there must be some mechanism in place to *revoke* access to a return capability, and avoid it is stored for later use, breaking well-bracketed control flow.

To understand the challenge more concretely, let's imagine a scenario with two different domains, and a sequence of transfers between them. Domain A calls twice into domain B by successively invoking B's entry point capabilities. At the first call, the subroutine in charge of domain transfers, which here takes the form of a calling convention implemented by a compiler, pushes a new stack frame unto the call-stack, and creates a back-link to A's stack frame. Domain B must be prevented from directly accessing A's stack frame. As such, the back-link is a closure capability around a continuation to A's code and its stack frame, which will henceforth be referred to as the return capability. Note that the return capability is created dynamically, that it is added to B's available capabilities, and that domain B returns to A by invoking it. Below we sketch a high-level representation of the first call in the scenario. In particular, we depict the capabilities owned respectively by B and by A after control has been transferred to B.



Upon return, B's stack frame is popped, and A continues executing. Next, domain A calls B for a second time; the calling convention thus allocates a new stack frame for B, and creates a new return capability for returning from the second call. Note that while B's second stack frame is conceptually different from its first, they share the same space on the call-stack.

The challenge arises in controlling B's access to the first return capability. Without a revocation mechanism, B still has access to the return capability from the first call. As a result, B can break well-bracketed control flow by invoking it.



In essence, a secure calling convention must enforce specific lifetime guarantees on its return capabilities, since ownership of a stack frame and associated return capability is inherently *temporary*.

In principle, local capabilities are meant to capture this temporary authority. However, Skorstengaard et al. [129] show that temporary ownership of local capabilities can only be guaranteed with significant overhead. As Georges et al. [55] (Chapter 3) puts it; "if a local capability must be revoked before a second invocation of a compartment, the calling convention must make sure not to accidentally leak an old copy of the capability. While local capability rules ensure that such old copies can never end up in heap memory (because no write-local capabilities to heap memory exist), they may still be present in any location where the adversary was previously able to store them: capability registers, but also any region of memory which it had a write-local memory capability for, such as the shared call-stack. Practically, the only way accidental leaking can be avoided is by clearing unused registers and sweeping over this write-local memory to clear it entirely or at least erase local capabilities. In their secure calling convention built on local capabilities, Skorstengaard et al. [129] have to clear the entire unused part of the stack before any invocation of adversarial code. This requirement is very costly in practice, and also hard to avoid, since the stack must be made write-local if we want to allow invoked code to spill registers or store local capabilities away during sub-invocations. The performance impact might be mitigated with special hardware support [72], but it is unclear whether this is enough to make it realistic for practical use."

In later work, Skorstengaard et al. [132] address this issue by proposing a new kind of *linear* capability, used to securely guarantee local state encapsulation and well-bracketed control flow in a highly efficient calling convention with no stack clearing. As the name indicates, a linear capability is guaranteed to possess no alias; operations that move a linear capability from register to memory, or from one register to another, make sure to remove the copy from the source. Unfortunately, to uphold linearity, such operations must be performed atomically. As a result, existing optimizations employed by CHERI are not applicable, and it is unclear whether they can efficiently implemented in practice.

Throughout his work, Skorstengaard [128] formalizes a simple but expressive capability machine, and proves that the proposed calling conventions enforce local state encapsulation and well-bracketed control flow. The formalization builds on prior work for capturing object capability patterns [37], now applied to a low level capability machine. Since the goal is to create a model rich enough to reason about well-bracketed calls, Skorstengaard et al. [129] define a step-indexed Kripke logical relation that uses a variant of Dreyer et al. [39]'s public and private future worlds [39] to express the special lifetime properties of local capabilities [129]. While their early work presents a model that implicitly captures local state encapsulation and well-bracketed control flow, insofar as it can be used to prove the robust safety of examples that depend on these properties, Skorstengaard et al. [132] later go on to completely characterize stack safety using a novel method that they call an *overlay semantics* [132]. An overlay semantics defines the operational semantics of the machine language in question, overlaid with an abstract notion of a call-stack, and special semantics for calls and returns to manipulate it. The key idea is to define an overlay that inherently and obviously captures the desired stack behaviors, and to then prove that the overlay semantics is fully abstract with respect to the original semantics, in which the main meaningful translation takes the abstract call and return instructions, and map them to the calling convention subroutines that implement them. Thus, Skorstengaard et al. [132] propose a novel method for characterizing security properties of capability machines, and in so doing, contribute to the characterization of stack safety, as the combination of local state encapsulation and well-bracketed control flow.

Building on the groundwork laid by Skorstengaard [128], this dissertation takes the work further through two significant additions. First, it addresses the technical impracticality of linear capabilities by proposing alternative capability designs that stay faithful to the design choices of CHERI capabilities, but retain some of the properties granted by linear capabilities. Second, while the correctness of Skorstengaard [128]'s contributions are established in impressively rigorous proofs, the formalization is entirely done on paper. A key contribution of this dissertation is a fully mechanized formalization of multiple capability machines, each with a mechanized model capturing its deeper semantic properties, resulting in rich but practical frameworks to prove robust safety of low level machine programs.

1.4 Mechanized Reasoning about Hardware Architectures

Throughout this dissertation, we will present multiple mechanized frameworks for reasoning about security properties of low-level machine programs. There is a long history of formalizing low-level machines. We here discuss a (non-comprehensive) selection of such work, that together display a range of different approaches.

XCAP [107] is a framework for defining second-order Hoare-logics for proof carrying code systems of low level assembly languages with embedded code pointers, i.e. code pointers as data. When applying the framework, the resulting Hoare logic can be used to reason about certified assembly programs. XCAP builds on previous iterations of the framework; CAP (Certified Assembly Programming) [165], CCAP (Concurrent CAP) [164] which supports concurrency, and CMAP (Certified Multithreaded Assembly Programming) [51], which builds on CCAP but with the added functionality of code sharing and dynamic thread creation and termination. Like its predecessors, XCAP's target machine is an idealized and abstract low level machine, that remains expressive enough to capture the challenges of verifying assembly code, but simple enough to be tractable (for instance, code and data do not live in the same address space). Ni et al. [108] later apply XCAP to a more realistic x86 machine model, and use the resulting Hoare logic to certify a context management system. The x86 machine is realistic, but remains a subset of the full instruction set architecture, focusing on the parts needed to define a realistic context management system. Unlike the previously mentioned idealized assembly machines, the x86 machine model uses a continuous memory address space storing both code and data, and covers interesting x86 features such as variable instruction encoding length. Additionally, Ni et al. [108] build an abstraction layer for calls and returns, which abstracts away the underlying calling convention (implemented in x86), making reasoning about calls and returns more tractable.

The Certified Assembly Programming family of frameworks enable the mechanized verification of low level assembly programs. However, specifications must subsequently be proved manually, and can quickly become complex, and hard to maintain against the natural evolution of software systems. Chlipala [30] thus presents Bedrock, a separation logic framework, implemented in Coq, that supports the automated verification of low-level programs. Languages supported by Bedrock are low-level, insofar as they can include registers, linear memory, and a jump (GOTO) instruction. However, they remain sufficiently abstract to facilitate programmer friendly implementations, by including syntax for structured control flow, such as basic blocks and while loops. As such, Bedrock is a flexible framework, that is applicable at varying levels of abstraction.

CertiKOS [63] is an extensible architecture for building certified concurrent OS kernels. Using CertiKOS, Gu et al. [63] develop a concurrent OS kernel, mC2, on top of an x86 multicore machine with support for fine-grained locking, all fully certified in the Coq proof assistant. The mC2 kernel is proved to contextually refine a high-level functional specification, similar to those presented in [62], which in turn implies the safety and termination of all system calls and traps, and the guarantee that all high level specifications proved about a program also hold when that program runs on the kernel.

A core design philosophy of CertiKOS is to enable a compositional approach to OS kernel verification, with a particular emphasis on distinguishing each *abstraction layer* implemented by an operating system. In essence, each component of an OS kernel ought to be verified at the layer it lives in. A similar methodology was used in [149], in which a virtual memory management unit, a specific component of an operating system, is compositionally verified by distinguishing between its layers of abstraction, and proving a series of contextual refinements. In the case of mC2, the bottommost layer is a x86 multicore machine, and CertiKOS thus models x86 assembly. Specifically, CertiKOS covers a subset of the full x86 instruction set, limited to the instructions needed to implement mC2.

Since compilers typically target low level assembly languages, verifying a realistic compiler involves defining a model of possible target machine languages. CompCert [89, 90], a verified C compiler, and CakeML [86], a verified ML compiler, are two significant high-profile verified compilers, that target low level assembly languages.

Modern iterations of CompCert can compile Clight (a subset of C) into a variety of assembly backends, including RISC-V, ARM and x86. Each backend is faithfully defined in Coq, albeit with some abstractions; the semantics of instructions not needed by the compiler are occasionally left undefined, and, more importantly, the formalized backends include pseudo-instructions such as `Pallocframe` and `Pfreeframe`, which abstract away certain operations that are not native to machine languages. Likewise, the memory model of CompCert assumes an infinite memory, split into a series of blocks. The elimination of pseudo-instructions is subsequently done in a machine code generation phase, which takes CompCert's assembly and generates real executable machine code. CompCert has since then been extended to additionally verify various machine code generation phases; Stack-Aware CompCert [152] extends CompCert with an explicit and finite stack, and is used to develop CompCertMC, a fully verified compiler down to a low-level language without pseudo-instructions, and with a flattened memory model, unlike CompCert's block based memory model. Programs written in this new target language is thus much closer to real native code. Wang et al. [152] demonstrate how to convert the result into executable assembly by using RockSalt [101] to generate the final x86 machine code. Later, CompCertELF [153] completes the compilation chain, by verifying the final compilation phase which compiles the low level machine code to the standard object file format ELF. The result is a fully verified compiler from Clight to ELF on a 32-bit x86 architecture.

CakeML is a bootstrapping compiler written in 64-bit x86 machine code, and verified in the HOL4 theorem prover. The target language of the compiler, CakeML Bytecode, is a low-level assembly language with a single abstracted stack. CakeML Bytecode is designed to be sufficiently abstract to ease the implementation and verification of the compiler, while being sufficiently close to x86 assembly to enable straight-forward code generation. Useful abstraction is achieved by structuring stack data into blocks, and avoiding any notion of heap pointers, while realism is achieved by making sure each operation map to one or two real x86 instructions. As a result, the machine language is modeled with an abstract distinction between code space and the stack, and abstract instructions for calls and returns, and other stack related manipulations.

As mentioned above, Morrisett et al. [101] define a model of a subset of the x86 instruction set architecture in Coq, and use it to create RockSalt, a verifier that checks that code binaries respect a sandbox policy, similar to that of Google's Native Client (NaCl). The model of x86 specifies the semantics of over 70 instructions, and is validated against existing x86 implementations. To facilitate reasoning about the famously complex x86 semantics, RockSalt defines two domain specific languages for specifying instruction semantics, thus lifting the semantics to a higher-level representation. Kennedy et al. [79] define a shallow embedding of x86 semantics in Coq, which is used to create an assembler entirely in Coq. Both works model only subsets of the x86 ISA. Meanwhile, Dasgupta et al. [35] define a *complete* and *faithful* formal semantics of all sequential x86-64 user-level instructions in the K framework, using a combination of automated cross-checked translation of existing formalizations, and manual specification of remaining instructions. They leave a model of x86's concurrency, including its relaxed memory model, for future work.

Thus far, we can observe a general trend, in which each new application leads to the creation of a new model of some low level machine. This is somewhat expected, since many of the aforementioned models are subsets of the machines they represent, exactly suited to fit the needs of the system in question. However, this raises the question, what should a universally applicable model look like? Sail [13, 14] addresses this challenge, with a more general and comprehensive approach at mechanizing low level assembly. Sail is a domain specific language for instruction set architecture (ISA) semantics, that supports automatic generation of emulator code in C and OCaml, as well as proof-assistant definitions for Isabelle, HOL4 and Coq. Using Sail, Armstrong et al. [14] mechanize semantic models for large parts of ARMv8-A, RISC-V, MIPS and CHERI-MIPS architectures, and use the generated theorem prover definitions to prove functional correctness of ARMv8-A address translation. Later, Nienhuis et al. [109] use Sail to formalize key security properties of the CHERI-MIPS architecture. A core goal is to aim for realistic, complete and executable ISA definitions.

Thus far, we can notice a range of different approaches. One notable through line in many of these formalizations, are various high-level abstractions that facilitate proofs of safety of low-level programs, such as structured control flow, an idealized call-stack, block-like memory models, and a distinction between code and data.

The formalizations presented in this dissertation will purposefully *avoid* these exact abstractions, since our goal is to explore how low a level machine securely *implements them*.

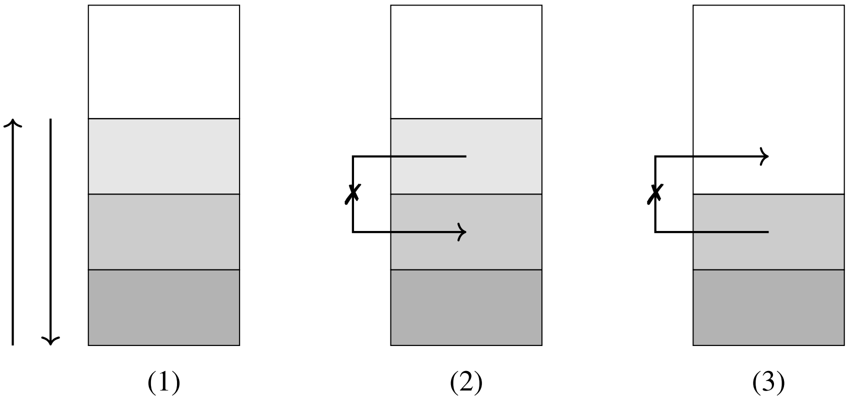
Nevertheless, the frameworks presented in this dissertation will, for the most part, be subsets of the architecture in question, with certain details abstracted away, such as the encoding of capabilities as a series of bytes. These simplifications allow us to reason about deeper semantic properties of the language, which can be difficult to accomplish for the highly detailed Sail mechanizations. In recent work, Islaris [122] bridges the gap between real-world ISA specifications and the verification of rich semantic properties, by taking advantage of sophisticated automation techniques to generate Iris-based program logics for a given ISA. This opens up future possibilities of scaling up the frameworks presented in this dissertation to realistic representations of capability machine architectures.

1.5 Contributions and Structure

The challenges of implementing domain transfers in capability-enabled security systems were identified early [87], and have since then been tackled from various perspectives. CheriBSD implements domain transfers as system calls, using an internal call-stack managed by the kernel. However, such an approach limits its applicability to machines that run an operating system with similar system operations for calls and returns. One the other hand, a compiler typically implement domain transfers via an implemented call-stack, which, unlike CheriBSD, is managed in user-space rather than kernel-space. However, Skorstengaard [128] identifies multiple vulnerabilities when dealing with a shared call-stack in the presence of untrusted code. Unfortunately, Skorstengaard [128]'s proposed solutions are arguably not feasible in practice; a secure calling convention with local capabilities leads to a significant overhead (full stack clearing at every call). On the other hand, while linear capabilities enable a calling convention with much less overhead, they cannot be efficiently implemented in hardware.

We propose two simple designs that grant all the security of previous proposals, while being sufficiently faithful to the design of CHERI capabilities to justify a practical hardware implementation. While the designs are simple, the arguments behind the security of the proposed calling conventions are subtle. We formalize each proposed capability design and associated calling conventions, and prove that they enforce secure domain transfers. The formalized capability machines models build on prior work [128], but are for the first time fully mechanized. The mechanized frameworks, used to reason about security properties of capability machines, serve as significant contributions of this dissertation. Finally, in anticipation of secure compilers that implement the proposed calling conventions, we formalize the full WebAssembly 1.0 standard in a framework to reason about WebAssembly's encapsulation properties, setting up the infrastructure for future work on a capability machine backend to the rapidly growing WebAssembly ecosystem.

Throughout this dissertation, we show how new capability designs can lead to efficient and secure calling conventions. These calling conventions implement secure domain transfers by managing a call-stack, which must behave exactly as expected, even when shared between different domains. The calling conventions must thus robustly enforce *Stack safety*, a property that can intuitively be understood as the following three disciplines (each of which are often taken for granted in high-level programming languages):



1. *Well-bracketed control flow*: stack frame are pushed and popped in first-in-last-out order

2. *Spatial stack safety*: the stack frame of the currently executing process cannot grant access to lower stack frames

3. *Temporal stack safety*: the stack frame of the currently executing process cannot grant access to the previously active, now popped, stack frames of returned calls

In summary, this dissertation makes the following contributions:

New Capability Designs We propose two new capability designs: Chapter 3 presents *uninitialized capabilities*, an extension of CHERI permissions which enable the efficient enforcement of spatial stack safety, and Chapter 4 *directed capabilities*, a new locality that enable the efficient enforcement of temporal stack safety. Each new design can be implemented with 1 additional bit of representation, with semantics that are similar to existing CHERI capability operations. More precisely, the new designs apply similar arithmetic bounds checks to existing operations, and allow for similar optimization patterns such as parallelizing the execution of the new operations.

Efficient Calling Conventions By using combinations of uninitialized and directed capabilities, we propose two new calling conventions which enforce the stack safety properties outlined above. The first calling convention, presented in Chapter 3, uses uninitialized capabilities to enforce well-bracketed control flow and spatial stack safety. While temporal stack safety is not enforced, vulnerabilities are mitigated by clearing stack frames before returning. However, this clearing is entirely avoided by using both uninitialized and directed capabilities, in a calling convention that additionally enforces temporal stack safety, presented in Chapter 4.

Three Mechanized Formalizations The new capability designs and associated calling conventions are rigorously formalized in mechanized frameworks to reason about security properties of capability machines. Section 1.4 presented a selection of mechanized formalizations of low level machines. To facilitate the formalization and reasoning of low-level code, many of the various formalizations depict subsets of instruction set architectures, arguably at higher levels of abstraction than the real machines (e.g. structure control flow, distinction between code and data, idealized call-stack, etc.). The frameworks presented in this dissertation similarly only capture a fraction of a real CHERI-like machine, and abstracts away details such as the hardware representation of capabilities as a sequence of bytes. Nonetheless, they maintain a low-level representation of key security related features of a CHERI-like capability machine. For example, our formalizations make no distinction between code and data, they define a flattened and finite memory model, use completely unstructured control flow, and do not depict an idealized stack. Instead, domain transfers and stack safety is implemented and maintained via the proposed calling conventions.

Chapter 2 presents Cerise, a capability machine model to reason about the security properties of a capability machine without locality bits. Cerise models capability safety, and the encapsulation properties enabled by capabilities. Chapters 3 and 4 present two version of Stack-Cerise, one with support for uninitialized capabilities, the other with support for uninitialized and directed capabilities. Each are used to prove that the intended stack safety properties are indeed enforced by the proposed calling conventions. Each framework provides a separation logic to reason about capability machine programs, an ideal logic to reason about capability safety, and a logical relations model to reason about their security properties. All three frameworks are defined in Iris, a higher-order separation logic frameworks [74, 75, 82], using the Iris interactive proof mode built in Coq [83].

Towards a Secure Compiler The mechanized capability machine frameworks establish safety and security properties of the proposed capability designs and associated calling conventions. Such calling conventions are eventually implemented by a compiler. The proposed calling conventions enforce high degrees of security, including properties that implicitly hold in certain high level languages, such as well-bracketed control flow and local state encapsulation. One such language is WebAssembly, a relatively small bytecode language with formally defined semantics, a type system, and an actively growing ecosystem. Its module system is designed to enforce various encapsulation properties. Furthermore, a recent proposal seeks to extend WebAssembly with support for fine-grained memory safety [95]. WebAssembly thus presents an interesting source language for a secure compiler targeting a capability machine. In Chapter 5, we explore the security properties enforced by WebAssembly, by implementing a language specification of the WebAssembly standard in the Iris logic. The formalization captures the isolation guarantees provided by WebAssembly modules, and gives a starting point to formally explore a compiler from WebAssembly to a capability machine.

1.5.1 Publications

This dissertation is comprised of the following publications and manuscripts, each accompanied by a Coq formalization.

[55] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, Lars Birkedal.
Efficient and Provable Local Capability Revocation using Uninitialized Capabilities
Proceedings of the ACM on Programming Languages (POPL), 2021, 5.

https://github.com/logsem/cerise-stack

[59] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, Lars Birkedal.
Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code
In Submission.

https://github.com/logsem/cerise

[57] Aïna Linn Georges, Alix Trieu, Lars Birkedal.
Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities
Proceedings of the ACM on Programming Languages (OOPSLA), 2022, 6.

https://github.com/logsem/cerise-stack-monotone

[115] Xiaojia Rao, Aïna Linn Georges, Conrad Watt, Maxime Legoupil, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal
Iris-Wasm: Robust and Modular Verification of WebAssembly Programs
Conditionally accepted at the International Conference on Programming Language Design and Implementation (PLDI) 2023

https://zenodo.org/record/7708441

1.6 Conclusion and Future Work

By proposing new capability designs we hope to inform the design of future capability machine architectures. Ultimately, the security of high level programming languages depends on the compiled backend that actually runs the code. The mechanized frameworks in this dissertation establishes a foundation for investigating secure compilers targeting capability machines.

Secure Compilation from WebAssembly to a Capability Machine WebAssembly is a portable low-level bytecode language, with a type system and runtime environment that enforces strict isolation guarantees between modules. It is sufficiently low level to efficiently compile to low-level executable assembly, while being sufficiently high level to enforce structured control flow and local state encapsulation. Furthermore, a growing number of compilers target WebAssembly, making it an intermediate representation within multiple compilation chain. The recent proposal to add memory safe pointers to WebAssembly [95] offers an excellent opportunity to investigate a secure compiler that targets a capability machine.

A secure compiler must not only guarantee that the produced code refines the source code, it must guarantee that the behavior *robustly* behaves like its source code counterpart, even when linked with arbitrary target code. Multiple criteria for defining secure compilation have been proposed [4]. A robustly safe compiler must guarantee that all safety properties that robustly hold on traces emitted by a source program must also robustly hold on the trace of the compiled target program [110, 111]. Both memory safety properties and control flow properties can be expressed in terms of trace-based safety properties. Implementing and proving the robustly safe compilation from WebAssembly to a capability machine is an ambitious, but interesting avenue of future work. Viable techniques for proving that a compiler targeting a capability machine is robustly safe have been proposed [44], but can get complicated by the presence of pointer passing [45]. Part of the complexity comes from reasoning about separation of partial heaps, and an interesting avenue of future work is to investigate whether using a separation logic facilitates the proof.

Cheri Backend to CompCert Another possibility is to build on an existing *verified* compiler, by extending it with a secure backend, and apply existing techniques to prove secure compilation that reuse verified compilation results [44]. An ideal candidate is CompCert, which could lay the foundation for the first realistic secure compiler from C to assembly. In an extended abstract, Thibault et al. [140] [140] propose to extend CompCert with a notion of compartments, and prove that it is a robustly safe compartmentalizing compiler. The secure compilation criteria includes a notion of dynamic compromise between mutually distrustful components, and is a criteria defined specifically for language with undefined behavior and compartments [3]. In essence, even if a component is compromised, it cannot compromise other components that are outside the reach of its interface. The same extended abstract proposes to add a CHERI backend to CompCert, to robustly enforce compartmentalization at the target level.

Scaling up the Capability Machine Formalization While the capability machines modeled in this dissertation express tricky features of low level languages, such as unstructured control flow, they remain small and idealized representations of real machines. On the other hand, the models describe deep semantic properties of capability machines, that could be interesting to reason about at scale, for example on the recent Morello architecture. By using techniques such as Islaris [122] and automation techniques for establishing ISA security guarantees [69], it should be possible to scale up the frameworks to realistic representations of CHERI, including features such as interrupts and a weak memory model.

Part II

Publications

Chapter 2

Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code

This chapter is an extended version of the following journal submission:

> Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, Lars Birkedal.
> *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code*

The extension consists of

- A section about a binary model to reason about confidentiality properties; Section 2.9

Abstract

A capability machine is a type of CPU allowing fine-grained privilege separation using *capabilities*, machine words that represent certain kinds of authority. We present a mathematical model and accompanying proof methods that can be used for formal verification of functional correctness of programs running on a capability machine, even when they invoke and are invoked by unknown (and possibly malicious) code. We use a program logic called Cerise for reasoning about known code, and an associated logical relation, for reasoning about unknown code. The logical relation formally captures the capability safety guarantees provided by the capability machine. The Cerise program logic, logical relation, and all the examples considered in the paper have been mechanized using the Iris program logic framework in the Coq proof assistant.

The methodology we present underlies recent work of the authors on formal reasoning about capability machines [55, 131, 135], but was left somewhat implicit in those publications. In this paper we present a pedagogical introduction

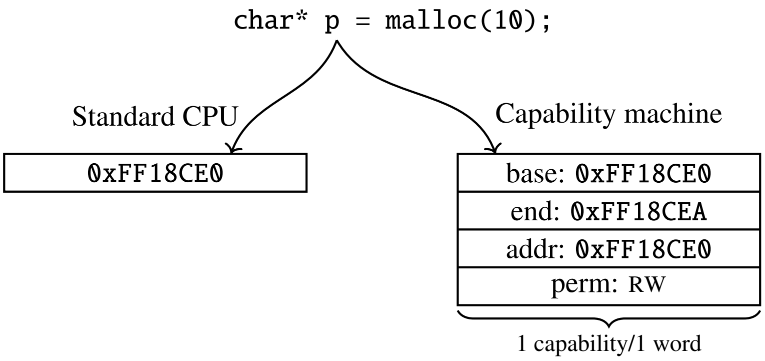


Figure 2.1: Representation of a pointer in a standard architecture vs. a capability machine. A capability is similar to a pointer with extra meta-data.

> to the methodology, in a simpler setting (no exotic capabilities), and starting from minimal examples. We work our way up to new results about a heap-based calling convention and implementations of sophisticated object-capability patterns of the kind previously studied for high-level languages with object-capabilities, demonstrating that the methodology scales to such reasoning.

2.1 Introduction

A capability machine is a type of CPU that enables fine-grained memory compartmentalization and privilege separation through the use of *capabilities*. This type of hardware architecture has been studied since the 60ies [36, 91], and in particular more recently as part of the CHERI project [158]. Capability machines offer fine-grained and scalable privilege separation at the hardware level and they are a compelling target for secure compilation [29, 44, 132, 147].

Capability machines distinguish, at the level of hardware, between machine integers and capabilities; and a capability can be understood as a pointer with associated metadata, cf. Fig 2.1. A machine word containing an integer value can only be used for numerical computations and cannot be used as a pointer to access memory. On the other hand, a machine word containing a capability can be used to access a given portion of memory, depending on the metadata contained in the capability. We also say that the capability *has authority over* some fragment of memory.

A capability thus corresponds to a native machine value, and can be stored in a CPU register or in memory. While this might seem wasteful due to the amount of extra metadata that needs to be carried around, for realistic capability machines a lot of work has been dedicated to the design of compressed representations for capabilities, see, e.g., [27, 162]. In this paper, we will abstract from these details and represent capabilities in their uncompressed form, as a tuple carrying the metadata.

A capability machine guarantees the integrity of capabilities: one cannot create fresh capabilities out of thin air or modify the metadata of existing capabilities in arbitrary ways. For instance, CHERI associates tags to machine words to identify whether they represent a capability or an integer. Such a tag bit is checked and set by the machine, and is not directly accessible by software. More generally, new capabilities can only be derived from existing capabilities using a restricted set of operations provided by the machine. As such, all capabilities on the system are recursively derived from the full-authority capabilities that are initially provided to software at boot time. Intuitively, the machine ensures that a given program cannot forge capabilities and obtain more authority than it held previously, a property sometimes referred to as capability monotonicity [109].

Capabilities therefore allow a piece of code to interact securely with untrusted third-party code, even within the same address space, by restricting the set of capabilities the untrusted code (transitively) has access to. In a system composed of mutually untrusted components (which might even contain malicious code), capabilities provide a way of enforcing that the overall system nevertheless satisfies some security properties.

Note, however, that capabilities are low-level, flexible, building blocks, which operate at the level of the machine code and whose metadata "just" triggers some additional runtime checks by the machine. This means that the *properties* we can actually enforce using capabilities crucially depend on how we *use* capabilities: the variety of properties that can be enforced stems from how one can use and combine capabilities.

In this paper we show how we can formally *prove* that security properties are enforced for some known verified code, *even when* that code is linked with unverified untrusted third-party code. Our model of interaction between the known and unknown code is very simple: we assume the code is in the same address space and that control is transferred from one to the other using an ordinary jump instruction. We focus on a restricted subset of the capabilities present in the CHERI architecture (using only "normal" read/write capabilities and a kind of so-called sentry capabilities, which provide a basic form of encapsulation, see Section 2.2.4). Because the security properties we consider hold even in the presence of unverified unknown code, they are sometimes referred to as *robust safety* properties [138]. The security properties we focus on are centered around memory compartmentalization, in particular, local state encapsulation. We consider a range of examples, starting with very basic examples (sharing a buffer with some unknown code), through implementations of closures with encapsulated state, and end up with a quite sophisticated low-level implementation of an interval library, for which we show that certain representation invariants are preserved, even when interacting with unknown code.

We proceed as follows:

- We first explain informally how one can program with capabilities and use capabilities to enforce memory compartmentalization (Section 2.2).

- We then introduce the formal operational semantics of a simple capability machine with sentry capabilities (Section 2.3).

- We define the Cerise program logic which can be used to formally verify the correctness of programs running on the capability machine. Our program logic is defined by instantiating the Iris framework [77], which provides an expressive separation logic with powerful reasoning principles, including, in particular, the notion of a *logical invariant* (Section 2.4).

- We define, using our program logic, the specification of what a "safe" capability and a "safe" program is. Intuitively, a capability (respectively, a program) is "safe" if it cannot be used to invalidate an invariant at the logical level. Hence, safe capabilities can be shared freely with unknown code. Safety of a capability is defined in the program logic as a unary logical relation (Section 2.5).

- We show that if a program only has access to "safe" values, then running the program itself is also "safe". This is a global property of the capability machine, expressing its capability safety: it is not possible to increase one's authority beyond what was available initially, independently of the sequence of instructions that one executes (Section 2.5). Concretely, the theorem takes the form of a contract that holds for arbitrary code,[1] and which can be combined in the program logic with manual proofs for trusted code. The last piece of the puzzle is then a so-called Adequacy theorem (Section 2.4), which relates invariants established in the program logic to the operational semantics of the machine. Given a concrete scenario (typically, a complete system mixing known verified code with unknown untrusted code), this makes it possible to obtain a theorem about the execution of the system which only depends on the operational semantics of the machine (not on the program logic).

- In Section 2.6 we then return to the examples from Section 2.2 and show how to use Cerise to formally prove that the desired memory compartmentalization results really do hold.

- In Section 2.7 we consider more sophisticated examples, which involve dynamic memory allocation. We focus on the low-level implementation of ML-like programs, and introduce a heap-based calling convention for closures implementing ML functions. We extend the earlier Adequacy theorem to account for dynamically allocated memory.

- In Section 2.8 we demonstrate how to use our methodology to establish correctness of object capability patterns (OCPs) from the literature. In particular, we consider the OCP of dynamic sealing, as presented by [138] in the context of a high-level language and we demonstrate that Cerise can be used to prove similar results about a low-level implementation of their example.

- In Section 2.9 we outline how Cerise can be used to construct a binary model to reason about confidentiality properties of object capability patterns.

[1] Because it holds for arbitrary code, we sometimes refer to this as a *universal contract*.

(a) Scenario 1: passing control to untrusted code

(b) Scenario 2: being called by untrusted code (possibly many times)

Figure 2.2: Two scenarios where a (trusted) component interacts with its (untrusted) context. The trusted component is represented with a plain background, while the untrusted context is represented with a red dotted background.

- Section 2.10 offers some perspectives on the relevance of our technical contributions and how we envision them being used in the development of secure systems.

- Finally, we discuss related work in Section 2.11.

This paper pedagogically introduces and explains the methodology underlying a sequence of recent research papers [55, 130, 131, 135], in the form of the Cerise program logic, but also contributes new material. The operational semantics, program logic and logical relation discussed in Sections 2.3, 2.4 and 2.5 are based on those used by [55] (but we have removed local and uninitialized capabilities as well as Kripke indexing for simplicity and instead added much more extensive explanations and proofs). Sections 2.2 and 2.6 are new; they provide a clear and accessible introduction to capability machine programming and our reasoning tools. The examples in Sections 2.7-2.8 are also new and represent a non-trivial verification effort.

The results and examples presented here have been fully formalized in Coq, and are available online: `https://github.com/logsem/cerise`. The development can also be viewed online at `https://logsem.github.io/cerise/journal/`; we use circled numbers such as ① to link directly to corresponding Coq formal statements in the following.

2.2 Programming with capabilities

Let us give a taste of how one might use capabilities when writing programs with the goal of enforcing some additional memory protection or encapsulation guarantees. We consider a fairly simple but quite general adversarial model, where we wish to verify the correctness of a *known component* interacting with a possibly adversarial *third-party component* whose code is unverified and untrusted.

In this section we detail two concrete example programs, which use capabilities in two different scenarios. In the first scenario, illustrated in Figure 2.2a, we consider a program that eventually passes control to the untrusted third-party code, but uses capabilities to protect a region of memory containing some secret data from being accessed by the untrusted code. In the second scenario (Figure 2.2b), we consider the case of a verified component being called by the third-party code. The goal is then for the verified component to use capabilities to protect (or “encapsulate”) a piece of private memory, which it may access during its execution, but which should remain inaccessible to the unverified code.

2.2.1 Anatomy of a capability (in our model)

We are interested in a subset of the capabilities available in a CHERI capability machine. We thus work with a simplified machine model, featuring basic capabilities that are used to give access to a range of memory, as well as so-called “sealed entry” capabilities (abbreviated as “sentry” capabilities [158, §3.8]) that provide encapsulation features. The sentry capabilities were also called “enter” capabilities in earlier work, e.g., in the M-Machine by [27].

Concretely, we model capabilities as 4-tuples (p, b, e, a). In actual hardware, capabilities are encoded as fixed-size binary words, but here we abstract over their concrete representation.

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{Capability: } (p, b, e, a)} \\
\quad p \in \{\mathrm{O}, \mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}, \mathrm{E}\} & \text{permission} \\
\quad b \in \mathit{Addr} & \text{base address} \\
\quad e \in \mathit{Addr} & \text{end address} \\
\quad a \in \mathit{Addr} & \text{current address}
\end{array}$$

A capability (p, b, e, a) represents a machine word that can be used to access memory within the region $[b, e)$ delimited by its base address b and end address e. The permission p specifies what is possible to do within this memory range: permission O specifies that the capability actually gives no access rights, RO grants read-only access to memory, RX grants the right to read and execute the contents of the memory, RW gives read and write access, and RWX gives read, write, and execute access. Capabilities with permission E behave a bit differently (they are used to provide a form of encapsulation), and will be explained later in Section 2.2.4.

A capability is meant to be used as a pointer, and thus additionally points to a specific address a (typically, but not necessarily, belonging to the range $[b, e)$). Each time the capability is used to access memory, the machine will automatically check that a is between bounds b and e, and that the access is permitted according to p. From a capability (p, b, e, a) it is easy to derive another capability (p, b, e, a') pointing to a different address a' also within range $[b, e)$ – in other words, while a capability points to a specific address, it really holds authority over the whole region delimited by its beginning and end address.

Note that, on a capability machine, machine words can represent not only binary-encoded capabilities, but also traditional fixed-size integers. However, unlike on a traditional computer architecture, integers cannot be used as pointers. In other words, without holding a capability, one cannot access memory at all. In this paper, we rely on difference in notation to distinguish between capabilities and integers. In actual hardware, this is done by associating an extra one-bit tag to each word to distinguish capabilities from integers.

2.2.2 Sometimes, failure is a good thing

It is worth pointing out a sometimes counter-intuitive aspect of reasoning about security of programs running on a capability machine, especially for readers with a background in reasoning about safety in higher-level languages. For a high-level language, program safety can be seen as the absence of undefined behavior or runtime errors. For instance, an out-of-bounds array access is undefined behavior in C, and it leads to a runtime error, such as raising an exception, in memory-safe languages such as Rust or OCaml. We are instead interested in *security* properties for which a runtime failure can actually be considered a good thing.

Generally speaking, a low-level machine has many cases where it can fail at runtime, stopping the normal course of execution. In a standard (non-capability) machine, this can happen, e.g., if the machine attempts to execute an invalid instruction which cannot be decoded. The addition of capabilities only adds more possibilities for runtime faults: each time a capability is used, the capability machine will check that it has adequate permission and bounds, and raise a runtime fault otherwise.

Now, the point is that, from a security perspective, these additional runtime faults are a good thing. Using these additional checks, the capability machine turns dangerous behavior (out-of-bounds accesses leading to buffer overflow attacks, etc.) into proper faults before they can cause damage. Thus, for our purposes, it is always safe for the machine to fail: it means that an illegal operation may have been attempted, and the execution has been stopped in response.

Of course, when writing concrete programs, we will typically want to verify that we do not involuntarily trigger faults, as this would make our programs less useful. But when interacting with adversarial code, this is a possibility that we have to take into account anyway: we cannot prevent unknown code from shooting itself in the foot, e.g. by trying to access memory it does not have a valid capability for, or by decoding illegal instructions.

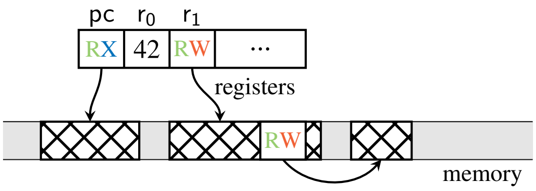
To sum up, in this work we reason about security properties that are not violated in the case of machine failure. This includes, for example, integrity of private data: no data can be compromised if the machine stops running. It is therefore useful to keep in mind that we consider failure to be trivially safe!

2.2.3 Restricting access to memory by constraining available capabilities

Consider Scenario 1 from Figure 2.2a: how can one write a program which passes control to untrusted code while protecting some secret data? That is, we wish to write a program that sets up capabilities so that its secrets are preserved even after it runs untrusted code.

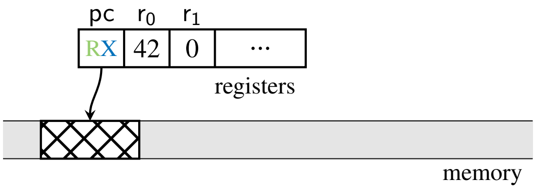
The key intuition is that, at any point of the execution, one can only access the part of memory that is accessible using the currently available capabilities. In other words, the authority of a running program comes from the set of capabilities which are transitively reachable from the CPU registers.

This is illustrated below, in a scenario where the pc register ("program counter") contains a capability with permission RX pointing to some memory region (containing the code of the program being executed), and register r_1 contains a capability with permission RW, pointing to a region of memory, which itself contains a RW capability pointing to another memory region. The collection of the "hatched" memory regions corresponds to the overall subset of memory accessible by the program.



If one wishes to reduce the set of available memory or its associated access rights—for instance to protect secrets from being leaked to an adversary—then it is be enough to constrain the capabilities currently available. This can be done in a few different ways.

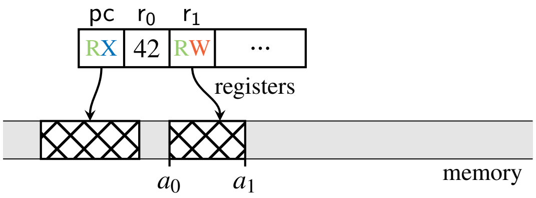
First, one can simply remove a capability from registers in order to remove access to the memory it was giving access to. For instance, after executing the instruction "`mov r1 0`", which overwrites the contents of register r_1 with the integer 0, one loses access to the memory regions which were previously accessible from the capability stored in that register.



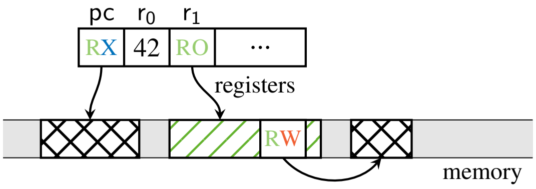
Alternatively, it is possible to restrict the range of a capability to point to a smaller memory region. This changes the set of accessible memory to a subset of what was previously available. For instance, starting from our initial scenario and running the instruction "`subseg r1 a0 a1`" will change the range of the capability stored in register r_1 to $[a_0, a_1)$. (The machine will check that $[a_0, a_1)$ is indeed included in the range of the original capability.) In our example scenario (illustrated below), we then only keep the beginning of the region accessible from r_1, and this entails that the third region of memory becomes inaccessible, since it was only reachable from a capability stored at the end of the region accessible from r_1.

```
; initially, PC = (RWX, code, end, code)
;             r0 = (unknown) pointer to the continuation
code:
  mov r1 PC                   ; r1 = (RWX, code, end, code)
  lea r1 [data-code]          ; r1 = (RWX, code, end, data)
  subseg r1 [data] [data+3]   ; r1 = (RWX, data, data+3, data)
  jmp r0                      ; jump to unknown code: we give it
                              ; read-write access to the first 3
                              ; words of the data,
                              ; but not the secret value
data:
  ; the first 3 data words contain public data that will be passed
  ; to the unknown code (the "Hi" string)
  'H', 'i', 0,
  ; they are followed by secret data (the integer 42)
  42
end:
```

Figure 2.3: Program sharing a buffer with possibly adversarial code.



Finally, one can restrict the permission of a capability to a permission that grants less access rights. For instance, running the instruction “`restrict r1 RO`” in our initial scenario modifies the capability stored in r_1 to only grant read-only access to its corresponding memory region. Note that we still have read-write access to the last memory region, as we can still read the capability (with permission RW) pointing to it.



Example: sharing a sub-buffer with unknown code Using some of the mechanisms detailed above, we can implement a very simple program that shares a buffer with unknown, possibly adversarial, code while using capabilities to protect some data that would otherwise be vulnerable to buffer overflow attacks.

The assembly code for the program is shown in Figure 2.3. It consists of a code section containing the instructions of the program, followed by some data which (for simplicity) we simply assume to be statically allocated. The data section holds the zero-terminated string `"Hi"`, which we wish to share with the untrusted code, and the integer 42 which represents our secret data.

Initially, we assume the program counter to contain a RWX capability for the whole region holding our program. This capability serves two purposes: it allows the machine to execute our program, but can also be manipulated by the program itself to derive a capability pointing to its own data. By convention, the register r_0 is assumed to contain a pointer to the continuation of the program, i.e. other code that the program will pass control to after it is done executing. As no assumption is made about the contents of r_0, it is conservatively assumed to point to unknown, arbitrary code.

Our program executes as follows: it first loads the capability held by the program counter into register r_1. Then, using the `lea` instruction, it changes the "current address" of the capability to point to the `data` label (`lea` modifies a capability by adding an offset to its "current address"). In assembly programs, we use the brackets notation `[...]` to denote an arithmetic expression that is computed statically when assembling the program.

At this point, the capability held in r_1 points to the start of the `"Hi"` string, but has (RWX) authority over the whole code and data section. This capability would be unsafe to share with the untrusted code, as they could simply use `lea` to increment the capability's current address past the end of the string, and obtain a valid capability to the secret value (thus performing a basic "buffer overflow" attack). To prevent this from happening, we use the `subseg` instruction to obtain a capability whose range of authority is restricted to the sub-buffer holding the `"Hi"` string. Finally, we pass control to the untrusted code by using the `jmp` instruction, loading the contents of register r_0 into pc.

This example illustrates that even a basic mode of use of capabilities (restricting them appropriately) can easily prevent buffer overflow attacks. In Section 2.6.1, we show how we can formally prove that, for any untrusted code, the value of the secret data will be equal to 42 at every step of the execution, including after control has been passed to the untrusted code. We have also developed a relational model, which can be used to prove that the secret value cannot even be read by the unknown code, but the details of this relational model are out of scope of this paper.

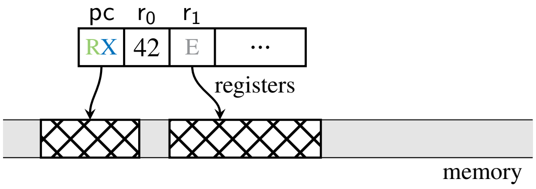
2.2.4 Securely encapsulating code and private capabilities

The previous example illustrates how to restrict available capabilities to prevent an adversary from accessing secret data. However, what if we additionally want our program to be called back by the untrusted code, as in Scenario 2.2b? In that case, when the trusted code gets invoked again we would like to recover access to the capabilities it previously had to its private state.

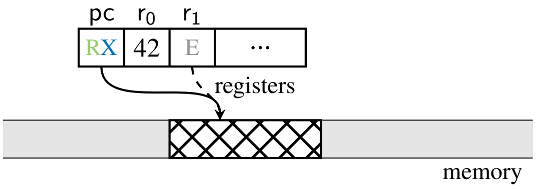
This is unfortunately not achievable with the capabilities that we have described so far. If we remove capabilities to private memory before passing control to untrusted code, then there is no way for us to get them back later on: the only capabilities we will get access to in a further invocation are capabilities the untrusted code itself has access to.

Sentry capabilities provide this missing feature. They implement a form of encapsulation that resembles the use of closures with encapsulated local state in high-level languages, and they allow implementing *compartments* which encapsulate private state and capabilities but can be called from untrusted code. From a security perspective, sentry capabilities allow setting up protection boundaries: the code executing before and after an invocation of a sentry capability has different authority and thus represent distrusting components. We denote sentry capabilities with permission E (for "Enter", a terminology originating from the M-machine [27]).

One typically creates a sentry capability pointing to a region of memory describing a compartment containing executable code and local state (or private capabilities to that local state). A sentry capability is opaque: it cannot be used to read or write to the memory region it points to, and it cannot be modified using `restrict` or `subseg`. It can thus be safely shared with untrusted third-parties: they will not be able to access the encapsulated code and data. In the figure below, the memory region pointed to by r_1 (hatched in gray) is not accessible for either reading or writing.



The only possible operation is to "invoke" the sentry capability using the `jmp` instruction, thus passing control to the code held in the region pointed to by the capability (in other words, "running" the compartment). When `jmp` is called on a sentry capability, it turns the capability into a capability with permission read-execute (RX) over the same memory region, and puts it into the program counter register pc. This simultaneously runs the encapsulated code, and gives access to the data and capabilities stored there, which were previously inaccessible. Running instruction `jmp r1` on the scenario of the previous figure leads to the machine state shown below.



Register pc now contains an RX capability to the previously opaque region, meaning that code contained in that region can execute. Furthermore, it may access other capabilities stored in that region, which can in turn be used to transitively access other private regions of memory.

Example: a counter compartment To illustrate the use of sentry capabilities, let us consider the example of a simple secure compartment implementing a counter. An instance of the counter holds a private memory cell containing the current (integer) value of the counter. Every time the code in the counter's compartment is invoked, it increases the value stored in the memory cell. Using a sentry capability, one can expose the counter to an untrusted context, without giving it direct access to the counter value.

It is worth pointing out that this is similar to the use of closures encapsulating local state in high-level languages. Typically, a similar counter program could be implemented in a high-level language as follows, using a function closure to encapsulate a reference holding the counter value.

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda().\, x := !x + 1; !x)$$

As before, our actual counter program is implemented in assembly, and its code appears in Figure 2.4. Its implementation is divided into two parts. First, the code starting at label `init` (and ending at `code`) is used to set up the counter compartment; it is intended to run only once at the beginning of the program. Then, the region between `code` and `end` corresponds to the contents of the counter compartment itself, including its executable code (between `code` and `data`) and private data (between `data` and `end`).

The role of the initialization code is to create a sentry capability encapsulating the `code`–`end` region, and then pass control to the (untrusted) context, giving it access to the newly created sentry capability. Additionally, the initialization code stores at address `data` a capability giving read-write access to the compartment's region, and pointing to the counter's value at address `data`+1.

One might wonder why we have this extra indirection to the counter's value through the capability in `data`. Recall that after calling `jmp` on a sentry capability, the program counter is only provisioned with an RX capability. For the counter code to be able to actually increment the counter value (at address `data`+1), it needs to have write access to it. The additional RWX capability stored at address `data` by the initialization code is thus used to "promote" read access on the compartment's region into read-write access to that same region.

The code of the counter's compartment can then run many times, once each time the context chooses to invoke the sentry capability it got from the initialization code. At each invocation, the counter's implementation (at address `code`) reads the RWX capability stored in the data section, uses it to increment the value of the counter, and passes control back to its caller.

Let us walk through the details of the code. The initialization code is assumed to run starting with a program counter giving RWX access over the whole program region. The first four instructions derive, from the program counter, RWX capabilities pointing to addresses `data` and `data`+1. Then, using the `store` instruction, the capability (RWX, `init`, `end`, `data`+1) is stored at address `data`. Next, after using `lea` and `subseg` to adjust the address and bounds of the capability, a sentry capability

```
; initially, PC = (RWX, init, end, init)
;            r0 = (unknown) pointer to the context
init:
  mov r1 PC              ; r1 = (RWX, init, end, init)
  lea r1 [data-init]     ; r1 = (RWX, init, end, data)
  mov r2 r1              ; r2 = (RWX, init, end, data)
  lea r2 1               ; r2 = (RWX, init, end, data+1)
  store r1 r2            ; mem[data] <- (RWX, init, end, data+1)
  lea r1 [code-data]     ; r1 = (RWX, init, end, code)
  subseg r1 [code] [end] ; r1 = (RWX, code, end, code)
  restrict r1 E          ; r1 = (E, code, end, code)
  mov r2 0               ; r2 = 0
  jmp r0                 ; jump to unknown code: we only give it
                         ; access to an enter capability pointing
                         ; to 'code'
; when 'code' gets executed from the E capability,
;   PC = (RX, code, end, code)
;   r0 = (unknown) return pointer to the continuation
code:
  mov r1 PC              ; r1 = (RX, code, end, code)
  lea r1 [data-code]     ; r1 = (RX, code, end, data)
  load r1 r1             ; r1 = (RWX, init, end, data+1)
  load r2 r1             ; r2 = <counter value>
  add r2 r2 1            ; r2 = <counter value> + 1
  store r1 r2            ; mem[data+1] <- <counter value> + 1
  mov r1 0               ; r1 = 0
  jmp r0                 ; return to unknown code
data:
  0xFFFF, ; will be overwritten with (RWX, init, end, data+1), i.e.
          ; a read-write capability to the counter value
  0       ; our private data: the current value of the counter
end:
```

Figure 2.4: Program implementing a secure counter

is created pointing to the compartment's region [code, end). This is done using the `restrict` instruction, turning a capability with permission RWX into a capability with permission E. Register r_2 is then cleared, to make sure that the RWX capability pointing to the counter value is not leaked to the context. Finally, the initialization code jumps to the pointer in r_0, which by convention points to the context.

The compartment's code (starting at address code) then gets executed each time the context invokes the sentry capability. Because we have only shared a sentry capability (E, code, end, code) with the context, we know that when the compartment gets executed, the program counter must contain (RX, code, end, code). By reading the program counter, the first two instructions of the code then derive an RX capability pointing to address data, and use it (with `load`) to read the capability that was stored there, granting RWX access to `data`+1. The subsequent `load`, `add` and `store` instructions use this second capability to increment the value of the counter. Finally, before returning to the context by jumping to r_0, the program takes care of clearing register r_1, overwriting its contents with 0. This is quite crucial, as otherwise an RWX capability would be leaked to the context, giving it direct access to the counter's private state!

To sum up, our example program carefully selects which capabilities it shares with unknown code, and leverages the encapsulation properties of sentry capabilities provided by the machine. Consequently, it should seem clear, at least informally, that the integrity of the counter's value is guaranteed through the execution. More precisely, we should be able to formally prove some invariant about it: for instance, that it is nonnegative at every step of the execution, for any untrusted context. In Section 2.6.2, we show in more detail how to formally establish this property.

In this section, we have showcased how one might program with capabilities in order to obtain security guarantees, and make it possible to interact with adversarial code while protecting private data and invariants.

In the rest of this paper, we show how we can make the intuitions that we have developed so far more precise, and formally prove capability safety for machine code programs that interact with untrusted code. Namely:

- We expect to have some concrete known code, which has some private data and invariants, and interacts with untrusted code.

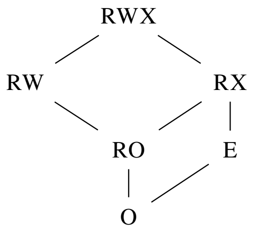
- We formally define the operational semantics of the capability machine that we consider (Section 2.3). This precisely defines the behavior of the machine on which the rest of our framework is built.

- Then we develop (Section 2.4) a program logic which supports formally verifying correctness properties about known code. Given some verified known code, we would then like to be able to conclude some result about a complete execution of the machine, when it runs a combination of the known code and some arbitrary untrusted code.

- To that end we need a way of formally capturing the fact that the machine effectively restricts the behavior of arbitrary code at runtime, by limiting the capabilities it has access to. We do this (Section 2.5) by defining a logical relation capturing "capability safety" of arbitrary code.

- By combining the Adequacy theorem of our program logic and the Fundamental theorem of our logical relation, we can prove safety of concrete examples (Section 2.6) and obtain theorems about complete executions of the machine.

$$\begin{array}{lcll}
a & \in \mathrm{Addr} & \triangleq & [0, \mathrm{AddrMax}] \\
p & \in \mathrm{Perm} & ::= & \mathrm{O} \mid \mathrm{E} \mid \mathrm{RO} \mid \mathrm{RX} \mid \mathrm{RW} \mid \mathrm{RWX} \\
c & \in \mathrm{Cap} & \triangleq & \{(p,b,e,a) \mid b,e,a \in \mathrm{Addr}\} \\
w & \in \mathrm{Word} & \triangleq & \mathbb{Z} + \mathrm{Cap} \\
reg & \in \mathrm{Reg} & \triangleq & \mathrm{RegName} \rightarrow \mathrm{Word} \\
m & \in \mathrm{Mem} & \triangleq & \mathrm{Addr} \rightarrow \mathrm{Word} \\
s & \in \mathrm{ExecState} & ::= & \mathsf{Running} \mid \mathsf{Halted} \mid \mathsf{Failed} \\
\varphi & \in \mathrm{ExecConf} & \triangleq & \mathrm{Reg} \times \mathrm{Mem}
\end{array}$$



Lattice defining the $\preccurlyeq$ relation.

(We have $p_1 \preccurlyeq p_2$ if there is a path going up from p_1 to p_2 in the diagram.)

$r \in \mathrm{RegName} ::= \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_{31}$ $\qquad \rho \in \mathbb{Z} + \mathrm{RegName}$

$$\begin{array}{rl}
i ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{mov}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
& \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
& \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}$$

Figure 2.5: Base definitions for the machine's words, state, and instructions.

2.3 Operational semantics of a capability machine

The very basis of our framework is a formal description of the capability machine we consider: which instructions it supports, and its behavior when it runs and executes programs. Technically speaking, this description corresponds to the operational semantics of the machine, upon which the program logic described next in Section 2.4 is built.

Our capability machine draws inspiration from CHERI [158], albeit in a simplified form, and only covers a subset of the features found in CHERI machines. Compared to a realistic CHERI machine, we consider a number of simplifications: our instruction set is minimal, our machine does not have virtual memory or different privilege levels, machine words can store unbounded integers, every instruction can be encoded in a single machine word, we do not consider memory alignment issues, and we abstract away from the binary encoding of capabilities. Nevertheless, our semantics does capture many of the aspects that make reasoning about machine code programs challenging: our machine has a finite amount of memory, a fixed number of registers, and there are no distinctions between code and data nor structured control flow for programs, owing to the fact that program instructions are simply encoded and stored in memory as normal integers.

Figure 2.5 gives the basic definitions that will play a role in the operational semantics of machine instructions. The set of memory addresses Addr is finite, and corresponds to the integer range $[0, \mathrm{AddrMax}]$. A memory word $w \in \mathrm{Word}$ is either an (unbounded) integer or a capability c. Capabilities are of the form (p,b,e,a), giving

$$\mathrm{updPC}(\varphi) = \begin{cases} (\mathsf{Running}, \varphi[\mathrm{reg}.\mathsf{pc} \mapsto (p,b,e,a+1)]) & \text{if } \varphi.\mathrm{reg}(\mathsf{pc}) = (p,b,e,a) \\ (\mathsf{Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\mathrm{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{if } \rho \in \mathrm{RegName} \end{cases}$$

$$\mathrm{updatePcPerm}(w) = \begin{cases} (\textsc{rx}, b, e, a) & \text{if } w = (\textsc{e}, b, e, a) \\ w & \text{otherwise} \end{cases}$$

Figure 2.6: Operational semantics: auxiliary definitions

access to the memory range $[b,e)$ with permission p, while currently pointing to a. The permissions p are ordered according to the lattice appearing at the top-right of the figure, inducing a bottom-to-top partial order $\preccurlyeq$ on permissions. There are six different permissions; the null (O), read-only (RO), enter (E), read-write (RW), read-execute (RX) and read-write-execute (RWX) permissions.

The state of the machine is modeled by the semantics as a pair of an execution state s and a configuration φ. An execution state flag indicates whether the machine is presently running (Running), has successfully halted (Halted), or has stopped execution by raising an error (Failed). A configuration φ contains the state of the registers $\varphi.reg$ and the memory $\varphi.mem$. A register file reg consists of a map from register names r to machine words, while the memory m maps addresses to words.

Next, Figure 2.5 shows the list of instructions of our machine. An instruction i typically operates on register names r, but can also sometimes take integer values as parameters; ρ denotes an instruction parameter which can be either a register name or a constant integer. Our machine features general purpose registers ($\mathsf{r}_0 - \mathsf{r}_{31}$), on top of the pc register, which points to the address in memory where the currently executing instruction is stored. (Technically speaking, pc must point to a memory cell containing an integer which can be successfully decoded into an instruction.) pc should therefore always contain a capability with at least permission RX; in any other case, the machine fails immediately.

Figure 2.7 defines the small-step operational semantics for the capability machine, using the auxiliary definitions from Figure 2.6. The rule EXECSINGLE describes how a single instruction gets executed. An execution step requires an executable and in-bounds capability in the pc register, and fails otherwise. It expects the memory cell pointed to by the capability to store an integer z, decodes it into an instruction and executes the instruction on the current state φ; the new configuration is denoted $[\![decode(z)]\!](\varphi)$. The table making up most of Figure 2.7 defines the operational behavior $[\![i]\!](\varphi)$ for each instruction i of the machine.

Most instructions use the auxiliary function updPC to increment the pc register after their proper operations. Because the address space is finite, pointer arithmetic such as incrementing pc can result in illegal addresses. To avoid notational clutter, we will always write as if arithmetic operations succeed, and consider that otherwise the

EXECSINGLE

$$(\mathsf{Running}, \varphi) \to \begin{cases} \llbracket decode(z) \rrbracket(\varphi) & \text{if } \varphi.\mathrm{reg}(\mathsf{pc}) = (p,b,e,a) \land b \leq a < e \land \\ & \quad p \in \{\mathrm{RX}, \mathrm{RWX}\} \land \varphi.\mathrm{mem}(a) = z \\ (\mathsf{Failed}, \varphi) & \text{otherwise} \end{cases}$$

i	$\llbracket i \rrbracket(\varphi)$	Conditions
`fail`	$(\mathsf{Failed}, \varphi)$	
`halt`	$(\mathsf{Halted}, \varphi)$	
`mov` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho)$
`load` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,b,e,a)$ and $w = \varphi.\mathrm{mem}(a)$ and $b \leq a < e$ and $p \in \{\mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}\}$
`store` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ and $b \leq a < e$ and $p \in \{\mathrm{RW}, \mathrm{RWX}\}$ and $w = \mathrm{getWord}(\varphi, \rho)$
`jmp` r	$(\mathsf{Running},$ $\varphi[\mathrm{reg}.\mathsf{pc} \mapsto newPc])$	$newPc = \mathrm{updatePcPerm}(\varphi.\mathrm{reg}(r))$

`jnz` $r_1\ r_2$	if $\varphi.\mathrm{reg}(r_2) \neq 0$, then
	$(\mathsf{Running},$
	$\varphi[\mathrm{reg}.\mathsf{pc} \mapsto newPc])$
	else $\mathrm{updPC}(\varphi)$
		$newPc = \mathrm{updatePcPerm}(\varphi.\mathrm{reg}(r_1))$

`restrict` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ and $p' = \mathrm{decodePerm}(\mathrm{getWord}(\varphi, \rho))$ and $p' \preccurlyeq p$ and $w = (p',b,e,a)$
`subseg` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ and for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \mathrm{E}$ and $w = (p, z_1, z_2, a)$
`lea` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$, $z = \mathrm{getWord}(\varphi, \rho)$ and $p \neq \mathrm{E}$ and $w = (p,b,e,a+z)$
`add` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 + z_2$
`sub` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 - z_2$
`lt` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and if $z_1 < z_2$ then $z = 1$ else $z = 0$
`getp` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto z])$	$\varphi.\mathrm{reg}(r_2) = (p, \_, \_, \_)$ and $z = \mathrm{encodePerm}(p)$
`getb` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto b])$	$\varphi.\mathrm{reg}(r_2) = (\_, b, \_, \_)$
`gete` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto e])$	$\varphi.\mathrm{reg}(r_2) = (\_, \_, e, \_)$
`geta` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto a])$	$\varphi.\mathrm{reg}(r_2) = (\_, \_, \_, a)$
`isptr` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto z])$	if $\varphi.\mathrm{reg}(r_2) = (\_, \_, \_, \_)$ then $z = 1$ else $z = 0$
_	$(\mathsf{Failed}, \varphi)$	otherwise

Figure 2.7: Operational semantics: execution of a single instruction.

machine transitions to a Failed state. The auxiliary function getWord is used to get the value corresponding to the argument ρ of an instruction: either its corresponding integer value if it is an immediate integer, or the contents of the corresponding register if it is a register name. The auxiliary function updatePcPerm is used in the definition of the behavior of the `jmp` and `jnz` instructions to unseal sentry capabilities. As mentioned previously, an additional effect of these jump instructions is to unseal sentry (E) capabilities into RX capabilities.

We now describe the semantics of the instructions of the machine, as formally defined in the table of Figure 2.7. The `fail` and `halt` instructions stop the execution of the machine, in the Failed and Halted state respectively. `mov` $r\ \rho$ copies ρ (either an immediate value or the contents of the corresponding register name) into register r. The instructions `load` and `store` allow reading and writing memory: `load` $r_1\ r_2$ reads the value pointed to by the capability in r_2 provided it has the permission R and points within its bounds; `store` $r\ \rho$ stores ρ to the location pointed to by the capability in r provided it has the W permission and points within bounds. The `jmp` and `jnz` instructions correspond to an unconditional and conditional jump respectively, thus loading the provided capability into pc. Using updatePcPerm, in the case of a sentry (E) capability, they unseal it into a RX capability first. Three instructions allow deriving new capabilities from existing ones. `restrict` $r\ \rho$ allows restricting the permission of a capability (where ρ provides an integer encoding of the desired permission), provided it is less permissive than the current permission according to $\preccurlyeq$. `subseg` $r\ \rho_1\ \rho_2$ restricts the range of authority of the capability stored in r, provided it is a subset of the current range of the capability. `lea` $r\ \rho$ modifies the current address of the capability in r, by adding to it the integer offset ρ. As should be expected, `subseg` and `lea` fail for sentry capabilities. Arithmetic operations are provided by the `add`, `sub` and `lt` instructions, which implement addition, subtraction, and comparison on integers, respectively. Finally, a number of instructions allow inspecting machine words and capabilities. `isptr` can be used to query whether a machine word is an integer or a capability, and `getp`, `getb`, `gete`, and `geta` return the different parts of a capability (permission, bounds and address). (More precisely, `getp` returns an integer encoding the permission, as given by encodePerm.) If any of the capability checks for an instruction are not satisfied, the machine fails.

An important aspect of our operational semantics is how it explicitly accounts for errors: when a capability check fails (for instance when a program tries to use a capability outside of its range), the semantics does not get stuck (meaning that it would not be able to reduce): instead, it explicitly transitions to a state with the Failed execution state flag.

2.4 Program logic

The operational semantics presented in the previous section formally define the behavior of our machine when it runs and executes code. Based on that, we expect to be able to formally verify concrete programs running on the machine.

The most direct approach would be to manually establish properties of sequences of reduction steps, based on the sole definition of the operational semantics. We do not follow this approach, because it would quickly become very tedious even for simple programs.

Instead, we draw from previous research in program logics and separation logic, and define Cerise: a program logic which provides a convenient framework in which to modularly reason about programs running on our machine. Indeed:

- It is typically more convenient to devise a system of proof rules for programs, rather than work directly at the level of abstraction provided by the bare operational semantics. Such rules form a program logic, which can be proved sound according to the operational semantics, and then can be used to verify properties of concrete programs.

- Separation logic, a family of program logics, has been widely used to reason about programs manipulating shared mutable state (such as memory). On our capability machine, not only do all programs access a mutable shared memory, but programs are themselves represented as unstructured data in memory; so the use of separation logic is particularly called for. Separation logic enables modular reasoning about programs that operate only on a sub-part of the global state, allowing them to be freely composed with programs that operate on a disjoint part of the state.

The first step is to consider what part of the machine state should be described by separation logic assertions. Here, the machine state consists of both the machine memory and the machine registers. Indeed, it is useful to modularly reason about programs operating on both a subset of memory and a subset of the available registers.

Technically speaking, we build the Cerise program logic on top of the Iris framework [77], which provides us with additional useful features, such as invariants. In the following we introduce both the basic separation logic assertions describing the machine state and additional features inherited from Iris (Section 2.4.1). Then, we describe the rules that are used to specify the execution of machine instructions and programs (Section 2.4.2).

Note that the program logic is, in a sense, only a technical device. The end goal is to obtain theorems that only refer to reductions in the operational semantics of our machine. To that end, we present (Section 2.4.3) an Adequacy theorem for our logic, which allows us to “extract” a correctness theorem expressed in terms of the operational semantics of the machine from a proof established in the program logic.

2.4.1 Basic resources

Figure 2.8 shows the syntax of our Cerise program logic based on Iris. We write *iProp* for the universe of propositions. These feature the standard connectives of higher-order logic and separation logic, including the separating conjunction $*$ and

$$
\begin{array}{ll}
\multicolumn{2}{l}{P, Q \in \mathit{iProp} ::=} \\
\quad \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \ldots & \text{higher-order logic} \\
\quad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \lceil \phi \rceil \mid \Box P \mid \triangleright P & \text{separation logic} \\
\quad \mid \mathsf{a} \mapsto w \mid r \Mapsto w \mid \vec{a} \mapsto \vec{l} & \text{machine resources} \\
\quad \mid \boxed{P} & \text{invariants} \\
\quad \mid \langle P \rangle \to \langle s.\, Q \rangle \mid \{P\} \rightsquigarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \bullet & \text{program logic}
\end{array}
$$

Figure 2.8: The syntax of our program logic.

the magic wand $\mathrel{-\!\!*}$ (read as an implication). The proposition $\lceil \phi \rceil$ asserts that the pure proposition ϕ holds, where ϕ is a proposition from the meta logic.

Iris assertions can be divided in two categories: *ephemeral* assertions and *persistent* assertions. Ephemeral assertions describe facts or resources that are available at a given point but might become false or unavailable later. Persistent assertions describe facts that never cease to be true. The assertion $\Box P$, read "persistently P", is persistent, and asserts ownership over resources whose duplicable part satisfies P. In other words, $\Box P$ is like P except that it does not assert any exclusive ownership over resources. As the knowledge associated with a persistent assertion can never be invalidated, persistent assertions can be freely duplicated.

The modality $\triangleright P$ expresses (roughly) that the assertion P holds after one "logical step" of execution. In this paper, we mainly use it to define recursive predicates using guarded recursion. It is not necessary to understand how the modality behaves in detail and the reader can safely ignore it for the most part and just recall that it supports an abstract accounting of execution steps.

Our logic includes resources (predicates) that describe parts of the current state of the machine. The assertion $\mathsf{a} \mapsto w$ expresses that the memory cell at address a contains the machine word w. Furthermore, this assertion should be read as giving *unique ownership* over location a, giving the right to freely read and update the corresponding memory cell. Similarly, the assertion $r \Mapsto w$ asserts ownership of a CPU register r containing the word w. We write $\vec{a} \mapsto \vec{l}$ for the ownership of contiguous memory cells at addresses $\vec{a}$ containing $\vec{l}$.

A key feature of the logic is the notion of an invariant. The assertion $\boxed{P}$ asserts that P should hold at all times, now and for every future step of the execution (where P can be any separation logic assertion). An invariant is a persistent assertion. An invariant $\boxed{P}$ can be created (or "allocated") by handing over the resources for P, turning them into $\boxed{P}$. Then, whenever we know that $\boxed{P}$ holds, we can get access to the resources P held in the invariant, but only for the duration of one program step. Indeed, since the invariant must hold at every step of the execution, when accessing its resources, one needs to show that it holds again no later than one program step after. A more precise rule for accessing invariants is given next in Section 2.4.2 (rule INV).

2.4.2 Program specifications

The predicates for machine resources we just presented allow describing the state of the machine. Our logic, moreover, includes assertions that can be used to specify machine executions, similar to *Hoare triples* used in program logics for high-level languages. Because we work with a low-level machine (where code is located in memory), we distinguish between three different types of program specifications:

$\langle P \rangle \rightarrow \langle s.\, Q \rangle$	single instruction
$\{P\} \rightsquigarrow \{s.\, Q\}$	code fragment
$\{P\} \rightsquigarrow \bullet$	complete safe execution.

In each case, P are Q are separation logic assertions describing the state of the machine (registers and memory). P corresponds to a pre-condition, Q a post-condition, and s binds in Q the corresponding execution state (of type ExecState, see Figure 2.5).

Informally, $\langle P \rangle \rightarrow \langle s.\, Q \rangle$ holds if, starting from a machine state satisfying P, the machine can execute one step of computation, and reach a state satisfying Q in an execution state s. The predicate $\{P\} \rightsquigarrow \{s.\, Q\}$ holds if, starting from a state satisfying P, then the machine can diverge (i.e. loop) or reach a state satisfying Q in an execution state s. This is typically used to describe the execution of a code fragment. Finally, $\{P\} \rightsquigarrow \bullet$ holds if, starting from a machine state satisfying P, then the machine loops forever or runs until completion, ending in either a Halted or Failed state. In this case, we say that the initial configuration described by P is *safe*. (Not every configuration is safe: the resources in P describing registers and memory must suffice for the machine to run and execute the code pointed to by pc: we do not have $\{pc \Mapsto w\} \rightsquigarrow \bullet$ in general.)

Additionally, these three specifications *require the logical invariants to be preserved at every step of the execution*. This requirement is implicit in the definition of invariants, but it is a crucial reasoning principle that we will leverage.

Echoing back to Section 2.2.2, note that our program specification for a complete safe execution allows the program to fail (or diverge). Indeed, we will capture the preservation of security properties by preserving *invariants* throughout execution and having the machine fail is both fine (invariants are trivially preserved when the machine ends up in a failure state) and unavoidable (we cannot prevent unknown code from triggering a capability check failure). Similar considerations apply for divergence.

Notations In the rest of the paper, we will rely on a couple of additional notations when writing program specifications. Because we often want to reason about the case where an instruction (or program fragment) does not fail, we write $\langle P \rangle \rightarrow \langle Q \rangle$ (respectively $\{P\} \rightsquigarrow \{Q\}$) to denote a resulting execution state equal to Running:

$$
\begin{aligned}
\langle P \rangle \rightarrow \langle Q \rangle \quad &\triangleq \quad \langle P \rangle \rightarrow \langle s.\, \lceil s = \mathsf{Running} \rceil * Q \rangle \\
\{P\} \rightsquigarrow \{Q\} \quad &\triangleq \quad \{P\} \rightsquigarrow \{s.\, \lceil s = \mathsf{Running} \rceil * Q\}\,.
\end{aligned}
$$

When writing pre- and post-conditions, we will often need to include a points-to resource describing the contents of the pc register. We introduce a short-hand notation for that purpose, and write $w; P$ to assert P and additionally that pc is set to w:

$$w; P \quad \triangleq \quad \mathsf{pc} \Mapsto w * P$$

Using these two notations, the specification for a single instruction, in a case where it does not fail, is written as $\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle$ (typically, we have $w_1 = w_0 + 1$, except in the case of the `jmp` and `jnz` instructions, or when explicitly writing to the pc register).

Properties Our program specifications satisfy the well-known “frame rule” of separation logic, which permits local reasoning, and asserts that it is always possible to extend a specification by adding arbitrary resources not accessed by the program.

$$\overset{\textsc{FragFrame}}{\frac{\{P\} \rightsquigarrow \{s.\,Q\}}{\{P * R\} \rightsquigarrow \{s.\,Q * R\}}} \qquad \overset{\textsc{StepFrame}}{\frac{\langle P\rangle \rightarrow \langle s.\,Q\rangle}{\langle P * R\rangle \rightarrow \langle s.\,Q * R\rangle}} \qquad \overset{\textsc{FullFrame}}{\frac{\{P\} \rightsquigarrow \bullet}{\{P * R\} \rightsquigarrow \bullet}}$$

Program specifications can also be composed using sequencing rules. In order to establish a specification of the form $\{P\} \rightsquigarrow \{s.\,Q\}$, one typically uses single-instructions rules ($\langle R\rangle \rightarrow \langle s.\,S\rangle$) in a sequence, one for each instruction of the relevant code block. Specifications for two program fragments that follow each other can also be combined to obtain a specification for the sequence of the two fragments. We prove general sequencing rules for our three kind of specifications; for simplicity, we only reproduce here restricted rules that deal with successful executions (relying on the notations introduced before):

$$\overset{\textsc{SeqFrag}}{\frac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}}} \qquad \overset{\textsc{SeqFull}}{\frac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet}}$$

$$\overset{\textsc{StepFull}}{\frac{\langle P\rangle \rightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet}} \qquad \overset{\textsc{StepFrag}}{\frac{\langle P\rangle \rightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}}}$$

When reasoning about a single execution step, one can additionally access resources held in known invariants. This is done using the Inv rule, given below: [2]

$$\overset{\textsc{Inv}}{\frac{\langle P * \triangleright R\rangle \rightarrow \langle s.\,Q * \triangleright R\rangle}{\boxed{R} \vdash \langle P\rangle \rightarrow \langle s.\,Q\rangle}}$$

[2] For clarity of the presentation, we choose to omit additional details related to Iris invariant namespaces and masks. We refer to the Coq development for the full details ②.

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc}) \\ \mathrm{ValidSubseg}(p,b,e,z_1,z_2) \qquad \mathrm{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2\end{array}}{\begin{array}{l}\langle (p_{pc},b_{pc},e_{pc},a_{pc})\ ; \quad a_{pc} \mapsto n * r \Mapsto (p,b,e,a)\rangle \to \\ \langle (p_{pc},b_{pc},e_{pc},a_{pc}+1)\ ; \ a_{pc} \mapsto n * r \Mapsto (p,z_1,z_2,a)\rangle\end{array}}$$

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc}) \\ \neg\mathrm{ValidSubseg}(p,b,e,z_1,z_2) \qquad \mathrm{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2\end{array}}{\begin{array}{r}\langle (p_{pc},b_{pc},e_{pc},a_{pc})\ ; \ a_{pc} \mapsto n * r \Mapsto (p,b,e,a)\rangle \to \\ \langle s.\ \lceil s = \mathsf{Failed}\rceil * ((p_{pc},b_{pc},e_{pc},a_{pc})\ ; \ a_{pc} \mapsto n * r \Mapsto (p,b,e,a))\rangle\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc}) \qquad \mathrm{ValidLoad}(p,b,e,a) \qquad \mathrm{decode}(n) = \texttt{load}\ dst\ src}{\begin{array}{l}\langle (p_{pc},b_{pc},e_{pc},a_{pc})\ ; \quad a_{pc} \mapsto n * dst \Mapsto - * src \Mapsto (p,b,e,a) * a \mapsto w\rangle \to \\ \langle (p_{pc},b_{pc},e_{pc},a_{pc}+1)\ ; \ a_{pc} \mapsto n * dst \Mapsto w * src \Mapsto (p,b,e,a) * a \mapsto w\rangle\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc}) \qquad \mathrm{ValidStore}(p,b,e,a) \qquad \mathrm{decode}(n) = \texttt{store}\ dst\ src}{\begin{array}{l}\langle (p_{pc},b_{pc},e_{pc},a_{pc})\ ; \quad a_{pc} \mapsto n * dst \Mapsto (p,b,e,a) * src \Mapsto w * a \mapsto -\rangle \to \\ \langle (p_{pc},b_{pc},e_{pc},a_{pc}+1)\ ; \ a_{pc} \mapsto n * dst \Mapsto (p,b,e,a) * src \Mapsto w * a \mapsto w\rangle\end{array}}$$

$$\frac{\forall i \in [0,n),\ \mathrm{ValidPC}(p,b,e,a_i) \qquad n = \mathrm{length}(\mathrm{rclear\_instrs}\ l)}{\begin{array}{l}\big\{(p,b,e,a_0); \mathop{\Large\ast}_{r\in l} r \Mapsto - * \mathop{\Large\ast}_{i\in[0,n)} a_i \mapsto (\mathrm{rclear\_instrs}\ l)[i]\big\} \rightsquigarrow \\ \big\{(p,b,e,a_n); \mathop{\Large\ast}_{r\in l} r \Mapsto 0\ * \mathop{\Large\ast}_{i\in[0,n)} a_i \mapsto (\mathrm{rclear\_instrs}\ l)[i]\big\}\end{array}}$$

$$\begin{array}{lcl}
\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc}) & \triangleq & \mathrm{RX} \preccurlyeq p_{pc} \land b_{pc} \le a_{pc} < e_{pc} \\
\mathrm{ValidSubseg}(p,b,e,z_1,z_2) & \triangleq & p \neq \mathrm{E} \land b \le z_1 \land 0 \le z_2 \le e \\
\mathrm{ValidLoad}(p,b,e,a) & \triangleq & \mathrm{RO} \preccurlyeq p \land b \le a < e \\
\mathrm{ValidStore}(p,b,e,a) & \triangleq & \mathrm{RW} \preccurlyeq p \land b \le a < e \\
\mathrm{rclear\_instrs}\ l & \triangleq & \mathrm{map}\ (\lambda r.\ \mathrm{encode}(\texttt{move}\ r\ 0))\ l
\end{array}$$

Figure 2.9: Specifications for the machine instructions `subseg`, `load` and `store` and for the `rclear` macro that sets a given list of registers to zero. Changes to the machine state are highlighted in red.

Example specifications As illustrative examples, Figure 2.9 shows specifications for the `subseg`, `load` and `store` instructions, as well as the `rclear` macro which is used to clear the contents of a number of specified registers. The first rule shows a specification for the `subseg` instruction. It states that if the program counter contains a capability pointing to a memory location a_{pc}, if that location contains an integer n which decodes into `subseg` $r\ z_1\ z_2$, and if the register r contains a capability, then assuming that the program counter is valid (ValidPC(...)) and that z_1 and z_2 are valid new bounds (ValidSubseg(...)), the machine successfully increments the program counter and restricts the capability held in register r with new bounds z_1 and z_2.

The second rule is also a specification for `subseg`, but in a case where it fails a bound check, i.e. $\mathrm{ValidSubseg}(p, b, e, z_1, z_2)$ does not hold. (For instance, when the new bounds z_1 and z_2 would allow accessing more memory than what is available through the original capability.) Then, the rule does give us a specification for an execution step, but with a resulting execution state of Failed, meaning that the execution cannot continue afterwards.

The third and fourth rules give specifications for the `load` and `store` instructions (in non-failing cases). The specification for `load` states that `load` *dst src* loads a word from memory pointed to by a capability in register *src* and stores its contents in register *dst*. The specification for `store` states that `store` *dst src* reads a word from the *src* register and writes it into the memory location pointed to by the capability in *dst*.

Note that these specifications for `subseg`, `load` and `store` are not in fact the most general specifications for these instructions. They assume that some side-conditions hold, and specify the behavior of the instruction in the case of either a "normal" successful execution, or a failing one. These specifications are typically useful for reasoning about the correctness of a concrete program. We have also proved in Coq (e.g., ③ for the `subseg` instruction) "most general" specifications, covering in one lemma all possible cases for a given instructions. These are useful for deriving the more specific rules shown previously. Furthermore, we use them directly in the proof of the Fundamental Theorem (Theorem 2), for specifying the behavior of arbitrary instructions that might or might not fail.

The last rule of Figure 2.9 shows a derivable specification for a program composed of several instructions, the `rclear` macro. This macro (meaning, a small program that is typically inserted inline as part of a larger program) clears a number of registers by setting their content to 0. It is parameterized by a list l of register names, and its code consists of a sequence of instructions `move` r 0 for each register name r in l. We state `rclear`'s specification using the program specification for code fragments. This specification is provable using the basic reasoning rules for `move`. It requires that the body of the macro ("rclear_instrs l") is laid out contiguously in memory range $[a_0, a_n)$, while the program counter initially points to a_0. When the program counter eventually points to a_n, the address immediately after the macro's instructions, then all the registers in l have been cleared and now contain 0. (The "big star" $\ast$ denotes an iterated separating conjunction, here over the registers r in list l.)

2.4.3 Adequacy theorem

After establishing program specifications and properties at the level of our program logic, we ultimately want to transfer these results into properties of a program execution at the level of the operational semantics of the bare machine. Generally speaking, we prove using the rules of the Iris logic a statement of the form $\boxed{P} \vdash Q$, where P and Q are Iris propositions (read "Q holds assuming invariant P"). From this, we want to deduce that some mathematical proposition Φ holds (as a Coq proposition, in our case), where Φ describes some property of the machine execution expressed in terms of its operational semantics.

Because we are interested in establishing *invariants* about a program execution, we typically want to obtain in Φ that at every step of the execution, the state of the machine satisfies an invariant corresponding to the Iris assertion P.

Deriving mathematical facts from Iris proof derivations is made possible thanks to the so-called *adequacy* theorem of Iris ④. This theorem has a very general but intricate statement. In this section, we describe a simpler but more specialized adequacy theorem for our capability machine, which we can use to reason about the examples introduced in Section 2.2. (We also describe in Section 2.7 a more advanced adequacy theorem, suitable for reasoning about programs such as the case study of Section 2.8.) This specialized adequacy theorem is itself established on top of the general Iris adequacy theorem. When it applies, it is more convenient to use; but in the general case, it is always possible to directly leverage the general adequacy theorem.

We now present our specialized adequacy theorem. We first define a notion of *memory invariant* (Definition 1), which corresponds to a predicate over a finite subset of the machine memory. Typically, we will consider predicates of the form: "the value at this specific memory address holds a positive integer" (for instance, the value of the counter of Section 2.2.4). A memory invariant is given by a predicate I over machine memory and a set of addresses D (the "domain" of the invariant); we then require that I is not impacted by changes outside of D.

Definition 1 (Memory invariant ⑤)**.** *We say that I is a memory invariant over D if I is a predicate over machine memory, D a finite set of addresses, and:*

$$\forall m\, m'.\; (\forall a \in D.\; m(a) = m'(a)) \Longrightarrow I(m) \Leftrightarrow I(m').$$

We now present the statement of our specialized adequacy theorem; we explain the ingredients in the theorem statement below. Given a memory invariant I over a set D, our adequacy theorem (Theorem 1) can be used to show that I indeed holds of the memory at every step of the execution, provided we can show that it holds as an invariant in Iris.

Theorem 1 (Adequacy ⑥)**.** *Given a memory invariant I over D, a memory fragment $prog : [b, e) \to \mathrm{Word}$, a memory fragment $adv : [b_{adv}, e_{adv}) \to \mathrm{Word}$, an initial memory mem, and an initial register file reg, assuming that:*

1. *the initial state of memory mem satisfies:*

$$prog \uplus adv \subseteq mem \qquad D \subseteq \mathrm{dom}(prog) = [b, e)$$

2. *invariant I holds of the initial memory:*

$$I(mem)$$

3. *the adversary region contains no capabilities:*

$$\forall a \in \mathrm{dom}(adv).\ adv(a) \in \mathbb{Z}$$

4. *the initial state of registers reg satisfies:*

$$reg(\mathsf{pc}) = (\text{RWX}, b, e, b), \quad reg(\mathsf{r_0}) = (\text{RWX}, b_{adv}, e_{adv}, b_{adv}), \\ reg(r) \in \mathbb{Z} \text{ otherwise}$$

5. *the proof in Iris that the initial configuration is safe given invariant I:*

$$\forall reg,$$
$$\boxed{\exists m, \mathop{\ast}_{(a,w)\in m} a \mapsto w \ast \lceil \mathrm{dom}(m) = D \rceil \ast \lceil I(m) \rceil}$$
$$\vdash \left\{ (\text{RWX}, b, e, b); \begin{array}{l} \qquad \mathsf{r_0} \Mapsto (\text{RWX}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathop{\ast}_{\substack{(r,v)\in reg, \\ r \notin \{\mathsf{pc}, \mathsf{r_0}\}}} r \Mapsto z \ast \lceil z \in \mathbb{Z} \rceil \ast \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z \ast \lceil z \in \mathbb{Z} \rceil \ast \\ \mathop{\ast}_{\substack{(a,w)\in prog, \\ a \notin D}} a \mapsto w \end{array} \right\} \leadsto \bullet$$

Then, for any reg′, mem′, if $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $I(mem')$.

Theorem 1 establishes that, starting from an initial machine state (reg, mem), any subsequent machine state (reg', mem') satisfies $I(mem')$. This is subject to a number of conditions, in particular about the initial state of the machine.

First, the initial memory must be provisioned with relevant code and data. This means that the program that we wish to verify (both its code and data) given by memory fragment $prog : [b, e) \to \text{Word}$ should be included in the initial memory. Moreover, some additional "adversarial code" given by $adv : [b_{adv}, e_{adv}) \to \text{Word}$ should be included in the initial memory. Indeed, we are not only interested in reasoning about the execution of our verified program in isolation, but also its interaction with unverified, possibly adversarial code. The initial memory *mem* should therefore include *prog* and *adv*, in disjoint regions. Furthermore, the domain of the invariant I should be included in the program's region $[b, e)$. The intent is that I specifies an invariant about some private data of the verified program, and thus should not depend on other parts of memory.

Second, as should be expected, the invariant I must hold of the initial memory *mem*.

Third, the adversary memory *adv* is required not to contain any capabilities. This conservatively ensures that *adv* does not contain any “rogue” capability that would give undesired access to the verified program’s private state. No further assumption is made about *adv*, which is thus free to contain arbitrary code (i.e. instructions encoded as integers). Furthermore, note that the absence of capabilities in *adv* does not mean that code in *adv* will not be able to access memory at all: at runtime, it will still get access to a capability to its own region through the program counter pc.

Then, the initial register file *reg* should be provided with a RWX capability to the verified program in pc (meaning that it executes first), and a capability to the unverified code in register r_0 (as we have seen in Section 2.2, by convention r_0 holds the pointer to a program’s continuation). Other registers are conservatively required not to contain any capabilities.

Finally, one needs to establish at the level of the program logic that the program is safe to run under invariant I. Concretely, one needs to prove a specification for a complete safe execution (of the form $\{P\} \rightsquigarrow \bullet$), given “points-to” resources in the pre-condition that correspond to the initial state of registers and memory. In particular, we get access to points-to resources for the adversary region (along the fact that they contain integers) and points-to resources for the region containing the program to execute.

Note that no resources are given for the domain of I as part of the initial resources for the complete-execution specification. Instead, these resources are part of the logical invariant under which the specification must be established (inside $\boxed{\ldots}$). This corresponds to the intuition that these resources should only be modified in a way that preserves invariant I. This logical invariant therefore specifies that there exists a subset of memory m, which covers the memory region defined by D, such that the invariant holds the corresponding points-to resources and such that $I(m)$ holds, i.e. the memory invariant I holds of this memory subset. (Recall from Section 2.4.1 that $\lceil \phi \rceil$ denotes an Iris proposition that asserts that the mathematical proposition ϕ holds.)

The reader may be surprised to notice that the region containing “adversarial” code has no special status. Indeed, it simply corresponds to a memory region containing (a priori unknown) integers. Nevertheless, remember that we ultimately want our program to be able to pass control to the unknown adversary code by jumping to the capability in r_0, as we have seen our example programs do. This means we need to have a way of reasoning about “what it will do”, at least to ensure that it will not break our program’s invariants.

In the next section, we show how to reason about whether unknown code can be considered “safe to execute”, so that we can pass control to it while preserving previously established invariants.

$$
\begin{array}{ll}
\mathcal{V}(w) & \begin{cases}
\mathcal{V}(z), \mathcal{V}(\mathrm{O}, -, -, -) & \triangleq \mathsf{True} \\
\mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \triangleright \square \mathcal{E}(\mathrm{RX}, b, e, a) \\
\mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \mathop{\Large\ast}_{a \in [b,e)} \boxed{\exists w,\, a \mapsto w * \mathcal{V}(w)} \\
\mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \mathop{\Large\ast}_{a \in [b,e)} \exists P, \boxed{\exists w,\, a \mapsto w * P(w)} \\
 & \qquad * \triangleright \square \left(\forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)\right)
\end{cases} \\
\mathcal{E}(w) & \triangleq \forall reg,\ \left\{ w; \mathop{\Large\ast}_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet
\end{array}
$$

Figure 2.10: Logical relation defining “safe to share” and “safe to execute” values.

2.5 Reasoning about Untrusted Code in Cerise

Code running on a capability machine is constrained by the set of capabilities it has access to. This is a crucial idea for reasoning about adversarial code. Whatever code the machine is running, if this code does not have access to a capability for, e.g., writing to a memory region, then it will not be able to modify memory in that region. In other words, one can prove a theorem describing the behavior of arbitrary code depending only on the capabilities it has access to.

One major technical contribution of this work is to formulate and mechanize such a theorem. Specifically, we are concerned with the preservation of invariants established in the program logic. We will thus give a definition of which machine words that are “safe” to share with unknown code. Informally, a word is safe if it cannot be used to break any previously established logical invariants. We will then prove that, as long as some arbitrary code only has access to safe machine words, its execution indeed preserves logical invariants.

Interestingly, we can establish this result while staying within the framework of the Cerise program logic exposed in the previous section. This illustrates the generality of said program logic: verifying specifications for known programs or specifying the behavior of arbitrary code are only two of its possible applications.

2.5.1 Logical Relation

Our formal definition of what makes a machine word *safe*, meaning “safe to share with unknown code”, appears in Figure 2.10. It takes the form of a unary logical relation, defining simultaneously the notions of a machine word that is “safe to share” ($\mathcal{V}$) and “safe to execute” ($\mathcal{E}$). The names $\mathcal{V}$ and $\mathcal{E}$ originate from the tradition of logical relations, corresponding respectively to the “value relation” and the “expression relation”, although this interpretation is perhaps less obvious in the setting of low-level machine code. We explain the definition in detail below. The intuition is that:

- *A value which is safe to share* only gives transitive access to other values that are safe to share, or code that is safe to execute (in the case of a sentry capability).

- *A value which is safe to execute* allows the machine to run while preserving logical invariants (by definition of $\{\cdot;\cdot\} \rightsquigarrow \bullet$), provided the registers contain safe values.

Technically speaking, this informal definition is circular. Luckily, we can define it properly with the help of the "later" modality $\triangleright$. Iris provides us with a fixed-point operator that only requires recursive occurences to be guarded under a $\triangleright$, and we use that to formally define $\mathcal{V}$ and $\mathcal{E}$. Except for this technical requirement, the reader can in practice ignore the use of $\triangleright$ here.

Let us more closely examine the definition of $\mathcal{V}$, which is defined by case analysis on the shape of the given machine word w. If w is an integer (z), then it is always safe to share, since it cannot be used to access memory. Similarly, opaque capabilities with permission O are always safe as they also do not give access to memory.

A sentry capability E is safe to share if the code it encapsulates is safe to execute. Such a capability can be invoked at any moment and possibly several times: this is expressed through the use of the persistently modality $\square$. Technically speaking, this means that the property $\mathcal{E}(\textsc{rx}, b, e, a)$ must be established by only relying on persistent resources (typically, logical invariants) that will remain "available" throughout the entire execution.

A read-write capability RW or RWX gives read and write access to the memory region in its range. It is therefore safe as long as the words stored in the corresponding memory region are safe, and continue to be so when the memory gets modified. We thus say that it is safe when we have an invariant for each memory cell in the capability's region, which asserts ownership over the corresponding memory points-to resource, and asserts validity of its contents.

Finally, a capability with permission RO/RX cannot be used by unknown code to modify the memory words in its range. Therefore, these words can obey any property P as long as it entails safety ($\mathcal{V}$). Intuitively, the words in the interval have to be safe to share, because the adversary can read them. But since the adversary cannot modify them, it is possible to guarantee a stronger invariant about them. For instance, $P(w)$ could be the predicate "$w = 42$", describing that a value in the range stays equal to the integer 42.

Notice that this definition of safety does not distinguish between capabilities with permission RO and RX, or RW and RWX. This seems to strangely imply that permissions with the execute bit X have no additional expressive power over permissions without the execute bit. And indeed, *in terms of our model*—which "only" captures the ability to break memory invariants—their expressive power is the same![3] The crux of our main theorem (presented in the next sub-section) is that executing arbitrary code does not produce capabilities with more access to memory than was available before. Thus, being able to execute code within a memory region does not yield additional access to memory compared to what was available by simply reading the memory region (it only leads to additional machine behaviors).

Is this definition of safety trivial? One might wonder whether the definition in Figure 2.10 is trivial, meaning that any machine word w will in fact be considered

[3] Having read-only permission over a region also allows one to simply copy the contents of the region into any other read-execute region and execute them here.

safe. This is thankfully not the case; let us illustrate concrete cases where a memory word w is *not* considered safe to share with unknown code.

At a high level, $\mathcal{E}$ is not trivial because establishing $\mathcal{E}(w)$ requires proving that a full execution of the machine, starting from w, *preserves logical invariants*. This requirement is not explicit in the definition, but comes from the definition of the Cerise program logic. The definition of $\mathcal{V}(w)$ is also not trivial because, e.g., in the case of an RW capability, it requires the memory points-to $a \mapsto -$ predicate to be part of a specific invariant, $\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$. Since the resource "$a \mapsto -$" is not duplicable, there can be only one resource $a \mapsto -$, which cannot be simultaneously part of two different invariants. Memory cells whose contents evolve according to an invariant more specific (less permissive) than the one above thus cannot be associated with a safe capability (according to $\mathcal{V}$).

What is a concrete example of a capability which is *not* safe? Let us consider a memory cell at address x initialized to 0. Let us assume the following Iris invariant: $\boxed{x \mapsto 0}$. This invariant expresses that x will contain the integer 0 for the rest of the execution. Then, a capability $(\mathrm{RW}, x, x+1, x)$ is not safe to share with an adversary. Indeed, an adversary could use such a capability to write an arbitrary value at address x, thus invalidating the Iris invariant. (However, $(\mathrm{RO}, x, x+1, x)$ would be safe.) A bit more formally speaking, it is not possible to prove $\mathcal{V}(\mathrm{RW}, x, x+1, x)$, because it is not possible to create the invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$, as the resource for the memory cell x is already part of the invariant $\boxed{x \mapsto 0}$, and cannot be extracted to create a different invariant.

Similarly, one cannot prove $\mathcal{E}$ for a code fragment that writes another value than 0 at address x (after getting access to it through the pc register), because the proof would not be able to guarantee that the Iris invariant related to x is preserved at every step.

2.5.2 Fundamental Theorem

The Fundamental Theorem of our Logical Relation (Theorem 2) (hereafter, FTLR) establishes that any capability that is "safe to share" (in $\mathcal{V}$) is in fact "safe to execute" (in $\mathcal{E}$). In other words, if a capability only gives transitive access to safe capabilities, then it is safe to use it as a program counter capability and execute it: it will not be able to gain extra authority over memory or break any invariants. Importantly, this theorem is independent of the code that the capability points to, even though it is this code that will be executed. Hence the result applies to arbitrary code and we sometimes refer to it as a *universal contract* because of this.

Theorem 2 (FTLR ⑦)**.** *Let* $p \in \mathrm{Perm}, b, e, a \in \mathrm{Addr}$. *If* $\mathcal{V}(p, b, e, a)$*, then* $\mathcal{E}(p, b, e, a)$.

This is a non-trivial theorem, the proof of which requires checking all the possible cases of the semantics of each instruction of the machine. Indeed, one needs to check that there is no way for some machine instruction to create capabilities with further authority than what was available. This could, for example, happen if some runtime checks were missing, making it possible to create a capability $(\mathrm{RW}, b, e+1, a)$ from a capability (RW, b, e, a). One can imagine how this would break expected security guarantees, and reveal a design or implementation bug of the machine. Therefore, another informal interpretation of the fundamental theorem is that it expresses that the capability machine "works well" or that it is *capability safe*.

The fundamental theorem provides a universal security property satisfied by unknown code, and gives us a way of verifying the correctness of known code that includes calls to possibly malicious code. To sum up, our logical relation characterizes the interface between a piece of verified code which wishes to preserve invariants on some internal state, and "external" arbitrary code whose accessible, safe capabilities have been sufficiently restricted.

It is important to note that the distinction between "known" and "adversary" code only exists at the logical level: there is no such distinction at runtime. We can have two components that have been verified separately, and that do not mutually trust each other. In this case, from the point of view of each component, the other component is considered as being the adversary.

Rules for program verification. From the general statement of the FTLR, we can derive two corollaries, which can be used to instantiate our adequacy theorem (Theorem 1) with a program that passes control to an unknown adversarial code region.

Corollary 1 (Unknown integers are safe ⑧). *For* $m : [b, e) \to \text{Word}$,

$$\mathop{\ast}_{(a,z)\in m} a \mapsto z \ast \lceil z \in \mathbb{Z} \rceil \;\mathrel{-\!\!\ast}\; \mathcal{V}(p, b, e, a)$$

Corollary 1 can be used to trade ownership over a memory region of integers to the knowledge that a capability over this region is safe.[4] Since integers can encode program instructions, we can typically use this rule to reason about a memory region containing an (unknown) program. The rule follows directly from the definition of $\mathcal{V}$ for values of p different from E; when $p = \text{E}$, an additional application of the FTLR (Theorem 2) is required.

Notice that the pre-condition of the rule matches the resources that one gets in the Adequacy theorem (Theorem 1) for the adversary region. When using the Adequacy theorem, we will thus be able to conclude that capabilities pointing to the adversary region are safe.

Corollary 2 (Jump to a safe word ⑨).

$$\begin{array}{l} \mathcal{V}(w) \mathrel{-\!\!\ast} \\ \triangleright \forall reg.\, \left\{\text{updatePcPerm}(w); \ast_{(r,v)\in reg, r\neq \mathsf{pc}}\, r \Mapsto v \ast \mathcal{V}(v)\right\} \rightsquigarrow \bullet \end{array}$$

Corollary 2 gives us a specification for the execution of the machine after a jump to an unknown word w, assuming that w is safe. Recall that updatePcPerm(w)

[4] We simplify the presentation here a bit and omit a view shift from the statement of Corollary 1. See the Coq development for the exact formal statement ⑧.

corresponds to the value of the program counter after jumping to w (see the machine semantics in Figure 2.7). The full execution specification in the conclusion of the rule requires that the machine registers contain safe values: indeed, we must only share safe words with unknown code.

An important application of Corollary 2 is to reason about the last instruction of a program encapsulated in a sentry (E) capability, where it "returns" and passes control to its caller by calling `jmp` on the (unknown but safe) return pointer held in r_0. In this scenario, the return pointer provided by the caller is unknown but safe, so Corollary 2 gives us a specification for the continuation of the program.

Additionally, Corollary 2 is typically used in combination with Corollary 1 when instantiating the Adequacy theorem. Indeed, in order to prove the complete safe execution specification required by the theorem, one typically needs to justify that one can `jmp` and pass control to an adversary region, given the resources granted by the Adequacy theorem.

2.5.3 Proving the fundamental theorem

To give a more in-depth perspective of the ideas behind the Fundamental Theorem, we detail in this sub-section one of the interesting cases of its proof. This sub-section can be safely skipped on a first read.

Proof. (FTLR) We begin by unfolding the definition of $\mathcal{E}$.

$$\forall reg.\, \left\{(p,b,e,a); \mathop{\ast}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Rrightarrow v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

We proceed by Löb induction. The Löb rule is a powerful reasoning principle, which Cerise inherits from Iris, and which states that (in any context Q), if from $\triangleright P$ we can derive P, then we can also derive P without any assumptions.

$$\textsc{Löb}\quad \frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

The idea of the rule is that "after we do some work", we will be able to remove the $\triangleright$ modality from the assumption, and reach the conclusion. In our case, this means reasoning about one step of execution, for one instruction. Intuitively, if we show that our property holds for the execution of one arbitrary instruction, then it must hold for a sequence of many instructions.

We thus let:

$$\begin{aligned} \mathrm{IH} \;\triangleq\; & \forall p,b,e,a.\; \mathcal{V}(p,b,e,a) \mathrel{-\!\!*} \\ & \forall reg.\, \left\{(p,b,e,a); \mathop{\ast}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Rrightarrow v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet \end{aligned}$$

and assume $\triangleright \mathrm{IH}$; we then wish to show IH.

First, we consider the case where (p,b,e,a) is not a valid program counter (for instance, if it contains a non-executable capability, or an integer). Then the machine configuration will step into a Failed configuration. In that case, any full execution specification $(\{\cdot;\cdot\} \rightsquigarrow \bullet)$ trivially holds, and we are done.

In the case where (p,b,e,a) is a valid program counter, we will have to execute the next instruction of the program, pointed to by a. For (p,b,e,a) to be a valid program counter, the following needs to hold:

$$p \in \{\mathrm{RX}, \mathrm{RWX}\} \tag{2.1}$$

$$b \leq a < e \tag{2.2}$$

From (2.1), we can infer that $\mathcal{V}(p,b,e,a)$ will unfold to (at least) the following:

$$\mathop{\Large\ast}_{a\in[b,e)} \exists P, \boxed{\exists w,\, a \mapsto w * P(w)} * \triangleright \Box\, \forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Since we know that a is an address in the range $[b,e)$ (2.2), we can in particular infer that there exists a predicate P such that $\triangleright \Box\, \forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)$, for which the following invariant holds:

$$\boxed{\exists w,\, a \mapsto w * P(w)} \tag{2.3}$$

Ownership over $a \mapsto w$ is in fact required in order to apply any rule of the program logic (we need to be able to access memory for the instruction pointed to by pc). We will therefore first open the invariant (2.3) to get access to that resource.

Recall the invariant opening rule INV (Section 2.4.2). According to that rule, we can get access to the resources held inside the invariant *now*, as long as we give them back *after one execution step*. Since we wish here to reason about the execution of a single instruction, this is a perfectly good deal.

Once the invariant has been opened, the following propositions are added to our assumptions, for some word w (technically speaking, the Iris context also tracks the fact that these facts come from an invariant and must be given back next, but we choose to hide these details):[5]

$$a \mapsto w \tag{2.4}$$

$$\triangleright\, P(w) \tag{2.5}$$

Because pc points to a, and address a contains the word w, w should correspond to the (encoding of the) instruction to execute now. We thus reason by case analysis on $\mathrm{decode}(w)$.

This leads to as many cases as there are instructions in the machine. We will now detail a sub-case for the `load` instruction, which is one of the interesting cases. Many of the other cases are similar in nature.

Case: $\mathrm{decode}(w) = \texttt{load}\ r_{dst}\ r_{src}$.

[5] Notice that we directly get $a \mapsto w$ rather than $\triangleright a \mapsto w$, due to the fact that memory points-to are timeless.

We consider here the case where r_{dst} and r_{src} are two different registers, both different from pc. We also only consider the case where r_{src} contains a capability, which we are permitted to load from. In other words, our goal is as follows:[6]

$$
\begin{array}{l}
\triangleright \mathrm{IH} * a \mapsto w * \triangleright P(w) \\
\vdash \left\{ (p,b,e,a);\ \begin{array}{l} \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}, r_{dst}, r_{src}} r \Mapsto v * \mathcal{V}(v) \\ * \ r_{dst} \Mapsto w' * \mathcal{V}(w') \\ * \ r_{src} \Mapsto (p',b',e',a') * \mathcal{V}(p',b',e',a') \end{array} \right\} \leadsto \bullet
\end{array}
$$

As stated, we assume that (p',b',e',a') permits us to load from a'. We can thus infer the following two properties:

$$p' \in \{\mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}\} \tag{2.6}$$

$$b' \leq a' < e' \tag{2.7}$$

Just like before, we can from $\mathcal{V}(p',b',e',a')$ conclude that the following invariant holds, where P' is a predicate such that $\triangleright\Box\ \forall w, P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$:

$$\boxed{\exists w, a' \mapsto w * P'(w)} \tag{2.8}$$

We consider the (more interesting) case where $a \neq a'$. We can thus open the invariant (since it has not been opened already), meaning that we have for some word w_{src} the following (again, plus some invariant-tracking resources not shown here):

$$a' \mapsto w_{src} \tag{2.9}$$

$$\triangleright\ P'(w_{src}) \tag{2.10}$$

With these assumptions, we now have all the necessary resources to take a step in the program logic, using the rule for the `load` instruction (Figure 2.9). A feature of single-instruction rules of our program logic is that they include a built-in $\triangleright$ modality. In other words, after applying a single-instruction rule, we are "one execution step later", and we can remove one occurrence of $\triangleright$ for each assumption of our context. In particular, this means that we can turn $\triangleright$IH into IH, and similarly for $P(w)$ and $P'(w_{src})$. We now have to show:

$$
\begin{array}{l}
\mathrm{IH} * a \mapsto w * P(w) * a' \mapsto w_{src} * P'(w_{src}) \\
\vdash \left\{ (p,b,e,a+1);\ \begin{array}{l} \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}, r_{dst}, r_{src}} r \Mapsto v * \mathcal{V}(v) \\ * \ r_{dst} \Mapsto w_{src} \\ * \ r_{src} \Mapsto (p',b',e',a') * \mathcal{V}(p',b',e',a') \end{array} \right\} \leadsto \bullet
\end{array}
$$

We now have direct access to IH (our initial goal) as an assumption, so the proof is nearly done. Before we can invoke IH and conclude the goal, we must do two things: (a) close all the open invariants (as required by the invariant opening rule), and (b)

[6] We again omit details involving masks and update modalities, and refer to the Coq formalization for the full details.

show that the contents of all registers satisfies $\mathcal{V}$ (required by the definition of IH). (We actually need to show (b) *before* addressing (a), as we will make use of resources from the open invariants.)

Addressing (b), we already know that the contents of registers satisfy $\mathcal{V}$ for all registers except for r_{dst}—the only register whose contents were changed by the instruction. We must thus prove $\mathcal{V}(w_{src})$. Luckily, w_{src} is not a completely arbitrary word: it was accessible from available memory, so it must be safe as well. More precisely, from the invariant about a' (previously opened), we know that $P'(w_{src})$ holds, and furthermore we know that:

$$\Box\, \forall w, P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Owing to the fact that $\mathcal{V}(\cdot)$ is persistent, we can shave off the $\Box$ modality, and conclude that $\mathcal{V}(w_{src})$ holds, concluding the proof of (b).

Finally, addressing (a) is straightforward, since we did not change the contents of memory at either address a or a'. We can therefore close the invariants again, by giving up the same resources as we initially got from opening them, concluding the proof of (a) and thus the case of the proof for `load`.

In the proof sketch above, we followed one specific subcase of the proof for the `load` instruction. In the complete proof, we must go through all the possible cases of the semantics for the instruction. In some cases, the machine fails which terminates the proof easily (for instance, if the capability in r_{src} does not in fact allow reading memory, or if r_{src} does not in fact contain a capability). In some other cases, the machine does not fail, and the proof is similar to the case highlighted here but slightly different (for instance when r_{dst} and r_{src} are the same register).

The proofs for the other instructions of the machine follow a similar pattern. In particular, in the `store` case, the register state is not modified except for the `pc` register, but memory is modified. As such, closing the invariants is not as easy since we need to establish that the stored word is at least safe. This is established by using the fact that we assumed that the register only contains safe words. The case of the `restrict`, `subseg` and `lea` instructions require showing that a capability with smaller authority remains in the value relation $\mathcal{V}$, and the `jmp`, `jnz` and `mov` cases show that `pc` (or other registers) can be updated with arbitrary safe words. The other remaining cases are rather trivial, as they all only change a register state to an integer, which is always safe. □

2.6 Reasoning with capabilities: two examples

In this section, we return to the motivational examples introduced in Section 2.2, and show how to prove that they enforce the desired properties, using Cerise's reasoning tools, laid out in the previous sections.

2.6.1 Sharing a sub-buffer with an unknown adversary

Let us recall (on the right) the code for our buffer-sharing program, previously introduced in Figure 2.3. The labels code, data, secret and end denote addresses in memory. We wish to prove formally that the program can share the data between addresses data and secret (excluded), while protecting the integrity of the data at address secret.

```
code: mov r1 PC
      lea r1 [data-code]
      subseg r1 [data] [data+3]
      jmp r0
data:   'H', 'i', 0, ; public
secret: 42           ; secret
end:
```

Using the reasoning rules from our program logic, we can first prove a specification for the program, specifying its behavior from its first instruction up until the final jmp. The corresponding specification is as follows, where code_instrs is the list of integers corresponding to the encoded instructions of the program, i.e., code_instrs = map encodeInstr $[\texttt{mov}\ \mathsf{r}_1\ \mathsf{pc}; \ldots; \texttt{jmp}\ \mathsf{r}_0]$.

Lemma 1 (Program specification (10)).

$$\left\{ (\mathrm{RWX}, \mathsf{code}, \mathsf{end}, \mathsf{code}); \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto - * \\ [\mathsf{code}, \mathsf{data}) \mapsto \text{code\_instrs} \end{array} \right\} \leadsto$$

$$\left\{ \text{updatePcPerm}(w_{adv}); \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto (\mathrm{RWX}, \mathsf{data}, \mathsf{secret}, \mathsf{data}) * \\ [\mathsf{code}, \mathsf{data}) \mapsto \text{code\_instrs} \end{array} \right\}$$

One can read from the specification that executing the program will store in r_1 an RWX capability to the memory segment between addresses data and secret (our “buffer”), and pass control to the word w_{adv} found in register r_0.

Proving this specification is easy: it is enough to successively apply the program logic rule of each individual instruction found in the program.

This specification shows that the program ultimately jumps to the word initially passed in register r_0, but does not describe what happens after, in the case where this word points to a region containing unknown code. For this, we use the reasoning principles from Section 2.5.2 (built on top of the Fundamental Theorem), and derive a specification for a complete execution of the machine, see Lemma 2 below. The lemma specifies that, starting by executing our program, and given that r_0 contains a capability to a region containing unknown integers, then the machine is safe to run. Notice that we do not assume a points-to resource for the secret data: instead, this points-to will be part of an invariant—specifying that it contains the same secret value at every step—and we do not need to access that here.

Lemma 2 (Full execution specification (11)).

$$\left\{ (\mathrm{RWX}, \mathsf{code}, \mathsf{end}, \mathsf{code}); \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto - * \\ \mathop{\Large\ast}_{\substack{(r,v) \in reg, \\ r \notin \{\mathsf{pc}, \mathsf{r}_0, \mathsf{r}_1\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\mathsf{code}, \mathsf{data}) \mapsto \text{code\_instrs} * \\ [\mathsf{data}, \mathsf{secret}) \mapsto [\texttt{'H'}; \texttt{'i'}; 0] * \\ \mathop{\Large\ast}_{(a,z) \in adv} a \mapsto z * \lceil z \in \mathbb{Z} \rceil \end{array} \right\} \leadsto \bullet$$

Proof. By Lemma 1, the frame rule FRAGFRAME and the sequence rule SEQFULL, it suffices to show the following goal, which corresponds to a specification about the execution of the machine *after* the execution of the verified code:

$$\textit{Goal:}\qquad \left\{ ;\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathop{\ast}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1\}}} r \Mapsto z * \lceil z\in\mathbb{Z}\rceil * \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z * \lceil z \in \mathbb{Z}\rceil * \\ [\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs} * \\ [\mathsf{data},\mathsf{secret}) \mapsto [\texttt{'H'};\texttt{'i'};0] \end{array} \right\} \rightsquigarrow \bullet$$

We now rely on the reasoning rules derived from the Fundamental Theorem (Section 2.5.2). First, from the fact that the adversary region *adv* does not contain capabilities, we get using Corollary 1 that any capability on that region is safe, in particular we have $\mathcal{V}(\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv})$. Then, from Corollary 2 we get a specification for the execution of the machine starting from $\mathcal{V}(\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv})$ (recall that updatePcPerm is the identity on non-E capabilities):

$$\textit{Fact:}\qquad \forall reg.\ \left\{(\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv});\ \mathop{\ast}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

From this fact, we can prove our goal provided that we show that the contents of all machine registers satisfy $\mathcal{V}$. For registers other than r_0 and r_1, this holds by definition of $\mathcal{V}$, as we know they only contain integers. Register r_0 contains a capability to the adversary region, which we have already proved to be safe using Corollary 1. Finally, register r_1 contains the capability pointing to the public buffer. We can again leverage Corollary 1 to obtain $\mathcal{V}(\mathrm{RWX}, \mathsf{data}, \mathsf{secret}, \mathsf{data})$ from the memory points-to for the buffer ($[\mathsf{data},\mathsf{secret}) \mapsto [\texttt{'H'};\texttt{'i'};0]$), thus concluding the proof. □

Finally, from Lemma 2, established in the program logic, we wish to obtain a final result in terms of the operational semantics of the machine. The toplevel end-to-end theorem that we obtain is shown in Theorem 3. We consider a machine whose memory is initially loaded with our program and unknown adversarial code, and that starts by executing our verified code. The theorem establishes that the adversary will not be able to tamper with the value held at address `secret`: at every step of the execution, it will be unchanged and equal to 42.

Theorem 3 (End-to-end theorem: integrity of the secret data is preserved [12])**.** *Starting from an initial state of the machine (reg, mem) where:*

- *$prog \uplus adv \subseteq mem$, for $adv : [b_{adv}, e_{adv}) \to \mathsf{Word}$ and $prog : [\mathsf{code}, \mathsf{end}) \to \mathsf{Word}$*
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities: $\forall a. adv(a) \in \mathbb{Z}$;*

- *the initial state of registers satisfies:*
  $reg(\mathsf{pc}) = (\mathrm{RWX}, \mathsf{code}, \mathsf{end}, \mathsf{code})$*,*
  $reg(\mathsf{r}_0) = (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv})$*,*
  $reg(r) \in \mathbb{Z}$ *otherwise;*

Then, for any reg'*,* mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $mem'(\mathsf{secret}) = 42$*.*

Proof. We first invoke Theorem 1, choosing the memory invariant I and its domain D to be the invariant I_{buf} and domain D_{buf} defined below, asserting that the value at address `secret` is equal to 42:

$$\begin{aligned} &I_{buf} \triangleq \lambda m.\ m(\mathsf{secret}) = 42 \\ &\text{and } D_{buf} = \{\mathsf{secret}\}. \end{aligned}$$

Most side-conditions of the adequacy theorem can be easily discharged. What remains is the following specification in Iris:

$$\begin{array}{ll} & \boxed{\exists m, \mathop{\Large\ast}_{(a,w)\in m} a \mapsto w \ast \lceil \mathrm{dom}(m) = D_{buf} \rceil \ast \lceil I_{buf}(m) \rceil} \\ \textit{Goal:} & \vdash \left\{ (\mathrm{RWX}, \mathsf{code}, \mathsf{end}, \mathsf{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \mathop{\Large\ast}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0\}}} r \Mapsto z \ast \lceil z \in \mathbb{Z} \rceil \ast \\ \mathop{\Large\ast}_{(a,z)\in adv} a \mapsto z \ast \lceil z \in \mathbb{Z} \rceil \ast \\ \mathop{\Large\ast}_{\substack{(a,w)\in prog,\\ a\notin D_{buf}}} a \mapsto w \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

We can simplify this goal by unfolding the definition of I_{buf}, D_{buf}, $prog$ and massaging the goal to extract relevant points-to resources. The goal then becomes:

$$\begin{array}{ll} & \boxed{} \\ \textit{Goal:} & \vdash \left\{ (\mathrm{RWX}, \mathsf{code}, \mathsf{end}, \mathsf{code});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv}) \ast \\ \\ \mathop{\Large\ast}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0,\}}} r \Mapsto z \ast \lceil z \in \mathbb{Z} \rceil \ast \\ \\ \mathop{\Large\ast}_{(a,z)\in adv} a \mapsto z \ast \lceil z \in \mathbb{Z} \rceil \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

Note how the points-to resource for the `secret` address is held as part of the invariant, asserting that it contains the value 42 at each step. This simplified goal now follows from the full execution specification established earlier in Lemma 2 by applying the rule FULLFRAME, which concludes the proof. □

2.6.2 Creating a closure around local state

Let us now come back to the example introduced in Section 2.2.4, whose code is reproduced below. In this example, the control flow is somewhat more involved, as we have two separate pieces of known code that run at different times. The initialization code between init and code runs first, and creates a sentry capability before passing control to the unknown code. The code and data located between code and end are encapsulated in the sentry capability created by the initialization code. Because the sentry capability is exposed to the unknown code, the code it encapsulates may be invoked several times, incrementing the value of the counter each time.

We wish to prove formally that the value of the counter is correctly encapsulated. We prove that it remains non-negative at every step: starting from zero, it can only get incremented by the code routine encapsulated in the sentry capability.

```
init:
  mov r1 PC
  lea r1 [data-init]
  mov r2 r1
  lea r2 1
  store r1 r2
  lea r1 [code-data]
  subseg r1 [code] [end]
  restrict r1 E
  mov r2 0
  jmp r0
```

```
code:
  mov r1 PC
  lea r1 [data-code]
  load r1 r1
  load r2 r1
  add r2 r2 1
  store r1 r2
  mov r1 0
  jmp r0
data:
  ; will be:
  ; (RWX, init, end, data+1)
  0xFFFF,
  0 ; counter value
end:
```

Using the rules of our program logic, we can first prove a specification for the initialization code, shown in Lemma 3. This specification describes the behavior of the code between init and code, where init_instrs denote the corresponding list of encoded instructions.

Lemma 3 (Specification for the initialization code ⑬)**.**

$$\left\{ (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{init});\ \begin{array}{l} \mathsf{r_0} \Mapsto w_{adv} * \mathsf{r_1} \Mapsto - * \mathsf{r_2} \Mapsto - * \\ \mathsf{data} \mapsto - * [\mathsf{init}, \mathsf{code}) \mapsto \text{init\_instrs} \end{array} \right\} \rightsquigarrow$$

$$\left\{ \text{updatePcPerm}(w_{adv});\ \begin{array}{l} \mathsf{r_0} \Mapsto w_{adv} * \mathsf{r_1} \Mapsto (\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code}) * \mathsf{r_2} \Mapsto 0 * \\ \mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data} + 1) * \\ [\mathsf{init}, \mathsf{code}) \mapsto \text{init\_instrs} \end{array} \right\}$$

From this specification, one can read that running the initialization code will store in register $\mathsf{r_1}$ a sentry capability to $[\mathsf{code}, \mathsf{end})$, and write at address data an $\textsc{rwx}$ capability pointing to the location holding the counter value. The initialization code then passes control to the unknown word w_{adv} stored in $\mathsf{r_0}$.

We can also use the program logic rules to prove a specification for the code routine in $[\mathsf{code}, \mathsf{data})$ which increments the counter, and which will run each time the sentry capability is invoked. The specification appears in Lemma 4, where code_instrs refers to the list of encoded instructions for the routine.

Lemma 4 (Specification for the increment routine (14)).

$$\begin{array}{l}
\boxed{[\mathsf{code},\mathsf{data}) \mapsto \text{code\_instrs}}\,, \\
\boxed{\mathsf{data} \mapsto (\textsc{rwx},\mathsf{init},\mathsf{end},\mathsf{data}+1)}\,, \boxed{\exists n.\,(\mathsf{data}+1) \mapsto n * \lceil n \geq 0 \rceil} \\
\vdash \{(\textsc{rx},\mathsf{code},\mathsf{end},\mathsf{code}); \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto -\} \rightsquigarrow \\
\quad \{\text{updatePcPerm}(w_{cont}); \exists n.\, \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto 0 * \mathsf{r}_2 \Mapsto n\}
\end{array}$$

This specification assumes a number of Iris invariants, describing the contents of the $[\mathsf{code},\mathsf{end})$ memory region. Indeed, because the increment routine is invoked by unknown code, it cannot make many assumptions about the state of the machine. The only thing that it can assume is that previously established invariants still hold (because, by definition, capability-safe unknown code has to preserve invariants).

The specification thus assumes, as invariants: 1) that the region $[\mathsf{code},\mathsf{data})$ contains the code of the routine; 2) that data contains the $\textsc{rwx}$ capability to the counter value previously stored there by the initialization code, and finally 3) that the counter value (at address $\mathsf{data}+1$) is a non-negative integer.

The specification asserts that the routine can run, starting with pc containing an $\textsc{rx}$ capability to the $[\mathsf{code},\mathsf{end})$ region, while preserving the invariants. (In particular, this means that incrementing the counter indeed preserves the fact that it is a non-negative integer.) Recall that the $\textsc{rx}$ permission in pc corresponds to what one gets after jumping to a sentry capability.

Finally, we prove as before a specification proving safety of complete executions, starting from the initialization code, then followed by the execution of unknown code, including its possible invocations of the sentry capability. This specification appears below in Lemma 5.

Lemma 5 (Full execution specification (15)).

$$\begin{array}{l}
\boxed{\exists n.\,(\mathsf{data}+1) \mapsto n * \lceil n \geq 0 \rceil} \\
\vdash \left\{ (\textsc{rwx},\mathsf{init},\mathsf{end},\mathsf{init});\ \begin{array}{l}
\mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto - * \\
\mathop{\Huge *}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\
[\mathsf{init},\mathsf{code}) \mapsto \text{init\_instrs} * \\
[\mathsf{code},\mathsf{data}) \mapsto \text{code\_instrs} * \mathsf{data} \mapsto - * \\
\mathop{\Huge *}_{(a,z)\in adv}\, a \mapsto z * \lceil z \in \mathbb{Z} \rceil
\end{array} \right\} \rightsquigarrow \bullet
\end{array}$$

Proof. By using Lemma 3 (the specification for the initialization code), the frame rule FragFrame and sequence rule SeqFull, it is enough to show the following goal, which specifies the execution of the machine after the initialization code has run:

$$\textit{Goal:}\quad \begin{array}{l} \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \\ \vdash \left\{ ;\; \begin{array}{l} \mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto * \mathsf{r}_2 \Mapsto * \\ \mathop{\Huge *}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}} r \Mapsto z * \lceil z\in\mathbb{Z}\rceil * \\ [\mathsf{init},\mathsf{code}) \mapsto \mathsf{init\_instrs} * \\ [\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs} * \\ \mathsf{data} \mapsto * \\ \mathop{\Huge *}_{(a,z)\in adv} a \mapsto z * \lceil z\in\mathbb{Z}\rceil \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

We then allocate two new invariants, one containing the code of the sentry capability, the other the points-to resource at address data.

$$\textit{Goal:}\quad \begin{array}{l} ,\, , \\ \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \\ \vdash \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv});\; \begin{array}{l} \mathsf{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto (\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code}) * \mathsf{r}_2 \Mapsto 0 * \\ \mathop{\Huge *}_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}} r \Mapsto z * \lceil z\in\mathbb{Z}\rceil * \\ [\mathsf{init},\mathsf{code}) \mapsto \mathsf{init\_instrs} * \\ \mathop{\Huge *}_{(a,z)\in adv} a \mapsto z * \lceil z\in\mathbb{Z}\rceil \end{array} \right\} \rightsquigarrow \bullet \end{array}$$

From Corollary 1 and the fact that the adversary region *adv* does not contain capabilities, we get that any capability on that region is safe, and therefore that $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ holds. From Corollary 2, we get that a full execution starting from $(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ is safe:

$$\textit{Fact:}\qquad \forall reg.\, \left\{ (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}); \mathop{\Large *}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

In combination with rule FULLFRAME, this fact allows us to conclude the proof, *provided we can prove safety of values stored in all registers*. We have already proved the capability in r_0 to be safe. Registers r_2 to r_{31} contain integers, so they are safe by definition of $\mathcal{V}$. Safety of the sentry capability created by the initialization code and stored in r_1 remains to be proven.

$$\textit{Goal:}\quad \begin{array}{l} \boxed{[\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs}}, \boxed{\mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \\ \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \\ \vdash \mathcal{V}(\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code}) \end{array}$$

By definition of $\mathcal{V}$ and $\mathcal{E}$, this goals unfolds to the following:

$$\textit{Goal:}\quad \begin{array}{l} \boxed{[\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs}}, \boxed{\mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \\ \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \\ \vdash \triangleright \Box\, \forall reg, \left\{ (\textsc{rx}, \mathsf{code}, \mathsf{end}, \mathsf{code}); \mathop{\Large *}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet \end{array}$$

For technical reasons, we can shave off the later modality ($\triangleright$) in front of the goal (we refer to the Coq formalization for more details). The persistent modality ($\Box$) is more interesting: it expresses the fact that safety of the callback should only depend on persistent assumptions. This corresponds to the fact that the callback may be invoked several times, in future execution states and because of this it cannot rely on non-persistent assumptions that only hold at the callback's creation time. Fortunately, invariants are persistent, so they remain available for proving the callback's safety.

Then, let us name w_0 the contents of register r_0: we get to assume $\mathcal{V}(w_0)$ (as for the contents of other registers). By using Lemma 4 (the specification for the increment routine) with rules FRAGFRAME and SEQFULL, it is enough to prove the following goal, which asserts safety of the execution *after* passing control back to unknown code by jumping to w_0:

$$\textit{Goal:} \qquad \vdash \left\{ \text{updatePcPerm}(w_0); \begin{array}{l} \exists n.\, \mathsf{r}_0 \Mapsto w_0 * \mathsf{r}_1 \Mapsto 0 * \mathsf{r}_2 \Mapsto n * \\ \mathop{\scalebox{1.5}{$\ast$}}_{(r,v)\in reg, r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1,\mathsf{r}_2\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Informally, the increment routine returns to the unknown code by passing control to some unknown word provided in r_0: it is safe to do so, since such word can be assumed to be itself safe. Formally speaking, we know $\mathcal{V}(w_0)$, so we apply Corollary 2 which concludes the proof. □

Similarly to the previous example, we derive from Lemma 5 a toplevel theorem which only refers to the operational semantics of the machine, shown below in Theorem 4. We consider a machine initially loaded with our program and unknown adversarial code. The theorem establishes that the value of the counter is properly encapsulated: at every step of the execution, it will be a non-negative integer.

Theorem 4 (End-to-end theorem: integrity of the counter value is preserved [16])**.** *Starting from an initial state of the machine (reg, mem) where:*

- *$prog \uplus adv \subseteq mem$, for $adv : [b_{adv}, e_{adv}) \to$ Word and $prog : [\mathsf{init}, \mathsf{end}) \to$ Word*
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities: $\forall a. adv(a) \in \mathbb{Z}$;*
- *the initial state of registers satisfies:*
  $reg(\mathsf{pc}) = (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{init})$,
  $reg(\mathsf{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$,
  $reg(r) \in \mathbb{Z}$ otherwise;

Then, for any reg', mem', if $(reg, mem) \longrightarrow^* (reg', mem')$, then $mem'(\mathsf{data}+1) \geq$ 0.

Proof. We invoke Theorem 1, with invariant and domain I_{cnt} and D_{cnt} defined as follows:

$$I_{cnt} \triangleq \lambda m. m(\mathsf{data}+1) \geq 0$$
$$\text{and } D_{cnt} = \{\mathsf{data}+1\}$$

The main step of the proof is to show that the full execution specification for the initial machine configuration holds, as stated by the theorem. After some basic unfolding of definitions, it is easy to show that it follows from the specification we previously established in Lemma 5. □

2.7 Dynamic Memory Allocation and Closures

In the previous sections, we have shown how to use capabilities for memory protection and compartmentalization in the setting of relatively simple scenarios. In particular, the examples that we have presented so far only relied on memory allocated statically as part of the initial program region.

We now investigate how we can use and reason about more complicated programming patterns. More precisely, we show how we can implement features found in higher-level languages, such as dynamic memory allocation and function calls which guarantee encapsulation of local variables. Additionally, we implement an `assert` routine which we use to formally express properties about dynamically allocated memory.

This section focuses on presenting the aforementioned higher-level building blocks (§2.7.1–2.7.3), an updated adequacy theorem that incorporates the use of these components (§2.7.4), then followed by a simple illustrative example (§2.7.5). In Section 2.8, we then apply them to build a larger, more significant case study, demonstrating how these building blocks can work at scale.

2.7.1 Dynamic memory allocation as a library routine

We show how dynamic memory allocation can be implemented as a library, for which: 1) we prove an Iris specification making it usable from verified code, and 2) we show that it is safe to share with untrusted code, so that an adversary can also use the library to allocate memory for its own uses.

Note that this task is made easier by the fact that we do not attempt to provide a way of deallocating memory. As such, memory provided by the allocation routine is never reclaimed. We leave deallocation for future work, as it likely requires a significantly more complex runtime mechanism to ensure that no dangling capabilities remain pointing to previously allocated memory regions [52, 163].

Concretely, we implement our allocator library as a simple bump-pointer allocator. The library provides a `malloc` entry point, to be called with an integer argument n, which works as follows:

1. the routine encapsulates a contiguous region of memory $[b,e)$, as well as a capability $(\textsc{rwx}, b, e, a)$ where the interval $[b,a)$ represents already allocated memory, and $[a,e)$ represents memory that can still be allocated;

2. the routine checks that the input size n is strictly positive;

3. if $a+n$ is greater than e, the routine fails (there is not enough memory available);

4. otherwise, it then records that memory has been allocated by updating its internal capability to $(\mathrm{RWX}, b, e, a+n)$, and returns to the caller the capability $(\mathrm{RWX}, a, a+n, a)$.

Figure 2.11 outlines the code for our simple `malloc` implementation. The code assumes that it is stored in memory in an interval $[b_m, b_{mid})$ and that b_{mid} points to a capability $(\mathrm{RWX}, b_{mid}, e_m, a)$ giving access to: itself (so it can be updated), and the memory pool (between address $b_{mid}+1$ and e_m). For simplicity, we assume that the non-allocated memory is already initialized to 0. These requirements are represented by the following invariant [17]:

$$\mathrm{mallocInv}(b_m, e_m) \triangleq \boxed{\begin{array}{ll} \exists b_{mid}, a, & [b_m, b_{mid}) \mapsto \texttt{malloc\_instrs} * \\ & b_{mid} \mapsto (\mathrm{RWX}, b_{mid}, e_m, a) * \\ & [a, e_m) \mapsto [0 \cdots 0] * \\ & \lceil b_{mid} < a \leq e_m \rceil \end{array}}$$

The core property of our safe `malloc` is that is does not hand out the same addresses across multiple dynamic allocations. This can be expressed elegantly in separation logic, by specifying that `malloc` hands out points-to resources for the allocated memory. Indeed, points-to resources ($a \mapsto w$) express full ownership over the data at address a: possessing a resource $a \mapsto w$ guarantees that one is the only owner of address a.

Consequently, remark that the invariant holds memory points-to for the region corresponding to non-allocated memory (between a and e_m), but not for the memory that has already been allocated (between $b_{mid}+1$ and a): these resources have been handed out to previous callers of the library.

We show below the specification for `malloc` [18]. First, note that because `malloc` can fail if it runs out of memory or is given a wrong size, the specification documents that the resulting execution state is either Running or Failed. In the case where it does not fail, we can read that `malloc` hands out points-to resources for the allocated range in its post-condition: this expresses the fact that no piece of code but the caller of `malloc` can access the newly allocated memory.

$$\begin{array}{l} \boxed{\mathrm{mallocInv}(b_m, e_m)} \\ \vdash \left\{ (\mathrm{RX}, b_m, e_m, b_m);\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto n * \\ r_2, r_3, r_4 \Mapsto - \end{array} \right\} \rightsquigarrow \\ \quad \left\{ s. \begin{array}{ll} \lceil s = \mathsf{Running} \rceil * & \mathsf{pc} \Mapsto \mathsf{updatePcPerm}(w_0) * \\ & \exists b_a, e_a, \lceil b_a + n = e_a \rceil * \\ & \quad \mathsf{r}_0 \Mapsto w_0 * \\ & \quad \mathsf{r}_1 \Mapsto (\mathrm{RWX}, b_a, e_a, b_a) * \\ & \quad \mathop{\Large *}_{a \in [b_a, e_a)} a \mapsto 0 * \\ & \quad \mathsf{r}_2, \mathsf{r}_3, \mathsf{r}_4 \Mapsto 0 \\ \vee \lceil s = \mathsf{Failed} \rceil & \end{array} \right\} \end{array}$$

The `malloc` routine can furthermore be encapsulated using a sentry capability, which can be shown to be safe to share with an adversary (Lemma 6).

```
;; r1: integer determining the number
;; of words to allocate
;;
;; malloc fails if size <= 0 or if it
;; does not have enough space left
;;
;; returns in r1 a capability to the
;; allocated memory
bm:
  lt r3 0 r1 ;; check that size > 0
  mov r2 pc  ;; jmp after fail if
  lea r2 4   ;; yes; continue and
  jnz r2 r3  ;; fail if not
  fail
xm:
  mov r2 pc
  lea r2 [bmid - xm]
  ;; r2 = (RWX, bm, em, bmid)
  load r2 r2
  ;; r2 = (RWX, bmid, em, a)
  geta r3 r2
  lea r2 r1
  ;; r2 = (RWX, bmid, em, a+size)
```

```
  geta r1 r2
  mov r4 r2
  subseg r4 r3 r1
  sub r3 r3 r1
  lea r4 r3
  mov r3 r2
  sub r1 0 r1
  lea r3 r1
  getb r1 r3
  lea r3 r1
  ;; r3 = (RWX, bmid, em, bmid)
  store r3 r2
  ;; bmid <- (RWX, bmid, em, a+size)
  mov r1 r4
  ;; r1 = (RWX, a, a+size, a)
  mov r2 0
  mov r3 0
  mov r4 0
  jmp r0
bmid: (RWX, bmid, em, a)
;; ... already allocated memory ...
a:
;; ... free memory ...
em:
```

Figure 2.11: A simple `malloc` subroutine

Lemma 6 (`malloc` is safe ⑲)**.** $\mathrm{mallocInv}(b_m, e_m) \mathrel{-\!\!*} \mathcal{V}(\mathrm{E}, b_m, e_m, b_m)$

The proof is comparable to the proof that $\mathcal{V}(\mathrm{E}, \mathsf{code}, \mathsf{end}, \mathsf{code})$ on page 65. It relies on the `malloc` specification and the fundamental theorem.

2.7.2 Runtime checks: an `assert` routine

The final end-to-end theorems presented so far in Section 2.6 rely on establishing that a certain memory location satisfies a given invariant. This requires the relevant location is statically allocated in memory and thus known in advance, thus making it easy to tie it to an Iris invariant.

However, when using our `malloc` routine, we typically wish to enforce properties about the contents of dynamically allocated memory locations, whose address is, by definition, not known in advance. To address this issue, we implement an `assert` routine, to be linked alongside programs relying on `malloc`. One can invoke `assert` to dynamically test whether the contents of two registers are equal; if the test fails, `assert` sets a flag “assert has failed” at a fixed location in memory.

The idea is then that, to assert that some property holds about a piece of dynamically allocated memory, one can check dynamically whether it holds using `assert`. Then, one can *prove* that each `assert` check succeeds (meaning that the property indeed holds). Consequently, as a property of the whole execution, one gets that, at every step, the assert flag (initialized at 0) remains at 0 and is never set to 1 by `assert`.

The private memory of the `assert` routine is described by the following invariant ⑳:

$$\mathrm{assertInv}(b_a, e_a, a_{flag}) \triangleq \boxed{\begin{array}{l} \exists a_{cap},\ [b_a, a_{cap}) \mapsto \texttt{assert\_instrs} * \\ \qquad a_{cap} \mapsto (\mathrm{RW}, a_{flag}, a_{flag}+1, a_{flag}) * \\ \qquad \lceil a_{cap}+1 = a_{flag} \wedge a_{flag}+1 = e_a \rceil \end{array}}$$

The address a_{flag} denotes the address of the "assert flag", which is initialized to 0 and set to 1 by the routine in case of failure. As we are interested in using `assert` in programs where we can prove that the equality check succeeds, we establish the following specification ㉑, which asserts in a separate invariant that a_{flag} remains at 0. Registers r_4 and r_5 contain the two integers which are compared by the routine; we thus require that they are equal.

$$\boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\mathrm{RX}, b_a, e_a, b_a);\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_0 * \\ \mathsf{r}_4 \Mapsto n * \\ \mathsf{r}_5 \Mapsto n \end{array} \right\} \rightsquigarrow \left\{ \mathrm{updatePcPerm}(w_0);\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_0 * \\ \mathsf{r}_4, \mathsf{r}_5 \Mapsto 0 \end{array} \right\}$$

Note that, as opposed to `malloc`, the `assert` routine should only be shared with *verified* code, which calls it according to the specification above. Were `assert` shared with an unknown adversary, the adversary could simply call the routine with two different integers, setting the flag to 1, thus invalidating any guarantees established by verified code. Technically speaking, we cannot prove safety of the `assert` routine from the specification above: if we try to prove $\mathcal{V}(\mathrm{E}, b_a, e_a, b_a)$, then we get that registers r_4 and r_5 contain two unknown (valid) words, which could be two different integers. In that case, we cannot use the specification above, as we would violate the invariant specifying that a_{flag} stays at 0.

2.7.3 A secure heap-based calling convention

We define a heap-based calling convention that uses `malloc` to dynamically allocate activation records. An activation record is encapsulated in a closure that reinstates its caller's local state, and continues execution from its point of creation. Conceptually, our heap-based calling convention can be seen as a continuation-passing style calling convention (one passes control to the callee, giving it a continuation for returning to the caller). This is similar to the calling convention that was used for instance in the SML/NJ compiler to implement an extension of Standard ML with call/cc [11] (in the setting of a traditional computer architecture).

In the setting of a capability machine, our calling convention is furthermore *secure* in the sense that it enforces local state encapsulation. In other words, one can use it to pass control to unknown adversarial code, while protecting local data of the caller, thanks to the use of sentry capabilities to implement the continuation. Note that this calling convention does not enforce well-bracketed control flow (another desirable property); see [55, 131, 132] for stack-based calling conventions that do.

We provide a `call` macro implementing the calling convention, invoked as `call` *target locals params*, where *target* is the name of the register containing a pointer to the code to invoke, *locals* is the list of registers whose content corresponds to the local state to reinstate upon return, and *params* is the list of registers containing the parameters to the call (passed to the callee). Its implementation appears in Figure 2.12, and a representation of the corresponding memory layout in Figure 2.13. (Because `call` is defined as a macro, its code is used inline as part of a bigger program, here stored between addresses code and end.)

Before passing control to the callee, the call macro does the following:

1. Invoke `malloc` to dynamically allocate a region of memory $[l, l_{end})$ to store the local state from the registers specified in *locals*.

2. Allocate a region of memory $[act, act_{end})$ to store the activation record, composed of: activation code, a capability to the region $[l, l_{end})$, and a capability to the instruction of the program following the call.

3. Create a sentry capability $(\mathrm{E}, act, act_{end}, act)$ encapsulating the activation record; this is capability for returning to the caller which is passed to the callee.

4. Clear all registers except those in *params*.

5. Jump to *target*.

When the callee passes back control to the caller by jumping to the continuation, the code stored in the activation run first. It loads the capability pointing to local state, and returns to the old program counter set up by the call macro. As the last step, the macro will finally:

6. Restore the local state into the relevant registers from the activation record.

We show below the specification for the code of the macro up to step 5 (the jump to the target address) ㉒. Since the `malloc` routine is invoked by the macro, the specification relies on the corresponding invariant for `malloc`. The parameters of the macro are *params*, *locals* and *target*, respectively denoting the list of registers containing the parameters to the call, the list of registers containing local state, and the register containing the capability to jump to. The list of (encoded) instructions `act_instrs` denote the concrete instructions making up the activation code (in Figure 2.12 they are written as `act_instr1`...`act_instr5` ㉓), which are not shown here for simplicity.

The post-condition of the specification describes the state immediately after the jump, where: the activation record has been allocated and initialized in $[act, act_{end})$; register r_0 contains an enter capability pointing to the activation record, and the local data has been copied to a newly allocated region $[l, l_{end})$.

```
; initially, PC = (RWX, code, end, a)
;            target = register containing the address to jump to
;            locals, params = lists of register names
; locals, params and target are parameters of the macro;
; they are in practice instantiated with concrete values
code:
  ...
a:
  malloc (length locals)        ; 1. allocate and store local state
  store_locals r1 locals
  mov r6 r1
  malloc 7                      ; 2. allocate region for activation record
  mov r0 r1
  store act_instr1              ;    store the activation code
  lea r0 1
  ...
  store act_instr5
  lea r0 1
  store r0 r6                   ;    store the capability to locals
  lea r0 1
x:
  mov r1 pc                     ;    prepare and store the continuation
  lea r1 [cont - x]
  store r0 r1
  lea r0 -6                     ; 3. create the return capability
  restrict r0 E
  rclear RegName\{PC,r0,r1} ∪ params ; 4. clear all registers except params
  jmp target                    ; 5. jump to target
cont:
  restore_locals r1 locals      ; 6. reinstate local state
  ...
data:
  (RO, table, end, table)       ; environment table
table:
  (E, bm, em, bm)               ; entry point to the malloc subroutine
  ...                           ; possibly other routines
end:
```

Figure 2.12: Heap-based calling convention, with a the first instruction in the call macro

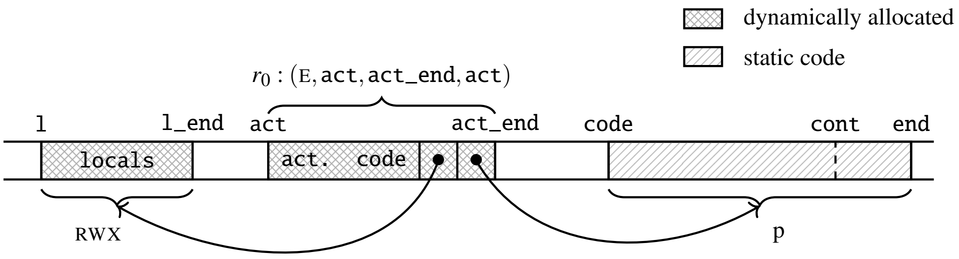


Figure 2.13: Memory layout dynamically created by the calling convention

$$
\boxed{\mathrm{mallocInv}(b_m, e_m)}
$$

$$
\vdash \left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{a});\;
\begin{array}{l}
[\mathsf{a}, \mathsf{cont}) \mapsto \mathtt{call\_instrs} \ast \\
\mathsf{data} \mapsto (\mathrm{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) \ast \\
\mathsf{table} \mapsto (\mathrm{E}, b_m, e_m, b_m) \ast \\
\mathit{params} \Mapsto \mathit{pws} \ast \mathit{locals} \Mapsto \mathit{lws} \ast \mathit{target} \Mapsto w_{adv} \ast \\
\mathop{\ast}\limits_{\substack{(r,v) \in reg, \\ r \notin \{\mathsf{pc}, \mathit{target}\} \\ r \notin \mathit{params} \,\cup\, \mathit{locals}}} r \Mapsto v
\end{array}
\right\} \rightsquigarrow
$$

$$
\left\{ \mathrm{updatePcPerm}(w_{adv});\;
\begin{array}{l}
\exists \mathit{act}, \mathit{act}_{end}, l, l_{end}, \mathit{reg}', \\
\mathsf{r}_0 \Mapsto (\mathrm{E}, \mathit{act}, \mathit{act}_{end}, \mathit{act}) \ast \\
\mathsf{data} \mapsto (\mathrm{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) \ast \\
\mathsf{table} \mapsto (\mathrm{E}, b_m, e_m, b_m) \ast \\
\mathit{params} \Mapsto \mathit{pws} \ast \mathit{target} \Mapsto w_{adv} \ast [l, l_{end}) \mapsto \mathit{lws} \ast \\
[\mathit{act}, \mathit{act}_{end}) \mapsto \mathtt{act\_instrs} \\
\qquad\qquad \mathbin{++} [(\mathrm{RWX}, l, l_{end}, l_{end}); \\
\qquad\qquad\quad (p, \mathsf{code}, \mathsf{end}, \mathsf{cont})] \ast \\
\mathop{\ast}\limits_{\substack{(r,v) \in reg', \\ r \notin \{\mathsf{pc}, \mathit{target}, \mathsf{r}_0\} \\ r \notin \mathit{params}}} r \Mapsto v
\end{array}
\right\}
$$

It is then up to the user of the `call` macro to establish that the capability in r_0 is safe to share with the (possibly unknown) callee. This can be done with the help of the specification for the activation code (24), shown next:

$$
\vdash \left\{ (\mathrm{RX}, \mathit{act}, \mathit{act}_{end}, \mathit{act});\;
\begin{array}{l}
\mathsf{r}_1 \Mapsto - \ast \mathsf{r}_2 \Mapsto - \ast \\
[\mathit{act}, \mathit{act}_{end}) \mapsto \mathtt{act\_instrs} \mathbin{++} \\
\qquad\qquad [(\mathrm{RWX}, l, l_{end}, l_{end}); \\
\qquad\qquad\; (p, \mathsf{code}, \mathsf{end}, \mathsf{cont})]
\end{array}
\right\} \rightsquigarrow
$$

$$
\left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{cont});\;
\begin{array}{l}
\mathsf{r}_1 \Mapsto - \ast \mathsf{r}_2 \Mapsto (\mathrm{RWX}, l, l_{end}, l) \ast \\
[\mathit{act}, \mathit{act}_{end}) \mapsto \mathtt{act\_instrs} \mathbin{++} \\
\qquad\qquad [(\mathrm{RWX}, l, l_{end}, l_{end}); \\
\qquad\qquad\; (p, \mathsf{code}, \mathsf{end}, \mathsf{cont})]
\end{array}
\right\}
$$

One can read from this specification that the activation code passes control back to the caller (at address cont), while loading in register r_2 a capability to the region holding the local state, which can be then loaded back into the corresponding registers by the `restore_locals` macro (step 6, which we do not detail here).

To sum up, the calling convention presented here allows one to make a “function call” as one would do in a higher-level language, while protecting local data of the caller. The code invoked this way can be completely untrusted: in particular, it does not need to implement the calling convention itself for the local state encapsulation guarantees to hold. (But of course it might never “return” and pass control back to the caller.)

In Section 2.7.5, we demonstrate the use of this heap-based calling convention on a simple example, showing the interaction of its local state encapsulation guarantees with read-only capabilities.

2.7.4 Adequacy in the Presence of Dynamically Allocated Memory

We can now provide an updated version of the adequacy theorem (Theorem 1) which directly incorporates the `malloc` and `assert` library routines. Instead of establishing that a memory invariant is always preserved at each step, the new adequacy theorem establishes that the flag held by `assert` is never modified.

Theorem 5 (Updated adequacy (25)). *Given memory fragments* $prog : [b,e) \to \mathrm{Word}$, $malloc : [b_m, e_m) \to \mathrm{Word}$, $assert : [b_a, e_a) \to \mathrm{Word}$, *and for any memory fragment* $adv : [b_{adv}, e_{adv}) \to \mathrm{Word}$, *assuming that:*

1. *the initial state of memory mem satisfies:*

$$prog \uplus malloc \uplus assert \uplus adv \subseteq mem$$

2. $[b_m, e_m)$ *contains the* `malloc` *routine;*

3. $[b_a, e_a)$ *contains the* `assert` *routine and its flag at address* a_{flag}*;*

4. *the assertion flag is initially set to 0:*

$$mem(a_{flag}) = 0$$

5. *prog contains a table linking to malloc and assert:*

$$\begin{aligned}\exists data, table, mem(data) &= (\text{RO}, table, table+2, table)\\ mem(table) &= (\text{E}, b_m, e_m, b_m)\\ mem(table+1) &= (\text{E}, b_a, e_a, b_a)\end{aligned}$$

6. *the adversary region contains no capabilities except for a table linking to malloc:*

$$\begin{aligned}&\exists data_{adv}, table_{adv}, \forall a \in \mathrm{dom}(adv) \setminus \{data_{adv}, table_{adv}\},\\ &\qquad adv(a) \in \mathbb{Z}\\ &\qquad adv(data_{adv}) = (\text{RO}, table_{adv}, table_{adv}+1, table_{adv})\\ &\qquad adv(table_{adv}) = (\text{E}, b_m, e_m, b_m)\end{aligned}$$

7. *the initial state of registers reg satisfies:*

$$reg(\mathtt{pc}) = (\text{RWX}, b, e, b), \quad reg(\mathtt{r}_0) = (\text{RWX}, b_{adv}, e_{adv}, b_{adv}),$$
$$reg(r) \in \mathbb{Z} \text{ otherwise}$$

8. *the proof in the program logic that the initial configuration is safe given the invariants:*

$$\forall reg,$$

$$\boxed{\mathrm{mallocInv}(b_m, e_m)}\,,\ \boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}\,,\ \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\mathrm{RWX}, b, e, b);\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathrm{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \bigast_{\substack{(r,v)\in reg, \\ r \notin \{\mathsf{pc}, \mathsf{r}_0\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \bigast_{\substack{(a,w)\in prog, \\ a \notin \{data, table, table+1\}}} a \mapsto w * \\ data \mapsto (\mathrm{RO}, table, table+2, table) * \\ table \mapsto (\mathrm{E}, b_m, e_m, b_m) * \\ table+1 \mapsto (\mathrm{E}, b_a, e_a, b_a) * \\ \bigast_{\substack{(a,z)\in adv \\ a \notin \{data_{adv}, table_{adv}\}}} a \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ data_{adv} \mapsto (\mathrm{RO}, table_{adv}, table_{adv}+1, \\ \qquad\qquad\qquad\qquad table_{adv}) * \\ table_{adv} \mapsto (\mathrm{E}, b_m, e_m, b_m) \end{array} \right\} \rightsquigarrow \bullet$$

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(a_{flag}) = 0$.

Theorem 5 assumes that the `malloc` and `assert` routines are loaded in memory disjoint from both *prog* and *adv*. Furthermore, the `assert` routine must have its flag initialized to 0. The verified program *prog* is given access to both the `malloc` and `assert` routines. The adversary program *adv* is given access to `malloc`. We assume that *prog* contains the code and a table that has been filled by a linker with capabilities giving access to the two routines. Likewise, we assume that *adv* contains its program (arbitrary integers) and a table filled by the linker with the capability to the `malloc` routine. Similarly to the first adequacy theorem, the theorem states that if the capability machine starts with the capability pointing to *prog* in the program counter, and if it has been proved in the program logic that the machine can run until completion, then the assertion flag is *never* modified.

In what follows, Lemma 5 will thus allow us to prove end-to-end theorems saying that the assertion flag will still be unset after a full execution. This corresponds to the end-to-end theorems of Swasey et al. [138] which are also phrased in terms of an assert primitive (albeit in a high-level language) that untrusted code does not get access to. Of course, such results remain a bit artificial: ultimately, in real systems, we are not directly interested in the contents of assertion flags in the system's memory, but rather in the system's interaction with the outside world: network communication, the content of displays etc. Our approach can be extended to reason about such properties, but we don't go into details here. Instead, we refer to Strydonck et al. [135], where we have done exactly this extension, by adding MMIO and external event traces to our operational semantics and using Iris invariants and ghost state to reason about them. This results in end-to-end theorems that prove security properties about the external

```
; initially, PC = (RWX, code, end, code)
;             r1 = (unknown) pointer to adversary function
code:
  malloc 1                  ; r1 = (RWX, b, b+1, b) where b is fresh
  mov r3 r1                 ; r3 = (RWX, b, b+1, b)
  mov r4 r1                 ; r4 = (RWX, b, b+1, b)
  store r3 1                ; b <- 1
  restrict r4 RO            ; r4 = (RO, b, b+1, b)
  call r1 [r3] [r4]         ; call macro that jumps to r1, keeps r3
                            ; as local state and passes r4 as parameter
  load r1 r3                ; r1 = 1, as long as b was not changed
                            ; during call
  mov r2 1
  assert r1 r2              ; assert (r1 = 1)
  halt
data:
  (RO, table, end, table) ; environment table
table:
  (E, bm, em, bm)           ; entry point to the malloc subroutine
  (E, ba, ea, ba)           ; entry point to the assert subroutine
end:
```

Figure 2.14: Program passing a read-only capability to unknown callee

event traces of a system, which we regard as a more realistic end goal of a verification effort.

2.7.5 Application: read-only sharing of dynamically allocated memory

We now present an example program sharing a read-only capability with adversary code, showcasing the combined use of the `malloc` (Section 2.7.1) and `assert` (Section 2.7.2) routines, the secure calling convention (Section 2.7.3), and exercising our updated adequacy theorem (Section 2.7.4).

Figure 2.14 shows the implementation of our program of interest. The program dynamically allocates a region of size 1, into which it stores the integer 1. Next, it creates a copy of the newly created capability, which is then restricted to read-only (RO). This restricted capability is shared with an unknown callee, while the original copy is kept as local state. Upon return, an `assert` statement checks that the region indeed still contains 1. We then wish to prove that the final assertion always succeeds.

Notice that in this example, control is passed to untrusted code, corresponding to the first scenario in Figure 2.2a. However, we also allow the callee to return, i.e. jump to a callback. This is achieved using our calling convention to create a secure two-way boundary between known code and the unknown callee.

In order to prove that the `assert` statement succeeds, we rely on two facts. First, the heap-based calling convention guarantees the encapsulation of $(\text{RWX}, b, b+1, b)$.

Second, sharing $(\text{RO}, b, b+1, b)$ with unknown code does not threaten the integrity of b, since RO capabilities cannot be used to write to memory. These two facts are key when proving the following specification:

Lemma 7 (Full execution specification (26)).

$$\boxed{\text{mallocInv}(b_m, e_m)}, \boxed{\text{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\text{RWX}, \mathsf{data}, \mathsf{end}, \mathsf{code}); \begin{array}{l} \mathsf{r}_1 \Mapsto w_{adv} * \mathcal{V}(w_{adv}) * \\ \mathop{\Large *}_{(r,v) \in reg, r \notin \{\mathsf{pc}, \mathsf{r}_1\}} r \Mapsto w * \\ [\mathsf{code}, \mathsf{end}) \mapsto \mathtt{code\_instrs} * \\ \mathsf{data} \mapsto (\text{RO}, \mathsf{table}, \mathsf{table}+2, \mathsf{table}) * \\ [\mathsf{table}, \mathsf{table}+2) \mapsto \begin{array}{l} [(\text{E}, b_m, e_m, b_m); \\ (\text{E}, b_a, e_a, b_a)] \end{array} \end{array} \right\} \rightsquigarrow \bullet$$

Proof. We begin by applying program logic rules until we make it to the call to unknown code. At that point, a (fresh) region has been dynamically allocated and initialized to 1, and thus we have the following Separation Logic resources:

$$r_2 \Mapsto (\text{RWX}, b, b+1, b) * b \mapsto 1$$

At the call site, the calling convention creates an activation record, and sets up a sentry capability as the return pointer in r_0. (The "..." on the second line below stands for the address of the continuation after the call.)

$$r_0 \Mapsto (\text{E}, act, act_{end}, act) * \tag{2.11}$$

$$[act, act_{end}) \mapsto \mathtt{act\_instrs} +\!\!+ [(\text{RWX}, l, l+1, l); (\text{RWX}, \mathsf{code}, \mathsf{end}, \ldots)] *$$

$$l \mapsto (\text{RWX}, b, b+1, b) *$$

$$\mathsf{r}_2 \Mapsto 0 *$$

$$\mathsf{r}_3 \Mapsto (\text{RO}, b, b+1, b) \tag{2.12}$$

Note in particular how the RWX capability pointing to b (part of the "local state") is only reachable from the capability (pointing to l) stored in the activation record, while the RO copy is available in register r_3.

The `call` macro then passes control to the adversary by jumping to w_{adv}. To reason about this jump, we apply Corollary 2 (assuming w_{adv} is safe). This requires us to show that all parameters in the current register state are valid. In particular, we must show that the sentry capability set up by the calling convention (2.11) is safe to execute, and that the read-only capability (2.12) is safe to share.

The latter is done by allocating an appropriate invariant, which is allowed to be *stronger* than the value relation itself, since the capability in question is read-only. To this end, we will allocate an invariant that remembers the current integer pointed to by b, namely 1.

$$\boxed{\exists w, b \mapsto w * w = 1}$$

That same invariant is then used to prove that (2.11) is safe to execute, in particular to show that the assert statement succeeds, and hence does not change the assert flag. □

From this functional specification, we can instantiate our updated adequacy theorem (Theorem 5) to then derive the following end-to-end theorem about our program.

Theorem 6 (End-to-end theorem: the read-only permission guarantees integrity (27)). *Starting from an initial state of the machine* (reg, mem) *assuming that:*

- $prog \uplus adv \uplus malloc \uplus assert \subseteq mem$, *where:*
  $adv : [b_{adv}, e_{adv}) \to \text{Word}$, $prog : [\mathsf{code}, \mathsf{end}) \to \text{Word}$
  $malloc : [b_m, e_m) \to \text{Word}$ *and* $assert : [b_a, e_a) \to \text{Word}$*;*

- *the contents of prog correspond to the encoded instructions and program data (i.e. table with capabilities to the malloc and assert subroutines);*

- *the adversary memory contains no capabilities except a table with a capability to the malloc subroutine;*

- *malloc contains the implementation of the* `malloc` *subroutine;*

- *assert contains the implementation of the* `assert` *subroutine, with its flag at address* a_{flag}*, initialized to 0;*

- *the initial state of registers satisfies:*
  $reg(\mathsf{pc}) = (\text{RX}, \mathsf{code}, \mathsf{end}, \mathsf{code})$,
  $reg(\mathsf{r}_1) = (\text{RWX}, b_{adv}, e_{adv}, b_{adv})$.

Then, for any reg', mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $mem'(a_{flag}) = 0$.

Proof. We apply the updated adequacy theorem (Theorem 5), using the specification proved in Lemma 7. All that remains is to prove the validity of the adversary capability: $\mathcal{V}(\text{RWX}, b_{adv}, e_{adv}, b_{adv})$. This is done in two steps. First, the adversary linking table is proved valid by applying validity of the `malloc` subroutine (Lemma 6). Next, the rest of the adversary region is proved valid through the assumption that it does not contain any other capabilities. The full proof can be found in the Coq mechanisation. □

2.8 Case study: a Library Implementing Dynamic Sealing and a Client

We have presented so far a variety of smaller examples enforcing interesting encapsulation properties while interacting with adversarial code. In this section, we demonstrate that our approach scales up to the verification of a larger case study, involving not only the building blocks of Section 2.7, but using them to build and modularly verify a number of libraries built on top of each other.

We take inspiration from the literature on *object capability patterns* (OCPs) from high-level languages, a technique that enables programmers to protect the private state of their objects from corruption by untrusted code. More precisely, we consider the *dynamic sealing* OCP as presented by [138]. Dynamic sealing enforces a form of data abstraction in the absence of static types. It can be implemented as a library providing pairs of seal/unseal functions, allowing their clients to "seal" private data into opaque objects which can be safely shared with untrusted code, and later unsealed in order to get back the original data.

In the context of a high-level language, [138] present a formally verified implementation of dynamic sealing, equipped with a specification that captures the abstraction guarantees it provides. The authors then use this dynamic sealing library to build and verify a library of abstract integer intervals, where the integrity of an interval value (representing a range $[i, j)$ with $i \leq j$) is protected using dynamic sealing. Finally, the authors use their verified integer library to establish *robust safety* of a simple client program checking integrity of intervals, establishing that an untrusted context cannot violate the internal invariants of the program and its underlying libraries.

We implement and verify low-level variants of the dynamic sealing OCP, interval library, and their robustly safe client. This represents a non-trivial amount of code: our implementation of those three components adds up to 632 machine instructions. Nevertheless, despite the fact that those libraries are implemented in low-level assembly code, we are able to give them specifications at a level of abstraction similar to their high-level counterparts.

For ease of reading, we will keep the explanations fairly high-level. We will first show high-level pseudo-code for the implementation of the interval library and its client, and informally discuss what kind of properties should be enforced. Then, we will present the key ideas for implementing dynamic sealing on a capability machine, and then for reasoning about it, in particular how to instantiate its specification to be able to verify the interval library.

2.8.1 Interval Library and Client

The interval library implements an abstract data type representing intervals. An interval has a lower and upper bound, which can be extracted via two functions; `imin` and `imax`. An interval is created via a function `makeint` that takes as input two integers, and chooses the smallest input as the lower bound, and the largest input as the upper bound. Crucially, the internal representation of an interval must stay hidden so as to guarantee its integrity.

We thus use *dynamic sealing* ([136]) to dynamically enforce data abstraction for the intervals representation. We detail our implementation of seals in Section 2.8.2. For now, it suffices to know that a seal is a pair of functions, `seal` and `unseal`, where the former hides the internal representation of some value, such that only the latter can expose it.

An interval can be represented as an ordered pair of integers. On the capability machine, we implement such a pair as a dynamically allocated region of size two, storing the lower and upper bound of the interval. Then, an interval itself consists of a capability with read/write authority over the corresponding region of size two. In Figure 2.15, we depict the high-level implementation of our interval library. Note that the library implements closures around a fresh `seal-unseal` pair, used to seal the

```
interval㉘ = $\lambda\_,$  let (seal, unseal) = makeseal() in
              let makeint = $\lambda\ z_1\ z_2,$  let $x$ = malloc(2) in
                                        $x \leftarrow \{min(z_1, z_2); max(z_1, z_2)\};$
                                        seal($x$)
              in
              let imin = $\lambda\ i,$ unseal($i$)[0] in
              let imax = $\lambda\ i,$ unseal($i$)[1] in
              (makeint, imin, imax)

client㉙ =     let (makeint, imin, imax) = interval() in
              let checkint = $\lambda\ i,$ assert(imin($i$) $\leq$ imax($i$)) in
              (checkint, makeint, imin, imax)
```

Figure 2.15: High-level pseudo-code for the implementation of the interval library and its client.

aforementioned internal representation of intervals. The low-level implementation that we formally reason about can be thought of as the result of compiling the high-level implementation shown in Figure 2.15.

The same figure also depicts a client of the interval library. The client exposes four entry points to the environment: in addition to entries to makeint, imin and imax from a fresh instance of the interval library, the client also exposes an encapsulated checkint function that, given an interval, dynamically asserts that the expected representation invariant holds for the interval, that is, that the minimum of the interval is indeed smaller than or equal to the maximum of the interval.

When formally verifying the interval library and its client, we will need an invariant to keep track of each interval created by makeint. The invariant should capture the properties enforced by the implementation of the interval library. We can already list the internal properties of an interval intuitively. First and foremost, the lower bound of an interval must be less than or equal to its upper bound. A perhaps more subtle property is that intervals are immutable. Thus, we will need to define an invariant that represents each interval as a dynamically allocated region of size two, which stores the lower and upper bound, and is immutable. The seal-unseal pair encapsulated by the library will be used only to seal intervals that adhere to this representation (satisfy this invariant). Keeping this intuition in mind, let us now explore the technical implementation of seals.

2.8.2 Dynamic Sealing

Dynamic sealing makes it possible to support data abstraction dynamically. The function makeseal creates a pair of functions, seal and unseal, where seal is used to seal a word w into a fresh sealed word σ. We will also refer to σ as the key to w.

SEAL SPEC ㉚

$$\left\{ (-,b_s,e_s,-);\ \begin{array}{l} [b_s,e_s) \mapsto \texttt{seal} * \\ \text{sealInv}\ ds\ \Phi * \\ \mathsf{r}_1 \Mapsto v * \Phi(v) * \cdots \end{array} \right\} \rightsquigarrow \left\{ s\ k.\ \begin{array}{l} \lceil s = \mathsf{Running} \rceil * \\ \quad \text{isSealedWord}\ k\ v * \\ \quad \mathsf{r}_1 \Mapsto\ k * \cdots \\ \lor \lceil s = \mathsf{Failed} \rceil \end{array} \right\}$$

UNSEAL SPEC ㉛

$$\left\{ (-,b_u,e_u,-);\ \begin{array}{l} [b_u,e_u) \mapsto \texttt{unseal} * \\ \text{sealInv}\ ds\ \Phi * \\ \mathsf{r}_1 \Mapsto k * \cdots \end{array} \right\} \rightsquigarrow \left\{ s\ v.\ \begin{array}{l} \lceil s = \mathsf{Running} \rceil * \\ \quad \text{isSealedWord}\ k\ v * \\ \quad \mathsf{r}_1 \Mapsto\ v * \Phi(v) * \cdots \\ \lor \lceil s = \mathsf{Failed} \rceil \end{array} \right\}$$

Figure 2.16: Specifications of `seal` and `unseal`

The only way to extract the word w from σ is with `unseal`. The key point is that this `seal`-`unseal` pair supports data abstraction by *sealing away* or *hiding* the internal representation of some value, only known and available to the owner of the associated `unseal` function.

Although capability machines such as CHERI include seals as a language primitive, we show here how we can implement seals in software, as a low-level library. The library is implemented via a data structure that stores each word sealed through `seal`, associating each sealed word with a key. A key in itself does not reveal any details about the word it is hiding. However, it can provide access to that word, granted one has the proper authority to unseal it. Only a valid key should grant access to a sealed word. Keys, and the data structure that uses them, should intuitively satisfy two properties; (1) the unforgeable nature of keys and (2) the unique association between a key and the word it seals.

The `seal` and `unseal` subroutines respectively perform insertions and lookups in this underlying data structure. `seal` takes a word as input, generates a fresh key, and adds the key value association to the data structure. `unseal` takes a key as input, checks that the key is associated to a value in the data structure, in which case it returns the value.

2.8.2.1 Reasoning about dynamic sealing

A shared `seal`-`unseal` pair can be used to seal any word. In practice, one typically encapsulates a `seal`-`unseal` pair within a library, performing additional checks and thus ensuring that words that are sealed always satisfy a specific property. Then, whenever one successfully unseals a given key, one gets that the corresponding word satisfies the chosen property. For instance, the interval library enforces that each sealed word is a region of size 2, storing the ordered bounds of an interval.

When reasoning about code invoking the dynamic sealing library, one will need to pick, for each `seal-unseal` pair, an *representation invariant* $\Phi : \text{Word} \rightarrow iProp$ describing the values to be sealed/unsealed by the pair[7]. Then, each `seal-unseal` pair maintains an Iris invariant sealInv describing the state of the pair itself, namely the data structure storing the key-values for all sealed entries. Additionally, this invariant stores the information that each sealed value satisfies Φ.

$$\text{sealInv}\ ds\ \Phi\ ㉜ \quad \triangleq \quad \boxed{\begin{array}{l}\exists wvals, \text{dataStructure}\ ds\ wvals \\ * \mathop{\Large\ast}_{(-,w)\in wvals}\ \Phi(w)\end{array}}$$

We require that Φ is persistent, since the representation invariant of a sealed word should always hold once sealed. The dataStructure predicate describes the state of the data structure internal to the seal library (see Section 2.8.2.2 for a formal definition). It asserts that *ds* can be used to access a data structure storing the key value pairs denoted by *wvals* (a sequence of pairs in Addr × Word). In other words, *wvals* is the complete list of all words that have been sealed so far, each paired with their associated key.

A sealed word is sealed forever. It is thus possible to persistently remember that a particular word is an element of *wvals*. The predicate isSealedWord k v states that the key k is uniquely associated with the sealed word v. We present the formal definition of isSealedWord in Section 2.8.2.2.

The functional specifications of the `seal` and `unseal` subroutines depend on an instance of the seal invariant sealInv, for a specific user-provided predicate Φ. Then, `seal` can only be applied to words for which the representation predicate Φ holds. `unseal` can fail if a given key is not valid, or if it is not associated with any sealed word, however if it succeeds, it will return a word for which Φ holds. The specification of `makeseal` allocates a fresh sealInv instance, for any Φ chosen by the client of the library. Figure 2.16 shows specifications for `seal` and `unseal` (where we omits low-level administrative details).

2.8.2.2 Implementing a low level seal library

We now present the data structure used to implement the low-level seal library. We implement it as a linked associative list with a twist, next refered to as a *linked list dictionary*. The trick is to take advantage of the unforgeable nature of capabilities, and use the capability to (a subrange of) a list node as a key to that node; the corresponding value being then stored in the node.

Figure 2.17 shows the in-memory representation of a linked list dictionary storing three key-value pairs. Each node is implemented as a region of size three, where the bottom address acts as the key address. To avoid access to sealed values, it is important that a key does not provide authority over the other parts of a node (the value and the next pointer). For instance, the value v_1 is uniquely associated to the capability $(\text{RWX}, b_1, b_1+1, -)$.

The linked list dictionary library contains two subroutines, `findB` ㉝ and `append` ㉞. `findB` expects as input an integer b, searches the linked list for a node

[7] An analogous representation invariant is used in the [138]

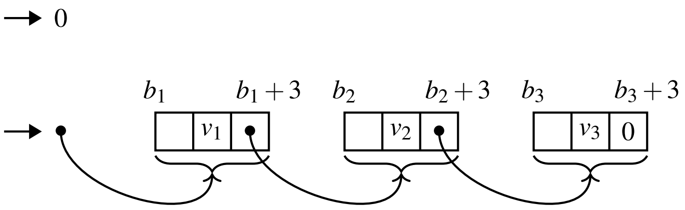


Figure 2.17: In-memory representation of an empty dictionary linked list and a dictionary linked list with three values v_1, v_2 and v_3.

of the form $(\text{RWX}, b, b+3, -)$ and returns the value that the associated node stores. It fails if no such node exists. `append` expects a word as input, invokes `malloc` to dynamically allocate a new node of size three, stores the input word in the second position of that node, and then stores that node as the new tail of the linked list. Finally a key can then be derived from the newly created node; we now explain in more detail how that is done.

A fresh instance of a `seal`-`unseal` pair is created by calling the `makeseal` subroutine, which returns a pair of closures encapsulating a new empty linked list dictionary. Sealing a word w adds it to the dictionary, and returns a *restricted* capability representing the key to the linked list dictionary entry. Say for instance that the input word w is appended to the list in a fresh node $(\text{RWX}, b, b+3, b)$. The `seal` subroutine will then return the key $(\text{RWX}, b, b+1, -)$ (the address pointed to does not matter, and is here omitted for clarity).

Recall that in the enclosed linked list dictionary, w will be stored at address $b+\mathit{1}$, for which the returned sealed value, or key, does not have authority. This sealed value is *unforgeable*. The only way to create it would be to derive it from a capability $(\text{RWX}, b', e', \_)$ where $[b, b+1) \subseteq [b', e')$. However, this is impossible since the appended node is freshly allocated using a safe `malloc` subroutine, which is guaranteed to hand out fresh regions upon invocation. Only `seal` has access to such a capability, and thus sealed values cannot be forged.

In turn, the `unseal` subroutine expects a RWX capability of range 1 as input. It reads its lower bound, searches the enclosed linked list for a node with matching lower bound, and returns the associated word. Let us consider a continuation of the previous example. Say that `unseal` receives $(\text{RWX}, b, b+1, -)$ as input. It begins by authenticating the key by dynamically verifying its permission to be RWX, and its size to be 1. Upon validating its permission and range, it then runs `findB` on the enclosed linked list dictionary with the integer b, and returns the word stored within the node $(\text{RWX}, b, b+3, -)$ at address $b+1$, namely the previously sealed word w. The authentication guarantees that a key has the same unforgeable authority as when it was created.

In summary, the `seal` and `unseal` subroutines are implemented as follows:

- `seal`:

1. append the input to the enclosed linked list dictionary
2. restrict the range of the fresh node capability to bottom address of node
3. return resulting restricted capability

- `unseal`:

1. check that permission of input is RWX
2. check that the range of input is 1
3. get the lower bound of input
4. find the node in the linked list dictionary with same lower bound
5. return the stored word at that node (fail if no such node exists)

We now have enough ingredients to revisit the predicates used in the previous section to define the seal invariant. Recall that the dataStructure predicate represents the state of the data structure internal to the seal library (now defined to be a linked list dictionary), and that the isSealedWord predicate describes a persistently known association between a sealed word and its key.

$$\begin{array}{rcl}
\text{dataStructure}\ \mathit{ds}\ \mathit{wvals} & \triangleq & \exists \mathit{hd}, \mathit{ds} \mapsto \mathit{hd} \\
 & & * \ \text{isList}\ \mathit{hd}\ \mathit{wvals} \\
 & & * \ \text{Exact}\ \mathit{wvals} \\
\text{isSealedWord}\ k\ v & \triangleq & \exists \mathit{wvals}, \text{Pref}\ \mathit{wvals} * \lceil (k,v) \in \mathit{wvals} \rceil^{8} \\
 & & * \mathcal{V}(\textsc{rwx}, k, k+1, -)
\end{array}$$

The head of the linked list dictionary is stored in location *ds*. isList corresponds to a standard inductive separation logic predicate for linked lists. Since the list monotonically grows, it is useful to persistently remember any prefix of the linked list dictionary. Exact *wvals* (the authoritative view of the list state) roughly states that *wvals* is the full state of the data structure. Pref *wvals* (the local fragment view) states that *wvals* is a prefix of the data structure. isSealedWord k v, a persistent predicate, states that the word v has been sealed with a key; a capability with with lower bound k. This key is safe to share, hence $\mathcal{V}(\textsc{rwx}, k, k+1, -)$ holds.

In the next section, we describe how we use the reasoning principles about `seal-unseal` to verify our interval library.

2.8.3 Verifying the Interval Library and its Client

The first key step is to formally define the representation invariant for an interval. Recall the intuitive description given in Section 2.8.1: an interval is a capability with authority over a region of size 2, storing the lower and upper bounds of an interval, and which is immutable.

[8] In the Coq mechanization, *wvals* associates the word w to $k+1$ rather than k, for technical reasons. This small discrepancy has otherwise no impact on the rest of the proof.

A first thought might be that one can define the representation invariant using two points-to predicates for the region. However, this does not capture the immutability of intervals, nor is it persistent. Instead, we use *persistent* points-to predicates ([150]). A persistent points-to predicate $a \hookrightarrow w$ asserts that address a stores the word w. It can be used to read from address a, but not write to it, and as such, is a persistent resource. This is exactly what we need for our immutable invariants. We formally define the representation invariant isInterval as follows:

$$\begin{array}{rcl} \text{isIntervalInt } z_1\ z_2\ w \text{ (35)} & \triangleq & \exists a, \lceil w = (\text{RWX}, a, a+2, a) \rceil * a \hookrightarrow z_1 * \\ & & (a+1) \hookrightarrow z_2 * \lceil z_1 \leq z_2 \rceil \\ \text{isInterval (36)} & \triangleq & \lambda w, \exists z_1\ z_2, \text{isIntervalInt } z_1\ z_2\ w \end{array}$$

(Note, in particular, that the invariant also captures the property that the lower bound is less than or equal to the upper bound.) Using properties of persistent points-to predicates, we can prove the following lemma:

Lemma 8 ((37))**.** isIntervalInt $z_1\ z_2\ w \twoheadrightarrow$ isIntervalInt $z_3\ z_4\ w \twoheadrightarrow \lceil z_1 = z_3 \wedge z_2 = z_4 \rceil$.

Because isInterval is persistent, we can use it as the representation predicate for a `seal`-`unseal` pair, which will thus operate over the following invariant:

$$\text{sealInv } ll \text{ isInterval}$$

This seal invariant is allocated by the specification for `makeseal`, which is invoked during the creation of an interval library closure.

When sealing a new interval using `makeint`, we must establish isInterval for the newly created interval. This requires us to transform the regular points-to predicates handed out by the `malloc` specification into persistent points-to predicates, and assert that indeed $min(z_1, z_2) \leq max(z_1, z_2)$.

Specifications for `imin` and `imax` return the respective lower and upper bound of a sealed interval. The seal invariant guarantees that the sealed word is an interval according to the representation invariant isInterval. In other words, if `imin` or `imax` succeeds for some word w, we know that w is the key to some associated capability pointing to the bounds of an interval $[l, r]$; specifically that isIntervalInt $l\ r\ w$ holds.

During the verification of `checkint`, the specification for `imin` gives us some value l and predicate isIntervalInt $l\ r\ w$. Similarly, the specification for `imax` gives us some value r' and predicate isIntervalInt $l'\ r'\ w$. Notice that the bounds may be different, but the sealed word w is the same in each instance. We can thus apply Lemma 8 on the two given instances of isIntervalInt, and use the definition of isInterval to conclude that the given `assert` statement succeeds, namely that $l \leq r$.

Finally, all that remains is to apply adequacy and prove the following final end-to-end theorem:

Theorem 7 (End-to-end theorem: the interval client does not trigger an assertion failure (38))**.** *Starting from an initial state of the machine (reg, mem) in which regions reserved for the interval library, the seal library,* `malloc`*, the assert flag, the client and the adversary are all disjoint, and initialized as expected, we have that, for any reg', mem', if $(reg, mem) \longrightarrow^* (reg', mem')$ then $mem'(a_{flag}) = 0$.*

2.9 Case study: Data Abstraction

Object capability patterns enable programmers to protect their private state against corruption from untrusted code. Thus far, we have defined a methodology to reason about the integrity properties granted by object capability patterns. However, capabilities can be used to go beyond integrity guarantees. For instance, sentry (E) capabilities act as closures around some code and data, hiding the internal representation of a component, and thus enable applications of *data abstraction*. Seals, both the CHERI language primitive, and the software implementation presented in Section 2.8, can similarly be used to enforce data abstraction [92, 118, 136]. Furthermore, seals can be interpreted as cryptographic primitives that enforce *data confidentiality* [100]. Data abstraction is the process of hiding the internal implementation of an object from the context, whereas data confidentiality is the process of hiding data from the context, guaranteeing the exclusive access of some secret value.

While there is a subtle distinction between the two properties, both data abstraction and data confidentiality are binary properties that can be expressed through *contextual equivalence*. As such, if we want to reason about either, we need binary reasoning principles to capture relational properties between two capability machine programs. In this section, we present a binary model capturing contextual refinement between programs, and use it in a simple illustrative example of data abstraction. The model is adapted from a binary model to reason about confidentiality properties of a capability machine with special capabilities for a call-stack, which we will present in Chapter 4.

The main goal is to define contextual equivalence for capability machine programs. Such a definition involves a notion of capability machine components, linking and contexts; however, for simplicity, we here omit any detailed description, and assume that each of these notions are in place. *comp* denotes a compartment, C denotes a context, and $C[comp]$ denotes the closed program resulting from linking C and *comp* (we refer to Chapter 4 for formal definitions).

Contextual equivalence can then be defined as follows:

$$comp_1 \approx_{ctx} comp_2 \quad \triangleq \quad \forall C, C[comp_1] \rightarrow^* \mathsf{Halted} \iff C[comp_2] \rightarrow^* \mathsf{Halted}$$

While $comp_1$ and $comp_2$ are known components, the universally quantified context C is arbitrary, and as a result, showing contextual equivalence involves reasoning about arbitrary code. More precisely, we will need to relate the behavior of an open capability machine program to itself. We do this via a general definition of logical refinement.

First, we apply the Cerise program logic to define a binary logical relation that formalizes program refinement. We use existing techniques for defining binary logical relations in Iris [54, 83, 85], but apply them here to a low-level capability machine language. Next, we prove a binary universal contract that holds for arbitrary code, namely that all programs logically refine themselves. Finally, we apply the universal contract for proving the contextual equivalence of a pair of counters, and discuss future applications of the model.

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc})\\ \mathrm{ValidSubseg}(p,b,e,z_1,z_2) \qquad \mathrm{decode}(n)=\texttt{subseg}\ r\ z_1\ z_2 \qquad \mathcal{N}.\mathsf{spec}\!\uparrow\ \subseteq \varepsilon\\ \mathrm{specInv} \ast \hookrightarrow_{\mathsf{s}} \mathsf{Running} \ast \mathsf{pc} \mapsto_{\mathsf{s}} (p_{pc},b_{pc},e_{pc},a_{pc}) \ast a_{pc} \mapsto_{\mathsf{s}} n \ast r \Mapsto_{\mathsf{s}} (p,b,e,a)\end{array}}{\Rrightarrow_{\varepsilon} \hookrightarrow_{\mathsf{s}} \mathsf{Running} \ast \mathsf{pc} \mapsto_{\mathsf{s}} (p_{pc},b_{pc},e_{pc},a_{pc}+1) \ast a_{pc} \mapsto_{\mathsf{s}} n \ast r \Mapsto_{\mathsf{s}} (p,z_1,z_2,a)}$$

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc})\\ \mathrm{ValidLoad}(p,b,e,a) \qquad \mathrm{decode}(n)=\texttt{load}\ dst\ src \qquad \mathcal{N}.\mathsf{spec}\!\uparrow\ \subseteq \varepsilon\\ \mathrm{specInv} \ast \hookrightarrow_{\mathsf{s}} \mathsf{Running} \ast \mathsf{pc} \mapsto_{\mathsf{s}} (p_{pc},b_{pc},e_{pc},a_{pc}) \ast a_{pc} \mapsto_{\mathsf{s}} n \ast\\ dst \Mapsto_{\mathsf{s}} - \ast src \Mapsto_{\mathsf{s}} (p,b,e,a) \ast a \mapsto_{\mathsf{s}} w\end{array}}{\begin{array}{c}\Rrightarrow_{\varepsilon} \hookrightarrow_{\mathsf{s}} \mathsf{Running} \ast \mathsf{pc} \mapsto_{\mathsf{s}} (p_{pc},b_{pc},e_{pc},a_{pc}+1) \ast a_{pc} \mapsto_{\mathsf{s}} n \ast\\ dst \Mapsto_{\mathsf{s}} w \ast src \Mapsto_{\mathsf{s}} (p,b,e,a) \ast a \mapsto_{\mathsf{s}} w\end{array}}$$

Figure 2.18: Specification program rules

Our goal is to define a binary logical relation that captures logical refinement. We will say that a program pointed to by pc_1 refines a program pointed to by $\mathbf{pc_2}$, if $\mathbf{pc_2}$ produces the same effects as pc_1. We denote logical refinement as follows:

$$pc_1 \leq_{log} \mathbf{pc_2}.$$

Where pc_1 refers to the implementation, and $\mathbf{pc_2}$ to the specification (throughout the section, we will use **this typesetting** when referring to the specification program). As mentioned above, we define the binary logical relation on top of the cerise program logic. However, the program logic describes the behavior of a single capability machine program, while we need to compare the behavior of two programs. The solution is to apply an established technique, in which ghost state is used to describe the behavior of the specification program.

2.9.1 Ghost State for the Specification Program

The specification program state is tracked via three new resources; $\mathbf{r} \Mapsto_{\mathsf{s}} \mathbf{w}$ describes the state of register r, $\mathbf{a} \mapsto_{\mathsf{s}} \mathbf{w}$ describes the state of address a, and $\hookrightarrow_{\mathsf{s}} \mu$ describes the current specification execution state. The full (or authoritative) specification state is kept in an invariant, henceforth referred to as specInv. Similar to weakest precondition steps, the specification state is then updated via proof rules for each instruction.

For each weakest precondition rule (recall the fragment presented in Figure 2.9), there is an equivalent specification rule to carry out the corresponding step in the specification program. Figure 2.18 (39) shows two such rules, one for `subseg` and one for `load`.

The rules mostly resemble their program logic counterpart, but with three notable differences. First, they require specInv as an additional assumption. Second, since there are no weakest preconditions, the execution state is instead represented by a specification execution state fragment. Third, the result is guarded by a so-called

$$
\begin{array}{rcll}
\gamma_{spec} & \in & \mathsf{GName} & \\
\mu & \in & \mathrm{exprR} & \triangleq \ \mathrm{Ex}(\mathrm{ExecState}) \\
\mathbf{reg} & \in & \mathrm{regR} & \triangleq \ \mathrm{RegName} \hookrightarrow \textsc{Frac} \times \mathrm{AG}(\mathrm{Word}) \\
\mathbf{mem} & \in & \mathrm{memR} & \triangleq \ \mathrm{Addr} \hookrightarrow \textsc{Frac} \times \mathrm{AG}(\mathrm{Word}) \\
\mathbf{s} & \in & \mathrm{confR} & \triangleq \ \textsc{Auth}(\textsc{Option}(\mathrm{exprR}) \times \mathrm{regR} \times \mathrm{memR})
\end{array}
$$

$$
\begin{array}{rcl}
\hookrightarrow_{\mathsf{s}} \mu & \triangleq & \boxed{\circ\,(\textsc{Some}(\mathrm{Ex}(\mu)), \emptyset, \emptyset)}^{\gamma_{spec}} \\
\mathbf{r} \Mapsto_{\mathsf{s}} \mathbf{w} & \triangleq & \boxed{\circ\,(\textsc{None}, \{[\mathbf{r} := \mathbf{w}]\}, \emptyset)}^{\gamma_{spec}} \\
\mathbf{a} \mapsto_{\mathsf{s}} \mathbf{w} & \triangleq & \boxed{\circ\,(\textsc{None}, \emptyset, \{[\mathbf{a} := \mathbf{w}]\})}^{\gamma_{spec}}
\end{array}
$$

Figure 2.19: Ghost state describing the state of the specification program

$$
\mathrm{specInv} \triangleq \exists s, \varphi,
\boxed{
\begin{array}{ll}
\exists s', reg, mem, & \bullet(\boxed{\textsc{Some}(\mathrm{Ex}(s'))}^{\gamma_{spec}}, \mathrm{toSpecMap}(reg), \mathrm{toSpecMap}(mem)) \\
 & * \ (s, \varphi) \to^{*} (s', (reg, mem))
\end{array}
}^{\mathcal{N}.\mathsf{spec}}
$$

Where toSpecMap *maps every element a in the range of a partial map to* $(1, \mathrm{AG}(a))$

Figure 2.20: Specification invariant

frame preserving update modality $\Rrightarrow_{\varepsilon}$, necessary for updating the ghost state in the specification invariant. Each rule is proved by opening the invariant, asserting properties over the specification state, carrying out the relevant changes, and proving that these correspond to a step in the operational semantics, upon which the invariant can be closed again. These rules are then used to carry out the steps taken by the specification program.

The following paragraph details how the ghost state responsible for tracking the specification program is formally defined, and assumes the reader is familiar with Iris ghost state constructions. It can safely be skipped.

Formal Ghost State Definitions The ghost state needs to track the state of the specification execution state (Running, Halted or Failed), the state of its registers, and the state of its memory, and should enable local reasoning about state fragments. Figure 2.19 (40) defines the resource algebra describing each of these; the execution state is exclusive, while the register and memory state match the resource algebras used in the state interpretation of the program logic.

The full configuration is tracked via the authoritative resource algebra, giving rise to the aforementioned fragments: $\hookrightarrow_{\mathsf{s}} \mu$ asserts the current specification execution state to be μ, while $\mathbf{r} \Mapsto_{\mathsf{s}} \mathbf{w}$ and $\mathbf{a} \mapsto_{\mathsf{s}} \mathbf{w}$ are the specification register and memory points to predicates.

In the cerise program logic, the register and memory states are tracked by the state interpretation, whose authoritative view is part of the weakest precondition definition. For the specification program, we instead keep the authoritative view of the configuration state in an Iris invariant, defined in Figure 2.20 ㊶. The invariant additionally asserts, that the specification configuration is the result of applying the capability machine operational semantics, starting from some fixed configuration (s, φ). Most often, this starting configuration will be existentially quantified, but note that it is a free variable (and thus fixed) within the invariant itself.

2.9.2 A Binary Logical Relation for Contextual Refinement

Our formal definition captures a refinement relation between two capability machine programs. Intuitively, the refined program must *observably* behave "the same" as the implemented program. In other words, starting from refined register states, the two machines must exhibit the same observable effects on memory, and if the implementation halts, then so must the specification (note that we do not consider failing to be an observable effect). To easily specify related executions, we add the following type of program specification to our roster of weakest precondition notations:

$$\begin{aligned}\{w_1 \leq \mathbf{w_2}; P\} \rightsquigarrow \bullet \quad \triangleq \quad & \{\mathsf{pc} \Mapsto w_1 * \mathsf{pc} \Mapsto_{\mathsf{s}} \mathbf{w_2} * \hookrightarrow_{\mathsf{s}} \mathsf{Running} * P\} \rightsquigarrow \\ & \{s.\, \lceil s = \mathsf{Halted} \rceil \twoheadrightarrow\!\!* \hookrightarrow_{\mathsf{s}} \mathsf{Halted}\}\end{aligned}$$

The above specification holds if, starting from an implementation at program counter w_1, and a specification at program counter w_2, as well as a machine state satisfying P, the implementation machine either loops, or runs into completion, and if it halts, then the specification machine must also halt. Recall that all logical invariants are preserved at every step of execution, which may include invariants relating fragments of the implementation state to fragments of the specification state.

We define two binary logical relations, a binary expression relation for program refinement, and a binary value relation for word refinement (Figure 2.21 ㊷). Each resemble their unary counterpart; the expression relation is defined as a weakest precondition, while the value relation imposes memory invariants over the range of authority of related capabilities.

Recall that in a capability machine, all the components of a capability are observable. As such, the value relation *must* imply syntactic equivalence of its arguments, to prevent a context from exhibiting different behavior based on the syntactic differences of a word. Additionally, if a word grants read or write authority over memory, the value relation imposes memory invariants for each address in its range. These invariants contain the relevant points-to predicate for the implementation $a \mapsto w$, the matching points-to predicate for the specification $\mathbf{a} \mapsto_{\mathsf{s}} \mathbf{w}'$, and relates w to $\mathbf{w}'$.

Thus far, the relation seems somewhat trivial, as it appears to simply impose the syntactic equivalence of related words and related memory regions. After all, this is not surprising, when we consider how much is actually observable on low level machines. However, capability machines do grant some mechanisms to hide internal representations, namely through seals (on CHERI) and sentry capabilities. This is precisely captured by the value relation when relating E-capabilities; while the two capabilities must be syntactically equal, the memory regions they point to do not fall

$$\mathscr{V}(w_1,\mathbf{w_2}) \begin{cases} \mathscr{V}(z_1,\mathbf{z_2}) & \triangleq z_1 = \mathbf{z_2} \\ \mathscr{V}((\textsc{o},\cdots),\mathbf{w_2}) & \triangleq (\textsc{o},\cdots) = \mathbf{w_2} \\ \mathscr{V}((\textsc{e},b,e,a),\mathbf{w_2}) & \triangleq (\textsc{e},b,e,a) = \mathbf{w_2} \land \\ & \quad \triangleright \Box \mathscr{E}((\textsc{rx},b,e,a),(\textsc{rx},\mathbf{b},\mathbf{e},\mathbf{a})) \\ \mathscr{V}((\textsc{rw}/\textsc{rwx},b,e,-),\mathbf{w_2}) & \triangleq (\textsc{rw}/\textsc{rwx},b,e,-) = \mathbf{w_2} \land \\ & \quad \mathop{\ast}_{a\in[b,e)} \boxed{\begin{array}{c}\exists w,\mathbf{w}',\ a \mapsto w \ast \mathbf{a} \mapsto_s \mathbf{w} \\ \ast\ \mathscr{V}(w,\mathbf{w}')\end{array}} \\ \mathscr{V}((\textsc{ro}/\textsc{rx},b,e,-),\mathbf{w_2}) & \triangleq (\textsc{ro}/\textsc{rx},b,e,-) = \mathbf{w_2} \land \\ & \quad \mathop{\ast}_{a\in[b,e)} \\ & \qquad \exists P, \boxed{\begin{array}{c}\exists w,\mathbf{w}',\ a \mapsto w \ast \mathbf{a} \mapsto_s \mathbf{w}' \\ \ast\ P(w,\mathbf{w}')\end{array}} \\ & \qquad \ast \triangleright \Box \left(\begin{array}{c}\forall w,\mathbf{w}', P(w,\mathbf{w}') \mathrel{-\!\!\ast} \\ \mathscr{V}(w,\mathbf{w}')\end{array}\right) \end{cases}$$

$$\mathscr{E}(w_1,\mathbf{w_2}) \quad \triangleq \forall reg_1, \mathbf{reg_2}, \left\{ w_1 \leq \mathbf{w_2}; \mathop{\ast}_{\substack{(r,v_1)\in reg_1,\\ (r,\mathbf{v_2})\in \mathbf{reg_2},\\ r\neq \mathsf{pc}}} \begin{array}{c} r \Mapsto v_1 \ast \mathbf{r} \Mapsto_s \mathbf{v_2} \\ \ast\ \mathscr{V}(v_1,\mathbf{v_2}) \end{array} \right\} \rightsquigarrow \bullet$$

Figure 2.21: Binary logical relation defining refinement between two words.

within any memory invariant, and may thus contain arbitrary and different words. Instead, two E-capabilities are related when they *behave* the same upon invocation, as captured by the expression relation. In turn, the expression relation is defined using the previously stated program specification for two related executions, starting from any refined pair of register states. The expression relation captures logical refinement.

$$pc_1 \leq_{log} \mathbf{pc_2} \quad \triangleq \quad \mathscr{E}(pc_1,\mathbf{pc_2})$$

In general, an arbitrary program ought to refine itself. The binary universal contract guarantees that all capability machine programs are related to themselves. More precisely, we state the following binary fundamental theorem of logical relations.

Theorem 8 (Binary FTLR (43)).

$$\text{specInv} \mathrel{-\!\!\ast} \mathscr{V}((p,b,e,a),(\mathbf{p}',\mathbf{b}',\mathbf{e}',\mathbf{a}')) \mathrel{-\!\!\ast} (p,b,e,a) \leq_{log} (\mathbf{p}',\mathbf{b}',\mathbf{e}',\mathbf{a}')$$

Recall that the value relation implies syntactic equivalence of the capability, and the region it points to. The conclusion of the theorem can thus be restated as $(p,b,e,a) \leq_{log} (\mathbf{p},\mathbf{b},\mathbf{e},\mathbf{a})$.

The proof proceeds much like the unary fundamental theorem, as a proof by cases over each possible instruction pointed to by a. The main difference, is that for each step taken in the implementation, we must apply the relevant specification rule to show that the same step can be taken in the specification program. In the final subsection below, we apply the binary fundamental theorem to prove the contextual equivalence of two simple counter implementations.

2.9.3 A Pair of Contextually Equivalent Counters

We showcase the model in a simple data abstraction example. In this example, we implement two counters; one that counts down, and one that counts up. Each data structure exports two methods, an increment function, and a read function. While the internal representation of the two counters differ, their observable behavior is the same: the increment function produces no observable behavior, while the read function of the decrementing counter returns the absolute value of its internal representation, which ought to equal the return value of the incrementing counter. The code of each counter is presented in Figure 2.22.

More subtly, while the state of the internal representation differs, the size it occupies is the same (a region of size one, allocated upon counter instantiation). As a result, no context can use memory related side effects, such as the address of a region allocated via malloc, to distinguish the two counters.

The two data structures are thus contextually equivalent. In this section, we outline how we can prove it. The proof can be split up into two phases. First, we show that the data structures fall within the binary logical relation. Next, we apply the fundamental theorem of logical relations and Iris adequacy to derive the final statement of contextual equivalence.

Before we begin the proof however, it's worth discussing some preliminary observations. First, while the example might be relatively trivial, it nicely illustrates how sentry capabilities can be used to implement data abstraction. The internal counter representation differs, and the subroutines contain different code, but no context can distinguish them.

However, the example also highlights the limits of low level data abstraction. The difficulty lies in the many observable side effects of low level machines, from the size of the modules themselves, to the size of dynamically allocated regions. Indeed, recall that related words must be *syntactically equal*. Entry points to a module must thus be composed of exactly the same fields. As a result, the two modules are assumed to be stored in the same memory region. This can be somewhat mitigated by implementing some clever indirection between module entry points and the modules themselves, but in this example we simplify the matter by padding the incrementing counter module such that its size matches that of the decrementing counter.

Furthermore, in the cases where malloc is a shared library, contexts can observe the next addressable address by allocating a new region, and reading its bounds. Given our simple implementation of malloc, the two modules must therefore carefully allocate regions of the same size, which in some cases can involve more padding.

Ultimately, the nature of low level machines inherently imposes undesirable restrictions on what can be considered contextually equivalent modules. However, we argue that this does not diminish the usefulness of our binary model. First, it is clear that the model still displays interesting abstraction properties granted by sentry capabilities. Second, it presents a general methodology for investigating other, more intentional, relational properties between capability machine programs. We leave such investigations to future work.

```
; initially, PC = (RX, init, end, init)
;            r0 = (unknown) return pointer
init:
  ...               ; dynamically allocate counter, initialized to 0.
   init_instrs      ; create sentry capabilities for closures around
  ...               ; read and write subroutines
  jmp r0
incr:
  load r1 renv
  add r1 r1 1       ; count up
  store renv r1
  mov renv 0
  mov r1 0
  jmp r0
read:
  load ra renv
  mov renv 0
  mov r1 0
  jmp r0
data:
  0xFFFF            ; padding such that the two modules equal in size
  (E, bm, em, bm) ; entry point to the malloc subroutine
end:

; initially, PC = (RX, init, end, init)
;            r0 = (unknown) return pointer
init:
  ...               ; dynamically allocate counter, initialized to 0.
   init_instrs      ; create sentry capabilities for closures around
  ...               ; read and write subroutines
  jmp r0
incr:
  load r1 renv
  sub r1 r1 1       ; count down
  store renv r1
  mov renv 0
  mov r1 0
  jmp r0
read:
  load ra renv
  sub ra 0 ra       ; calculate the positive value of the counter
  mov renv 0
  mov r1 0
  jmp r0
data:
  (E, bm, em, bm) ; entry point to the malloc subroutine
end:
```

Figure 2.22: Two counters

2.9.3.1 Proving that the Counters are Contextually Equivalent

As mentioned, the proof is split up into two phases. First, we show that the modules are logical refinements of each other. Since the two directions are entirely symmetric, we will focus on one direction only. The statement assumes four previously allocated invariants, one containing the implementation program, another containing the specification program, a binary variant of the malloc invariant, and the previously described specInv.

Lemma 9 ([44]).

$$\boxed{[\mathsf{init},\mathsf{end}) \mapsto \texttt{counter\_up\_instrs}}, \boxed{[\mathsf{init},\mathsf{end}) \mapsto_s \texttt{counter\_down\_instrs}},$$

$$\boxed{\mathrm{mallocBinaryInv}(b_m,e_m)}, \mathrm{specInv} \vdash (\mathrm{E},\mathsf{init},\mathsf{env},\mathsf{init}) \leq_{log} (\mathrm{E},\mathsf{init},\mathsf{env},\mathsf{init})$$

Proof. We begin by stepping through *init_instrs* in lock step. First, the local counter is allocated, initialized to zero:

$$r_1 \Mapsto (\mathrm{RWX},d,d+1,d) \ast \mathbf{r_1} \Mapsto_s (\mathrm{RWX},\mathbf{d},\mathbf{d}+\mathbf{1},\mathbf{d}) \tag{2.13}$$
$$\ast\, d \mapsto 0 \ast \mathbf{d} \mapsto_s \mathbf{0} \tag{2.14}$$

We highlight that the binary malloc invariant yields a binary variant of the malloc specification, which crucially assumes that the allocated regions are of equal size.

Next, closure entry points are created for the incr and read subroutines:

$$r_1 \Mapsto (\mathrm{E},b,e,b) \ast \mathbf{r_1} \Mapsto_s (\mathrm{E},\mathbf{b},\mathbf{e},\mathbf{b}) \tag{2.15}$$
$$\ast\, [b,e) \mapsto \mathsf{cls\_entry\_instrs} +\!\!+ [(\mathrm{RWX},\mathsf{init},\mathsf{end},\mathsf{incr});(\mathrm{RWX},d,d+1,d)] \tag{2.16}$$
$$\ast\, [\mathbf{b},\mathbf{e}) \mapsto_s \mathsf{cls\_entry\_instrs} +\!\!+ [(\mathrm{RWX},\mathsf{init},\mathsf{end},\mathsf{incr});(\mathrm{RWX},\mathbf{d},\mathbf{d}+\mathbf{1},\mathbf{d})] \tag{2.17}$$
$$\ast\, r_2 \Mapsto (\mathrm{E},b',e',b') \ast \mathbf{r_2} \Mapsto_s (\mathrm{E},\mathbf{b}',\mathbf{e}',\mathbf{b}') \tag{2.18}$$
$$\ast\, [b',e') \mapsto \mathsf{cls\_entry\_instrs} +\!\!+ [(\mathrm{RWX},\mathsf{init},\mathsf{end},\mathsf{read});(\mathrm{RWX},d,d+1,d)] \tag{2.19}$$
$$\ast\, [\mathbf{b}',\mathbf{e}') \mapsto_s \mathsf{cls\_entry\_instrs} +\!\!+ [(\mathrm{RWX},\mathsf{init},\mathsf{end},\mathsf{read});(\mathrm{RWX},\mathbf{d},\mathbf{d}+\mathbf{1},\mathbf{d})] \tag{2.20}$$

Finally, initialization ends by jumping to the unknown context. In order to reason about the unknown continuation, we apply the binary fundamental theorem of logical relations (Theorem 8), which leaves us with the following two proof obligations:

$$\mathcal{V}((\mathrm{E},b,e,b),(\mathrm{E},\mathbf{b},\mathbf{e},\mathbf{b})) \textit{ and } \mathcal{V}((\mathrm{E},b',e',b'),(\mathrm{E},\mathbf{b}',\mathbf{e}',\mathbf{b}')) \tag{2.21}$$

Each involve proving a weakest precondition, while taking steps in the specification program. The proof obligations are each guarded by an always modality. We must therefore first allocate invariants for all the resources that the execution depends on. Most interesting is the invariant for the local state d, which expresses the underlying relation between the two distinct internal counter representations. One counter counts up, while the other counts down, and thus one internal value must invariantly be the negation of the other.

$$\boxed{\exists z, d \mapsto z \ast d \mapsto_s -z}$$

With the invariants in place, we can prove the two value relations, by stepping through instructions in the implementation, as well as in the specification. Two instructions must be reasoned about in lock-step: the `store` instruction in incr, and the `load` instruction in read. Both require opening the above instruction, applying the relevant rules before closing it again. The `store` instruction is noteworthy, since the state of d changes to $z+1$ one the one hand, and $z-1$ on the other, thus preserving the necessary invariant. □

Once the above lemma has been established, we can derive the following end-to-end theorem. Note that the theorem depends on formal definitions of linking, contexts and components, which we forgo in this presentation (see Chapter 4 for formal definitions).

Theorem 9 (End-to-end theorem: the two counters are contextually equivalent (45)). *Starting from two initial states of the machine in which regions reserved for the counter library ($comp_{count-up}$ and $comp_{count-down}$ respectively),* `malloc` *and the reserved adversary region are all disjoint, and initialized as expected (linking between them is possible, and the adversary region is a context of instructions only, and has a start function capability), we have that,*

$$comp_{count-up} \approx_{ctx} comp_{count-down}$$

Proof. Proving contextual equivalence involves showing two contextual refinements, one in each direction. Each proof is symmetric, and we focus here on $\Rightarrow$. The proof starts by setting up the resources of the initial machine states, and allocating specInv at that starting state, taking care to set aside $\hookrightarrow_{\mathsf{s}}$ $\mathsf{Running}$.

Next, since the adversary context is made up of instructions only, it is possible to show that the entry point of the program (the start function capability) c is in the value relation $\mathcal{V}(c,c)$. By the fundamental theorem, we thus know that $\mathcal{E}(c,c)$.

We then instantiate $\mathcal{E}(c,c)$ to the starting register states, with the corresponding register resources and $\hookrightarrow_{\mathsf{s}}$ $\mathsf{Running}$. Next, we show that its contents are indeed refinements. In particular, this involves applying Lemma 9 for the counter modules, which are imported by the context.

Finally, we apply Iris adequacy on the resulting weakest precondition, from which we can assert that the implementation does not get stuck, and that allocated invariants hold at every step of execution (note that this includes specInv). If the implementation halts, we must show that the specification also halts. This follows from the postcondition of our weakest precondition.

If the implementation halts, the postcondition asserts $\hookrightarrow_{\mathsf{s}}$ Halted. We also know that specInv holds. We can thus derive that the specification execution state is currently Halted, and subsequently that $(\mathsf{Running}, \varphi) \rightarrow^{*} (\mathsf{Halted}, (reg, mem))$ for some reg and mem, where φ is the starting configuration, as established at the allocation of specInv. We conclude that the implementation contextually refines the specification. □

2.10 Discussion and Perspectives

In this paper we have introduced Cerise, a program logic for reasoning about a low-level capability machine. Moreoever, we have shown how Cerise can be used to define a logical relation for reasoning about unknown code. Thanks to the logical relation and the fundamental theorem from Section 2.5, Cerise can be used for *robust verification* [121, 138], i.e., to verify correctness of software that interacts with unverified components. The Cerise program logic is the culmination of ideas used in a sequence of earlier papers [55, 130, 131, 135] and this paper is intended to give an accessible and didactic introduction to Cerise and the application of Cerise to program verification in the presence of untrusted code, accompanied with new results on a heap-based calling convention and implementations of sophisticated object-capability patterns.

Throughout the paper we have introduced increasingly complex examples, which demonstrate how fine-grained abstractions can be implemented on a capability machine and reasoned about using Cerise. Our examples from Section 2.7 and Section 2.8 are modeled after examples from a paper about a high-level object capability language [138]. Because of the more low-level nature of our capability machine, we had to implement some abstractions ourselves (such as the calling convention in Section 2.7.3) but we think it is otherwise fair to say that our examples faithfully represent the examples used by Swasey et al., using the same granularity of encapsulation and attacker interaction. As such, this paper demonstrates that the low-level security primitives offered by our capability machine are expressive enough to implement high-level language abstractions, despite the stronger attacker model of a low-level adversary. At the same time, the examples show that Cerise is expressive enough to reason about these abstractions.

Cerise is the first instantiation of the Iris framework to such a low-level language and thus this work also demonstrates that the key features of Iris (such as guarded recursion, ghost state, and invariants) are equally applicable in this low-level setting as in the high-level settings they were originally intended for.

Of course, while we implement and reason about our examples directly in the capability machine assembly language, we are not proposing that real software should all be developed in that way. On the contrary, we think this is only realistic for low-level code in compiler back-ends [55, 131], operating systems and low-level security measures [135]. Other software should be developed and reasoned about in a more abstract setting, which suggests the need for a secure compiler that preserves high-level security guarantees in a low-level environment. In the context of capability machines, such compilers have been investigated already, both formally [44, 147], and practically [29, 116]. While we in this work have shown how to implement and reason about some high-level programming patterns at a low level, much interesting work remains to be done to further explore the design of a high-level language whose security abstractions map well to those offered by a capability machine.

An important aspect of the universal contract provided by our logical relation and fundamental theorem is that it formalizes the security guarantee of our capability machine without overspecifying implementations of the ISA. The contract specifies an authority bound that suffices to reason about adversarial code, but does not overly constrain future extensions or optimized implementations of the ISA. This is similar to how the ISA itself is designed to specify expected behavior that is sufficient for software authors to reason about their code without preventing CPU designers from constructing optimized or extended implementations. In fact, we believe universal contracts offer a general and powerful approach for formalizing ISA security guarantees. Such security guarantees are informally stated in informal ISA specifications but they have not yet been incorporated in formal definitions of ISAs [14, 23]. As such, a promising application of universal contracts like the one from Section 2.5 is to incorporate them into the ISA definition to formalize intended ISA security guarantees.

Finally, it is worth acknowledging that in this paper, we only describe a minimal capability machine that lacks many features from realistic capability machine ISAs. Our approach has been extended to support some additional features in the literature (e.g., local and uninitialized capabilities [55], and MMIO [135]), but other features are still missing for now (e.g. sealing, interrupts, virtual memory, etc.).

2.11 Related work

We now discuss several lines of work related to ours. First, we discuss earlier variants of Cerise by the authors and colleagues. Then, we discuss work on verifying object capability patterns in *high-level* languages, verification of ISA properties *in CHERI*, and other applications of *universal contracts* in the literature.

2.11.1 Earlier variants of Cerise

Earlier variants of Cerise focused on showing how capabilities can be used to implement a *secure, stack-based calling convention* [55, 131, 132] and *nested security wrappers* [135].

[131] were the first to show that capabilities can be used to implement a secure stack-based calling convention, i.e., a calling convention where the security guarantees of function calls at the machine code level are faithful to the high-level notion of a function call. They employed an additional kind of “local” capabilities and stack clearing to achieve security. Their work follows a similar methodology as the one described here, that is, they define a logical relation which characterizes a notion of safety. However, their proofs were not mechanized and the logical relation was defined using a non-trivial concrete model; in contrast we use the Cerise program logic to define and prove properties about our logical relation, which means that our development is done at a higher-level of abstraction and thus we, e.g., do not have to solve any recursive domain equations. In follow-up non-mechanized work, [132] achieved similar security guarantees with a novel calling convention based on so-called “linear” capabilities; capabilities that can never be duplicated. Although this calling convention avoids the stack clearing required in the previous work, linear

capabilities come with certain architectural restrictions [see e.g. 132, §6.2]. An efficient implementation of linear capabilities has so far not been demonstrated.

The subsequent work by [55] introduced a new type of capabilities (called "uninitialized") to avoid most of the stack clearing from Skorstengaard et al.'s first calling convention, thereby improving runtime efficiency. Importantly, uninitialized capabilities do not come with the same architectural hurdles as linear capabilities. As a second contribution, Georges et al. used Iris to formulate safety as a logical relation and mechanized their proofs of security.

The aforementioned logical relations of both Skorstengaard et al. and Georges et al. are more expressive and therefore significantly more complicated than the one presented here: they permit reasoning about revocation of local/linear/uninitialized capabilities and well-bracketedness properties of machine-code "function calls", on top of local-state encapsulation. In our present work, object capabilities ensure local state encapsulation, but we do not enforce calls and returns to be well-bracketed. In particular, we do not prevent an adversary from invoking a return pointer several times, or storing return pointers for later use. In other words, our calling convention implements the kind of function calls one has in a high-level language with control operators (e.g., call/cc), where calls and returns are not necessarily well-bracketed. (It is well-known that models of well-bracketed function calls are more involved than models of not-necessarily-well-bracketed function calls, see, e.g., [5, 39], and here we opted for the latter, to present a more accessible model, which suffices for a heap-based calling convention and for studying low-level implementations of object-capability patterns.)

In a different line of work, Strydonck et al. [135] employed a capability machine and logical relations model similar to the one presented here, but with additional support for MMIO, to verify safety properties for small, nestable wrappers around security-critical devices on a capability architecture. As part of the verification effort, multiple end-to-end security theorems were proven, which state that safety predicates of interest hold over the trace of IO events admitted by the machine. Here we have instead focused on demonstrating how a core model (without MMIO support) can be used to reason about low-level implementations of object-capability patterns.

2.11.2 Verifying object capability patterns in high-level languages

A number of high-level programming languages allow for programming patterns similar to object capabilities, that enable preserving local state while interacting with unknown code. Examples are closures, and high-level objects in capability safe languages.

[37] pioneered the use of a logical relation to give a semantic characterization of capability safety (earlier work used a more conservative syntactic approach based on whether or not objects contain references to each other and ignored the behaviour of objects). [37] focused on capability safety for a core calculus of Javascript, including a notion of observable effects, and used an explicit construction of their logical relation (not a program logic), which was the inspiration for the capability model by

[131] mentioned above and for the work by [138], who presented a program logic which allows reasoning modularly about object capability patterns in a high-level language. The methodology of [138] is close to the one presented here, but in contrast to [138] we reason about object capabilities on a low-level machine. For instance, Swasey et al. define two predicates to describe a reference: a predicate for "high integrity" locations ($\ell \hookrightarrow v$), and one for "low integrity" locations ($\mathsf{lowloc}\ \ell$). The first predicate grants exclusive access to the corresponding reference, and is therefore not safely shareable with an adversary. The second is shareable with an adversary, but can only be used to read and write "low integrity" values. In our setting, "high integrity" directly corresponds to the predicate $a \mapsto w$ for a memory location, and "low integrity" corresponds to the invariant used in the definition of $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Correspondingly, our definitions satisfy similar reasoning rules to the ones established by Swasey et al.. In particular, we believe that the various object capability patterns they verify can be implemented and verified in a similar way in the setting of a capability machine, using the principles presented in this paper. We demonstrated one such implementation by adapting their dynamic sealing example in Section 2.8. Additionally, the robust safety theorem of [138] is related to our template adequacy theorem with malloc and assert (Theorem 5); our assert flag plays a role similar to the OK flag in [138].

2.11.3 Verifying ISA properties in CHERI

[109] formally verify a number of "architectural" properties of CHERI capability machines. This constitutes a significant mechanization effort: the authors tackle the full generality of a realistic operational semantics for CHERI, which is significantly more complex than the minimal machine we consider here. The approach followed by Nienhuis et al. is different from ours: they state the properties they establish as trace properties, over a trace of "abstract actions" describing the various capabilities transiting through the machine during the execution. This approach makes it possible to state the desired properties in a very explicit and concrete fashion. For instance, the authors state and prove a property of "capability monotonicity": during the execution, the authority of available capabilities cannot increase (in other words, the machine does not allow forging new authority). Intuitively, this seems like a very reasonable property, required for proper operation of the capability machine. However, in practice it is more subtle: calls between components (in our case, jumping to an E-capability) do allow for some restricted form of non-monotonicity. The property proved by Nienhuis et al. is thus restricted to trace fragments that do not include calls to a different component. Our methodology is less explicit, but more expressive. In our setting, the fundamental theorem can be understood as expressing that "the machine works well". Its very extensional statement is admittedly harder to understand in terms of the operational semantics of the machine, but it enables deriving correctness statements in terms of the operational semantics that do apply to a full execution of the machine, including calls between an arbitrary number of components.

2.11.4 Other applications of universal contracts

As mentioned, our fundamental theorem constitutes a universal contract for arbitrary code, i.e., it allows deriving the guarantee that *any* adversarial capability is safe to execute, given validity of said capability. This safety is typically obtained by syntactically restricting the adversarial capability; e.g., requiring that the adressed memory only contains integers.[9] Similar notions of universal contracts have been used for high-level languages (explicitly or implicitly) in the literature. The aforementioned work of Skorstengaard et al. [131, 132], and Swasey et al. [138] all used a version of universal contracts, and placed varying syntactic restrictions on adversaries. The semantic type systems of Jung et al. [76] and Sammler et al. [121] permit similar reasoning about untrusted code based on a syntactic well-typedness restriction. The back-translation in the full-abstraction proof by Van Strydonck et al. [147] involved an explicit, universal separation logic contract for a C-like language with capabilities. Generally, whenever a semantic model is used to describe semantic guarantees satisfied by arbitrary code (possibly subject to syntactic restrictions), and when these guarantees are used in the manual verification of other code, this can be regarded as an application of a universal contract.

Acknowledgements Thanks to Léon Gondelman and Pierre Pradic for feedback on earlier drafts of this document.

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation; by the Research Foundation - Flanders (FWO); and by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU). Thomas Van Strydonck holds a Research Fellowship of the Research Foundation - Flanders (FWO). Amin Timany was postdoctoral fellow of the Flemish Research Foundation (FWO) during parts of this project.

[9] Note that instructions are encoded in memory as integers.

Chapter 3

Efficient and Provable Local Capability Revocation using Uninitialized Capabilities

This chapter is an extended version of the following conference publication:

> Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, Lars Birkedal.
> *Efficient and Provable Local Capability Revocation using Uninitialized Capabilities*
> Proceedings of the ACM on Programming Languages (POPL), 2021, 5.

The extension consists of

- Added technical details about the model used to reason about domain transfers in the presence of uninitialized capabilities; Section 3.6.6.
- New section detailing how the model is applied when reasoning about domain transfers; Section 3.6.7
- Detailed part of the proof of a key theorem in the paper; Section 3.6.8
- Detailed the proof of an application of the model on an example; Section 3.6.9

Abstract

Capability machines are a special form of CPUs that offer fine-grained privilege separation using a form of authority-carrying values known as capabilities. The CHERI capability machine offers local capabilities, which could be used as a cheap but restricted form of capability revocation. Unfortunately, local capability revocation is unrealistic in practice because large amounts of stack memory need to be cleared as a security precaution.

In this paper, we address this shortcoming by introducing *uninitialized capabilities*: a new form of capabilities that represent read/write authority to a block of memory without exposing the memory's initial contents. We provide a mechanically verified program logic for reasoning about programs on a capability machine with the new feature and we formalize and prove capability safety in the form of a universal contract for untrusted code. We use uninitialized capabilities for making a previously-proposed secure calling convention efficient and prove its security using the program logic. Finally, we report on a proof-of-concept implementation of uninitialized capabilities on the CHERI capability machine.

3.1 Introduction

Capability machines are a type of CPUs with support for fine-grained privilege separation, dating back to the 1960s [36, 91][157]. In this paper, we will specifically focus on a recent family of capability machines called CHERI [157]. Capability machines provide native support for capabilities: values which represent a certain authority to interact with memory, the operating system or other isolated components in the system. Capabilities come in several forms. Memory capabilities represent the authority to access a certain region of memory with a certain permission (e.g. RW or RX). On many capability machines, including CHERI, memory capabilities are designed to directly replace pointers, thus adding native bounds and permission checks with almost zero runtime overhead.

Additionally, capability machines usually offer a form of object capabilities [97]: a form of reified closures that represent the authority to invoke an isolated component without exposing its internal state and its private capabilities. Invoking such an object capability passes control to the other component and makes available its private capabilities and thus, its authority. As such, they offer a cheap form of context switches. On CHERI, object capabilities take the form of pairs of code and data capabilities, tied together by being sealed with a common seal [154, 155]. Sealing is a primitive CHERI operation that renders capabilities opaque and unusable, except that the pair can be invoked with a special instruction `CCall`.

Local capabilities are a new feature of CHERI [154]. Conceptually, they are intended as a form of ephemeral capabilities that can be used directly but not stored for later use. More technically, they are a form of capabilities that can be kept in registers but not stored in memory. There is, however, an exception to the latter rule: local capabilities can be stored in memory through memory capabilities with special "write-local" permission. This exception is specifically intended for the stack capability, so that the stack can be used for spilling local capabilities from registers and function arguments.

In principle, local capabilities make it possible to pass a capability to an untrusted component temporarily, without allowing the component to store it for later use. In other words, if the component is invoked again, the local capability is effectively revoked: the component cannot have access to it anymore. As such, local capabilities can be seen as a restricted revocation primitive with little performance overhead.

Unfortunately, this potential is not realized in practice. While CheriBSD (an adaptation of FreeBSD which makes use of CHERI capabilities) does use local capabilities to represent stack pointers, they work with private per-compartment stacks, and local capabilities are never passed to untrusted code in other compartments [154]. Hence, the CheriBSD system does not actually rely on local capabilities for enforcing security properties but only to mitigate the impact of potential bugs; specifically, to prevent accidental leaks of stack pointers. The latest CHERI ISA reference document mentions two additional dimensions of locality (kernel vs. user-space memory, garbage-collected vs manually managed memory), but neither involves a form of revocation [157, §D.13].

The likely reason for this limited use of local capabilities as a revocation mechanism is that its guarantees only hold under an important restriction. If we want to revoke a local capability before a second invocation of untrusted adversarial code, we must make sure not to accidentally leak an old copy of the capability. While local capability rules ensure that such old copies can never end up in heap memory (because no write-local capabilities to heap memory exist), they may still be present in any location where the adversary may have previously stored them: capability registers, but also any region of memory which it had a write-local memory capability for. Practically, the only way accidental leaking can be avoided is by clearing unused registers and sweeping over this write-local memory to clear it entirely or at least erase local capabilities. For example, in a secure calling convention built on local capabilities, Skorstengaard et al. [129] have to clear the entire unused part of the stack before any invocation of adversarial code. This requirement is very costly in practice, and also hard to avoid, since the stack must be made write-local if we want to allow invoked code to spill registers or store local capabilities away during sub-invocations. The performance impact might be mitigated with special hardware support [72], but it is unclear whether this is enough to make it realistic for practical use.

In this paper, we propose a way to redeem local capabilities as a restricted but efficient revocation primitive using *uninitialized capabilities*. This is a new form of capabilities that represents read-write access to a region of memory without access to its current contents. Regions of memory which the adversary has previously had write-local access to, specifically the stack, can be made available to the adversary through an uninitialized capability without the need to clear the memory beforehand. Technically, an uninitialized capability's range of authority is divided into two parts: the range *below* the address currently pointed to, say $[b,a)$, and the range *above* the current address, say $[a,e)$. The range below represents the initialized part of the capability, and the range above represents its uninitialized part. The capability grants read-write access to $[b,a)$, and write-only access to $[a,e)$. However, if the address a is written to, the boundary between the two parts is automatically changed to include the now-overwritten memory location, i.e., a is automatically incremented (pushing a value on the stack in the case of a stack capability). An uninitialized capability can be restricted by lowering the current address and thus “uninitializing" a range of memory (popping the stack), but its authority can only be increased by writing to it, thus overwriting its previous content. Additionally, regular capabilities can be made uninitialized and an uninitialized capability to $[b,e)$ can be restricted to a regular read-write capability to its initialized part $[b,a)$ which can be passed to existing code.

Although uninitialized capabilities are more generally useful, this paper focuses on how they redeem local capabilities as a revocation primitive. To this end, we formally establish the guarantees provided by local and uninitialized capabilities with a capability safety result based on the one by Skorstengaard et al. [129]. Capability safety is expressed as a universal contract—or specification—that holds for arbitrary assembly code. The universal contract is defined using a logical relation which captures the authority represented by a capability, and guarantees that this authority is respected and monotonically preserved by arbitrary assembly code. To simplify the definition of the logical relation and avoid some tedious book-keeping related to step-indexing and shared logical state, we make use of a program logic for our capability machine model which we define using the Iris program logic framework [74, 75, 77, 82]. We have mechanized all of the technical development using the Iris implementation in Coq [83, 84].

Our program logic and logical relation are the most important technical contributions of this work. To allow reasoning about the pattern of local capability revocation, we use a novel combination of Iris' invariants and saved predicates with more traditional Kripke world-indexing. We use this Kripke world-indexing with public/private transitions [39, 129] and a new idea of what we call frozen regions to support typical patterns of (temporary) local capability revocation.

To demonstrate both how uninitialized capabilities redeem local capabilities as a revocation primitive in practice and how our capability-safety result enables reasoning about programs using these features, we study a modification of Skorstengaard et al. [129]'s calling convention that avoids the problematic clearing of large parts of the stack. The resulting calling convention is another contribution in its own right. We demonstrate how our program logic can be used to prove correctness of programs using the calling convention, specifically for the classic "awkward" example which relies on well-bracketed control flow and stack frame encapsulation. The mechanization is highly called for because of the low-level nature of capability machines, and the large amount of bookkeeping that is necessary for reasoning about example programs (arithmetic manipulation of addresses, restriction of all relevant capabilities, setup of activation records, etc.).

Finally, more practically, we provide evidence that uninitialized capabilities can be realistically added to the CHERI capability machine by implementing them in the CHERI-MIPS ISA and the definition of its operational semantics in SAIL [13]. Additionally, we add support for the new instructions to the Clang/LLVM assembler. The simulator that we thus obtain from SAIL and the modified assembler have been used to experiment with the new calling convention in manually modified assembly programs.

To summarize, our contributions are centered around the new uninitialized capabilities:

- We propose uninitialized capabilities: a new form of capabilities that represents read-write access to memory without exposing the memory's initial contents (Section 3.4).

- We explain how uninitialized capabilities redeem CHERI's local capabilities as a restricted but efficient revocation primitive (Section 3.4).
- We characterize the combined guarantees of the two features with a capability-safety result, mechanized in Coq, as a universal contract that holds for arbitrary assembly programs. It uses a logical relation and a novel combination of Iris features like guarded recursion and shared invariants, with Kripke world-indexing and public/private transitions for reasoning about local capability revocation (Sections 3.5 and 3.6).
- We define a modified version of the calling convention of Skorstengaard et al. [129] which removes its performance problems. We provide evidence that it enforces well-bracketed control flow and local stack frame encapsulation by proving an implementation of the awkward example correct (Section 3.6.9).
- We implement uninitialized capabilities in the SAIL semantics of CHERI-MIPS and the Clang/LLVM assembler and use them to experiment with the modified calling convention (Section 3.7).

Finally we add that, to the best of our knowledge, our Iris-Coq mechanization of capability safety is the first mechanically verified account of key deep semantic properties (spanning several components, including unknown adversarial code) that are enforceable using capabilities. The Iris-Coq mechanization can be found at `https://github.com/logsem/cerise-stack/releases/tag/POPL2021`.

The idea and implementation of uninitialized capabilities has also been reported in the master thesis of one of Sander Huyghebaert [68], overlapping partly with Sections 3.4 and 3.7.

3.2 A capability machine with local capabilities

This section defines the operational semantics of our capability machine. Our machine model is defined along the same lines as the one from Skorstengaard et al. [129], and hence transitively draws from CHERI [154] and the M-Machine [27]. In Section 3.2.1 we describe the operational semantics for a bare-bones capability machine (without local and uninitialized capabilities) as a starting point. Then, we add support for local capabilities in Section 3.2.2. The semantics for uninitialized capabilities will be treated later in Section 3.4, resulting in the full definition of the capability machine semantics we assume in the rest of the paper.

Figures 3.1 to 3.5 summarize the operational behavior of our capability machine, and will be referenced on multiple occasions. They are color-coded as follows: the bare-bones capability machine is defined in black; additions related to local capabilities are typeset in red. Finally blue additions, introduced on top of the red ones, account for uninitialized capabilities and will be discussed in Section 3.4.

$$
\begin{array}{llll}
a & \in & \mathrm{Addr} & \triangleq \quad [0, \mathrm{AddrMax}] \\
p & \in & \mathrm{Perm} & ::= \quad \textsc{o} \mid \textsc{e} \mid \textsc{ro} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{rwx} \\
 & \multicolumn{3}{l}{\mid \textsc{rwl} \mid \textsc{rwlx} \mid \textsc{urw} \mid \textsc{urwl} \mid \textsc{urwx} \mid \textsc{urwlx}} \\
g & \in & \mathrm{Global} & ::= \quad \textsc{global} \mid \textsc{local} \\
c & \in & \mathrm{Cap} & \triangleq \quad \{(p,g,b,e,a) \mid b,e,a \in \mathrm{Addr}\} \\
w & \in & \mathrm{Word} & \triangleq \quad \mathbb{Z} + \mathrm{Cap} \\
r & \in & \mathrm{RegName} & ::= \quad \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots
\end{array}
$$

$$
\begin{array}{llll}
reg & \in & \mathrm{Reg} & \triangleq \quad \mathrm{RegName} \rightharpoonup \mathrm{Word} \\
m & \in & \mathrm{Mem} & \triangleq \quad \mathrm{Addr} \rightharpoonup \mathrm{Word} \\
\varphi & \in & \mathrm{ExecConf} & \triangleq \quad \mathrm{Reg} \times \mathrm{Mem} \\
\delta & \in & \mathrm{DoneState} & ::= \quad \mathsf{Standby} \mid \mathsf{Halted} \mid \mathsf{Failed} \\
\mu & \in & \mathrm{ExecMode} & ::= \quad \mathsf{SingleStep} \mid \mathsf{Repeat}\ \mu \mid \mathsf{Done}\ \delta
\end{array}
$$

$$
\begin{array}{lll}
\rho & \in & \mathbb{Z} + \mathrm{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
 & & \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt} \mid \\
 & & \texttt{loadU}\ r\ r\ \rho \mid \texttt{storeU}\ r\ \rho\ \rho \mid \texttt{promoteU}\ r
\end{array}
$$

Figure 3.1: Machine words, machine state and instructions.

3.2.1 Bare-bones Capability Machine

Figure 3.1 defines the syntax we use in our capability machine. The set of addresses Addr is finite, to make our model more realistic, and described by the integer range $[0, \mathrm{AddrMax}]$. The address AddrMax is the top address and cannot be dereferenced.

A memory word $w \in \mathrm{Word}$ is either an (unbounded) integer or a capability c. Capabilities are of the form (p,g,b,e,a) and allow exerting permission p on the memory range $[b,e)$, while currently pointing to a. The permissions p and locality bit g appear in the permission and locality lattices of Figure 3.2, which induce a bottom-to-top partial order $\preccurlyeq$ on permissions, localities and pairs thereof. The locality bit g only plays a role in the presence of local capabilities, and will be covered later in Section 3.2.2. The permission lattice, on the other hand, contains six different types of permissions; the null (O), read-only (RO), enter (E), read/write (RW), read/execute (RX) and read/write/execute (RWX) permissions. The sole non-standard permission, E, is inspired by the M-Machine [27], and referred to as "sealed entry" or "sentry" capabilities in CHERI [157]. Enter capabilities represent opaque closures, or object capabilities, encapsulating code and data, and hence cannot be read, written, executed or modified. They can only be jumped to, thereby loading them into the pc register and changing their permission from E to RX, effectively unsealing them. Chapter 2 illustrated the use of enter capabilities as primitives for creating boundaries between

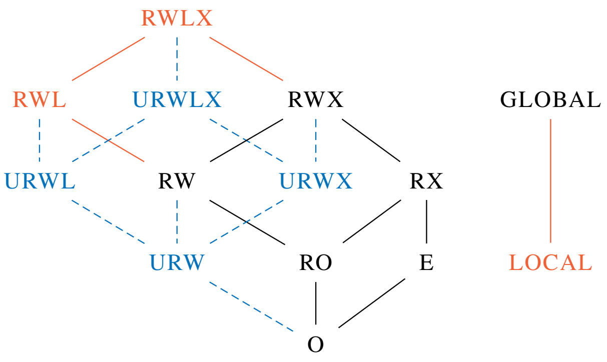


Figure 3.2: Permission and locality hierarchy.

REPEATSINGLE

$$\frac{(\mathsf{SingleStep},\ \varphi) \rightarrow (\mathsf{Done}\ \delta,\ \varphi')}{(\mathsf{Repeat\ SingleStep},\ \varphi) \rightarrow (\mathsf{Repeat}\ (\mathsf{Done}\ \delta),\ \varphi')}$$

REPEATSTANDBY

$$\begin{array}{l}(\mathsf{Repeat}\ (\mathsf{Done\ Standby}),\ \varphi)\\ \quad \rightarrow (\mathsf{Repeat\ SingleStep},\ \varphi)\end{array}$$

REPEATHALT

$$\begin{array}{l}(\mathsf{Repeat}\ (\mathsf{Done\ Halted}),\ \varphi)\\ \quad \rightarrow (\mathsf{Done\ Halted},\ \varphi)\end{array}$$

REPEATFAIL

$$\begin{array}{l}(\mathsf{Repeat}\ (\mathsf{Done\ Failed}),\ \varphi)\\ \quad \rightarrow (\mathsf{Done\ Failed},\ \varphi)\end{array}$$

EXECSINGLE

$$(\mathsf{SingleStep},\ \varphi) \rightarrow \begin{cases} [\![\mathit{decode}(z)]\!](\varphi) & \text{if } \ \varphi.\mathrm{reg}(\mathrm{pc}) = (p,g,b,e,a) \\ & \quad \wedge\ b \leq a < e \\ & \quad \wedge\ p \in \{\mathrm{RX}, \mathrm{RWX}, \mathrm{RWLX}\} \\ & \quad \wedge\ \varphi.\mathrm{mem}(a) = z \\ (\mathsf{Done\ Failed},\ \varphi) & \text{otherwise} \end{cases}$$

Figure 3.3: Operational semantics: reduction steps.

domains that guarantee local state encapsulation. Now, we apply similar ideas, but in the presence of a shared call-stack, while also enforcing well-bracketed control flows.

The machine's instructions i either operate on register names r, or on sums ρ of registers and constants. We detail their semantics below.

The state of the machine is modeled by the semantics as a configuration φ, containing the state of the registers $\varphi.reg$ and the memory $\varphi.m$. A register file reg consists of a map from register names r to words, while the memory m maps addresses to words.

Figure 3.3 defines the small-step operational semantics for the capability machine. At each step, the machine's state is described by an execution mode μ and a configuration φ. The mode μ models the machine's instruction cycle, which loops

$$\mathrm{updPC}(\varphi) =$$

$$\begin{cases} (\mathsf{Done\ Standby}, \varphi[\mathrm{reg}.\mathsf{pc} \mapsto (p,g,b,e,a+1)]) & \text{if } \varphi.\mathrm{reg}(\mathsf{pc}) = (p,g,b,e,a) \\ (\mathsf{Done\ Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\mathrm{getWord}(\varphi,\rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{if } \rho \in \mathrm{RegName} \end{cases}$$

Figure 3.4: Operational semantics: auxiliary definitions.

infinitely (expressed by Repeat SingleStep) until it reaches a successful done state Done Halted through REPEATHALT or a failed state Done Failed through REPEATFAIL. The REPEATSINGLE rule allows for the execution of single instructions through the EXECSINGLE rule. If the execution of the instruction is successful, i.e. execution in EXECSINGLE does not fail or halt and results in a Done Standby state, then REPEATSTANDBY allows for another iteration of the processor's instruction cycle.

An execution step (EXECSINGLE) requires an executable and in-bounds capability in the pc register, failing otherwise. It reads the word z at the memory address a, decodes it and executes the result on the current state φ, denoted $[\![decode(z)]\!](\varphi)$. Figure 3.5 defines the operational behavior $[\![i]\!](\varphi)$ for a number of representative instructions i, using the auxiliary definitions in Figure 3.4. The notation $\in$ is overloaded to deconstruct sum types, e.g. if $\rho \in \mathbb{Z} + \mathrm{RegName}$, then the statement $\rho \in \mathbb{Z}$ will automatically unwrap ρ if it is of the form inl _ and fail otherwise. Most instructions use the auxiliary function updPC to increment the pc register after their proper operations. Because the address space is finite, pointer arithmetic such as e.g. $a+1$ can result in illegal addresses, and *should* hence be represented as an option type. To avoid notational clutter, we assume this option type to be automatically unpacked through in the entire figure, resulting in failure in case of a None result. If an instruction operates on a value ρ, it either uses the constant value directly if $\rho \in \mathbb{Z}$, or it reads the value from the register if $\rho \in \mathrm{RegName}$. In what follows, *the contents of* ρ will be used to signify the resulting value of either option.

We now describe the semantics of instructions, in particular those listed in Figure 3.5. The `fail` and `halt` instructions terminate execution in the Failed and Halted state respectively. `move` r ρ copies the contents of ρ into r. Memory is accessed using the `load` and `store` instructions: `load` r_1 r_2 reads the value pointed by the capability in r_2 provided it has the permission R and points within bounds, and `store` r ρ stores the contents of ρ through the capability in r provided it has the W permission and points within bounds. The `jmp` instruction jumps to a capability, by writing it into the pc register. In the case of an enter (E) capability, it unseals it into a RX capability first, allowing us to jump to opaque closures, as previously mentioned. Three instructions allow modifying capabilities. `restrict` r ρ allows restricting the permission and locality of a capability, by decoding the contents of ρ into a pair (p',g'), and provided it is less permissive than the current permission-locality-pair of r according to $\preccurlyeq$,

i	$[\![i]\!](\varphi)$	Conditions
`fail`	$(\mathsf{Done\ Failed}, \varphi)$	
`halt`	$(\mathsf{Done\ Halted}, \varphi)$	
`move` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho)$
`load` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,g,b,e,a)$ and $w = \varphi.\mathrm{mem}(a)$ and $b \leq a < e$ and $p \in \{\mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}, \mathrm{RWL}, \mathrm{RWLX}\}$
`store` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $b \leq a < e$ and $p \in \{\mathrm{RW}, \mathrm{RWX}, \mathrm{RWL}, \mathrm{RWLX}\}$ and $w = \mathrm{getWord}(\varphi, \rho)$ and if $w = (\_, \mathrm{LOCAL}, \_, \_, \_)$, then $p \in \{\mathrm{RWLX}, \mathrm{RWL}\}$
`jmp` r	$(\mathsf{Done\ Standby},$ $\varphi[\mathsf{reg.pc} \mapsto \mathit{newPc}])$	if $\varphi.\mathrm{reg}(r) = (\mathrm{E},g,b,e,a)$, then $\mathit{newPc} = (\mathrm{RX},g,b,e,a)$ otherwise $\mathit{newPc} = \varphi.\mathrm{reg}(r)$
`restrict` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $(p',g') = \mathrm{decodePermPair}(\mathrm{getWord}(\varphi,\rho))$ and $(p',g') \preccurlyeq (p,g)$ and $w = (p',g',b,e,a)$
`subseg` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi,\rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \mathrm{E}$ and $w = (p,g,z_1,z_2,a)$
`lea` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $z = \mathrm{getWord}(\varphi,\rho)$ and $p \neq \mathrm{E}$ and $w = (p,g,b,e,a+z)$ and if $p = \mathrm{U}\text{-}$, then $z \leq 0$
`geta` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto a])$	$\varphi.\mathrm{reg}(r_2) = (\_,\_,\_,\_,a)$
`loadU` $r_1\ r_2\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,g,b,e,a)$ and $p = \mathrm{U}\text{-}$ and $\mathit{off} = \mathrm{getWord}(\varphi,\rho)$ and $b \leq a + \mathit{off} < a \leq e$ and $w = \varphi.\mathrm{mem}(a + \mathit{off})$
`storeU` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi'$ $[\mathrm{mem}.(a+\mathit{off}) \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $p = \mathrm{U}\text{-}$ and $\mathit{off} = \mathrm{getWord}(\varphi,\rho_1)$ and $w = \mathrm{getWord}(\varphi,\rho_2)$ and if $w = (\_, \mathrm{LOCAL}, \_, \_, \_)$ then $p \in \{\mathrm{URWLX}, \mathrm{URWL}\}$ and $b \leq a + \mathit{off} \leq a < e$ and if $\mathit{off} \neq 0$ then $\varphi' = \varphi$ else $\varphi' = \varphi[\mathrm{reg}.r \mapsto (p,g,b,e,a+1)]$
`promoteU` r	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $p = \mathrm{U}\pi$ and $w = (\pi,g,b,\mathrm{min}(a,e),a)$
$\ldots$		
$\_$	$(\mathsf{Done\ Failed}, \varphi)$	otherwise

Figure 3.5: Operational semantics: instruction semantics.

restricts r accordingly. `subseg` $r\ \rho_1\ \rho_2$ takes a subsegment of a capability range of authority. It uses the contents of ρ_1 and ρ_2 to restrict the range of authority of the capability in r, in case r is not an enter capability. Note that the inequality $0 \leq z_2 \leq e$ suffices to guarantee monotonicity of authority, since if $z_2 \leq z_1$, then the capability provides no authority over memory whatsoever. `lea` $r\ \rho$ modifies the address of the capability in r, by adding to it the integer offset in ρ. As expected, `lea` fails for enter capabilities. A number of instructions allow inspecting capabilities. We show `geta` that retrieves the address field of a capability; `getp`, `getl`, `getb` and `gete` work similarly for the other fields. Not shown in Figure 3.5 are `jnz` (conditional jump), arithmetic instructions (`add`, `sub`, `lt`) and `isptr` which checks whether a word is a capability. Finally, if the capability checks for an instruction are not satisfied, the last row defines the resulting state as $(\mathsf{Done\ Failed}, \varphi)$.

3.2.2 Capability Machine with Local Capabilities

The red parts of Figures 3.1 to 3.3 and 3.5 add local capabilities to our bare-bones capability machine. The locality hierarchy in Figure 3.2 receives a second element, LOCAL. As evident from this hierarchy, the `restrict` instruction allows deriving local capabilities from global ones, but not vice versa.

Local capabilities can only be stored to memory through capabilities with a write-local permission, a stronger version of the W permission that we denote as WL. The permission hierarchy in Figure 3.2 contains the two new write-local permissions RWL and RWLX at the top. The permission RWLX is a valid additional permission for the pc-register, as shown in Figure 3.3. The `restrict` instruction follows the order $\preccurlyeq$ and allows deriving writable capabilities from write-local ones.

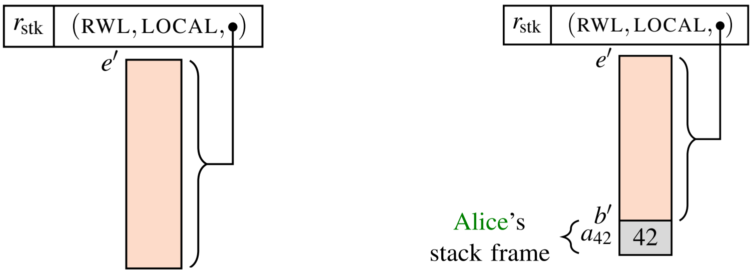
The semantics of locality comes into play when interacting with memory, i.e. in the `load` and `store` instructions in Figure 3.5. Both `load` and `store` permit loading, respectively storing, using the two new permissions. Additionally, `store` only permits storing local values if the capability's permission allows local writes.

3.3 Revocation using local capabilities

We now discuss the use of local capabilities as a (inefficient) revocation primitive. We use an incremental example consisting of **three scenarios** which build towards the secure calling convention of Skorstengaard et al. [129]. It will become clear why local capability revocation and the calling convention incur inherent performance issues because of stack clearing.

3.3.1 Using Local Capabilities for Revocation

Consider the following scenario, which we will refer to as **Scenario 1**: a client, Alice, wishes to invoke an untrusted adversary, `Bob`, twice. Alice owns a capability, c, that she wishes to share with `Bob`, through a register r, but only for the duration of the first call. During the second call to `Bob`, he should not be able to access the capability any



(a) The stack capability Alice starts out with.

(b) The stack capability Alice hands to Bob.

Figure 3.6: Register state for Scenario 2, involving a write-local stack capability.

more. In other words, Alice wishes to revoke capability c before the second call. If c is a GLOBAL capability, i.e. $c = (p, \text{GLOBAL}, b, e, a)$, Bob can simply store c in any part of memory he has access to, during the first invocation, and retrieve it during the second, thwarting Alice's plans. This is where local capabilities come in. In case c is local, i.e. $c = (p, \text{LOCAL}, b, e, a)$, and we disregard write-local permissions for the moment, Bob cannot store the capability c to memory for later use, and can therefore not recover c during the second invocation, provided Alice cleared it from the registers before the second call. In other words, as soon as Bob returns to Alice, Bob's access to c is effectively *revoked*.

3.3.2 Write-local Memory and Stack Clearing

The situation changes when we consider the existence of write-local permissions in an extended **Scenario 2**. Specifically, we extend Scenario 1 to handle the stack explicitly, through a local, write-local stack capability c_{stk} stored in a register we call r_{stk}, as shown in Figure 3.6a. First, Alice wants to enforce *local state encapsulation*, i.e. ensure that Bob cannot gain access to her local stack frame, including the value 42 stored at address a_{42} in Figure 3.6b. This is trivially enforced by not passing Bob a reference to the full stack capability. Second, Alice wants to enforce the temporal ownership of c, by revoking Bob's ownership of it before the second call.

Concretely, c_{stk} carries RWLX permission. The c_{stk} capability is write-local, to allow spilling of local arguments and other capabilities onto the stack. It will become clear in Section 3.3.3 why c_{stk} needs an execute permission. Finally, we cannot allow c_{stk} to be GLOBAL, since Bob could then, during the first invocation, store c_{stk} to memory, write c into stack memory through c_{stk}, and then, during the second invocation, read c again after retrieving c_{stk}. Since GLOBAL, write-local capabilities clearly break any attempts at building a sensible revocation schema using local capabilities, we forbid their existence.

Figure 3.6a shows the initial contents of r_{stk}, when Alice starts executing. When Alice calls Bob, she will `restrict` the stack capability and pass the unused part of the stack in r_{stk}, as shown in Figure 3.6b. At the time of the first call, we set

$c_{\text{stk}} = (\text{RWLX}, \text{LOCAL}, b', e', b')$. For simplicity, we assume that c_{stk} has the same value on the second call, i.e. Alice's stack frame does not change size in between calls. Notice that it is currently unclear how Alice obtains this c_{stk} capability for the second call, since c_{stk} itself is local and hence not easily stored in between the first and second call to Bob. We will clarify this point in Section 3.3.3. We also assume that the memory addressed by c_{stk} is initially zeroed out.

Alice still wants to prohibit Bob from accessing c during the second call. But now, Bob *does* have a way of storing local capability c during the first invocation; he can store it anywhere in $[b', e')$ through the write-local capability c_{stk}. Therefore, Alice has to make sure that the region $[b', e')$ does not contain any copies of c before invoking Bob a second time. The solution is to clear $[b', e')$ before the second invocation, or more generally, *clear all write-local memory* that Bob had access to. This means a potentially large runtime overhead, since the region $[b', e')$ may be quite large in practice. Note that we assume (here and elsewhere in the paper) that the stack is the only memory region that has write-local capabilities pointing into it; otherwise, Alice would have to find and clear all other write-local regions that Bob might have had access to as well.

3.3.3 A Secure Calling Convention using Local Capabilities

Having discussed the core performance issue in the calling convention of Skorstengaard et al. [129], namely the stack clearing caused by the use of local capabilities, we now extend our previous scenario to their full secure calling convention. Concretely, we need to make two additions.

First, the astute reader may have noticed that our scenario from Section 3.3.2 does not actually work. The problem is that after Bob returns, Alice has no capability to erase Bob's part of the stack $[b', e')$, or to access her old stack frame, since Alice's stack capability was itself local, and could only have been stored on the stack itself. We could require Bob to return his stack capability, but Alice would still have no way of accessing her own stack frame after the first call to Bob. To remedy this, Skorstengaard et al. [129] have Alice create a kind of return closure on the stack, and pass a capability for invoking it to Bob as a return capability c_{ret}. This capability is represented as an enter capability and points to restoration instructions, also called trampoline instructions, pushed onto Alice's stack frame, along with her stack pointer, before invoking Bob. When executed, these trampoline instructions reinstate Alice's old stack pointer and then resume execution by loading a previously pushed value for the pc register. The execution of these instructions on the stack is the reason we gave c_{stk} execute permission in Section 3.3.2. This constraint can be alleviated when using alternative object capability models, such as with code and data sealed pairs, where the code part can be in a separate memory space from the local data part, which in this case would be stored on the stack [132]. Since enter capabilities are opaque, Bob can only use c_{ret} as a jumping destination, and when he does, Alice's old return pointer is restored. Bob cannot simply store the capability c_{ret} for later use, since c_{ret} is itself local, as it was derived from the local capability c_{stk} using `restrict`.

Secondly, to ensure generality, we have to assume that Alice is called by a second untrusted party, Charlie, rather than being allowed to initiate execution. In this **Scenario 3**, the stack capability c_{stk} in Figure 3.6a that was previously assumed to be initially zeroed, is now passed to Alice by Charlie. Charlie has the option to protect his own stack frame by calling Alice in a fashion similar to Figure 3.6b. Alice again wishes to revoke Bob's access to c and to enforce local state encapsulation. With the introduction of a second adversary, Alice now also has the extra goal of enforcing *well-bracketed control flow*, i.e. ensure that Bob cannot bypass Alice and return to Charlie directly. To achieve all three goals, Alice needs to make sure that c_{stk} does not contain any capabilities that Bob should not have access to when invoking him. Since Charlie has access to a larger stack capability than both Alice and Bob, and could have stored his stack or return pointer high up in the stack, Alice now has to additionally erase the entire memory region $[b', e')$ even before the first call to Bob.

When Alice returns to Charlie, Skorstengaard et al. [129] originally proposed to erase the entire stack $[b', e')$ again, but as they later point out, it suffices for Alice to clear her own stack frame when returning to Charlie [131]. This is because any stack capabilities that Bob might want to smuggle to Charlie through the stack, ultimately originate from Charlie in the first place, and are not of any added value to him. Sharing his return pointer with Charlie will do Bob no good either, since it will jump to an address within Charlie's own stack. The formalization of this previously informal observation is one of the novelties in our logical relation in Section 3.6.

3.4 Uninitialized Capabilities

Now let us introduce *uninitialized* capabilities in Section 3.4.1 and see how they can be used to solve the issue of stack clearing in Section 3.4.2.

3.4.1 Adding Uninitialized Capabilities to the Capability Machine

Uninitialized capabilities are a new form of capabilities that represent read-write ability to a region of memory without access to its *current* contents. More specifically, they are represented as new permissions that are counterparts of the ones that have at least read-write ability. The blue labels in Figure 3.2 represent the additions to our permission lattice.

An uninitialized capability $(\text{U}\pi, g, b, e, a)$ has permission π on the range $[b, a)$ (the *initialized* part) and *write-only* permission on the range $[a, e)$ (the *uninitialized* part), assuming $b \leq a < e$ for simplicity. For instance, if π is RWX, then the capability can read, write, or execute anything in the initialized part of the capability, but can only write to the uninitialized part.[1] The initialized part of the capability can be extended by writing to the first uninitialized address, i.e. a.

[1] Using an URWLX, URWX or URX capability to execute is actually only possible after first initializing (a part of) it and converting it to a regular capability using `promote`, as explained in the next paragraph.

Capabilities that have at least read-write permissions can be restricted to their uninitialized counterparts. Uninitialized capabilities can be further restricted w.r.t. the initialized part, e.g., an URWLX permission can be restricted to an URW permission. Since an uninitialized capability (Uπ, g, b, e, a) represents authority π on the initialized part $[b,a)$, we also allow converting it to a regular capability (π, g, b, a, a) with authority π on the initialized range $[b,a)$, using a new `promote` instruction. We will make use of this instruction to construct return capabilities in Section 3.4.2.

We now discuss the changes to the operational semantics, indicated in blue in Figure 3.5. Instead of modifying `load` and `store` to support uninitialized capabilities, we define two new instructions `loadU` and `storeU` that can only be used with uninitialized capabilities. `loadU` r_1 r_2 ρ first checks that r_1 contains a capability (Uπ, g, b, e, a), that $b \leq a + \mathit{off} < a < e$ (where off is the contents of ρ). If both checks succeed, the value at address $a + \mathit{off}$ will be loaded into register r_2. Similarly, `storeU` r ρ_1 ρ_2 checks that r contains a capability (Uπ, g, b, e, a) and $b \leq a + \mathit{off} \leq a < e$ (with off the contents of ρ_1). It will then store the value in ρ_2 into the address $a + \mathit{off}$. If $\mathit{off} = 0$, then the capability in r is incremented.

From a hardware implementation perspective, the new `loadU` and `storeU` instructions do perform more work than `load` and `store`. In particular, they additionally need to compute an addition and an extra bounds check. Nevertheless, we expect that this should not drastically change the implementation complexity or the critical path for our new instructions. Woodruff et al. [162] show that bounds checks can typically be made efficient by running them in parallel with memory accesses: “any bounds check on the virtual address can be performed in parallel to [address] translation, making memory access a particularly convenient time to perform a bounds check”. We believe the same optimisation could be applied to an implementation of `loadU` and `storeU`.

Finally, one instruction must be slightly modified: we cannot allow `lea` to increase the current address of an uninitialized capability, as this would increase its read authority. Therefore, when using `lea` to change the address of a capability (Uπ, g, b, e, a) to a', the machine additionally checks that $a' \leq a$.

3.4.2 A New Calling Convention

Description of the calling convention. With uninitialized capabilities, we can now revisit the calling convention from Section 3.3.3 and use uninitialized capabilities to avoid the stack clearing requirement and fix its performance issues. Instead of using a RWLX stack capability, we give it permission URWLX. Let us consider again the example from Section 3.3.3, but let Alice pass the capability c_{stk} = (URWLX, LOCAL, b, e, b) to Bob. Bob now cannot use c_{stk} to read the contents of $[b,e)$ without overwriting it first, so stack clearing is no longer needed.

Alice still needs to provide an enter capability c_{ret} as a return pointer to Bob. However, Alice must now first `promote` c_{stk} back into a RWLX capability before she can use `restrict` on it to create the return capability. When Alice returns to Charlie, Charlie regains access to the entire stack, so Alice still needs to clear her own stack frame. This clearing requirement is very reasonable compared to the earlier case, as Alice only needs to clear the part of the stack she has actually used.

We recap the new calling convention formally:

At program start-up. A local URWLX capability stack pointer is in register r_{stk}.

When called by an adversary. Check that the received stack pointer has permission URWLX.

Before calling an adversary. Push activation record to the stack and create a local E-capability to use as return pointer. `Subseg` the stack capability to the unused part. Clear non-argument registers.

Before returning to an adversary. Clear non-return-value registers and the part of the stack we used.

While the changes may seem simple, there are some details to get right. Let's revisit **Scenario 3** from Section 3.3.3, and assume Alice receives stack capability $c_{\text{stk}1}$ = (URWLX, LOCAL, b_1, e, b_1) from Charlie and uses range $[b_1, b_2)$ to store data. She now calls Bob with $c_{\text{stk}2}$ = (URWLX, LOCAL, b_2, e, b_2). Suppose that after this first call, Alice needs less stack space. She can instead provide $c_{\text{stk}3}$ = (URWLX, LOCAL, b_3, e, b_3) as stack capability to Bob with $b_1 \leq b_3 \leq b_2$ for the second call. Alice does not need to clear the range $[b_3, b_2)$ since Bob cannot possibly read it as it is uninitialized. However, when returning to Charlie, Alice must be careful to clear everything she has ever written to, i.e. the whole range $[b_1, b_2)$ and not just $[b_1, b_3)$. This is because Alice cannot be sure that Bob overwrote what is in $[b_3, b_2)$ and she must ensure that any capabilities she may have inadvertently left there are scrubbed before returning to Charlie.

Informal cost analysis. With these details in mind, let's make sure that we can indeed witness a gain in performance. At first glance, since the new calling convention still clears the local stack frame upon return, it could appear as if the new calling convention only provides a minor constant factor performance improvement. However, in the calling convention by Skorstengaard et al. [129], the *full* stack space (even parts that will never be used) is cleared twice for each call. In our new proposed calling convention, each function only needs to clear its own stack frame once upon return.

Let us consider two concrete scenarios. First, assume that we are making n secure calls in sequence to untrusted components. Let us note m the size of the remaining unused stack space, and c the size of the stack frame that we currently use.

Skorstengaard et al.'s calling convention requires that we clear the whole unused stack space before each call, and finally, clear it again along with our own stack frame before returning. The cost of clearing is then:

$$n * m + m + c$$

In a typical scenario where c is small compared to m, the overall cost of clearing is quadratic: $\mathcal{O}(mn)$ (m is typically comparable to the overall available stack space, which counts in megabytes). In the improved calling convention, we only need to clear *our* stack frame before returning. We therefore only need to pay the small price of clearing c memory cells.

Now assume that we are making n nested secure calls, each call using a stack frame of size c.

Skorstengaard et al.'s calling convention now requires that we clear m before the first call and $m+c$ when returning, $(m-c)$ before the second call and m when returning, etc, i.e.:

$$2nm - n(n-1)c + c$$

Again, in a typical scenario where the portion of the stack actually used (nc) is small compared to the available stack space m, the cost remains a quadratic $\mathcal{O}(mn)$. With our calling convention, we only need to clear the individual stack frames, which amounts to an overall linear cost of nc. (Notice how this does not depend on the size of the available stack space.)

In summary, it seems like uninitialized capabilities solve the stack clearing requirement and associated performance issue of local capability revocation and the secure calling convention of Skorstengaard et al. [129, 131]. But security of the result relies on subtle arguments and invariants. Fortunately, in the next section, we'll see that we can build on Skorstengaard et al.'s approach for reasoning about capability machines and the guarantees they provide and prove security of local capability revocation and our new calling convention.

3.5 Program Logic

In order to reason about the behavior of programs running on the capability machine, we build a program logic on top of the machine operational semantics. The logic provides rules describing the execution of single machine instructions, and can then be used to establish a specification for a complete program running until the machine halts (or fails).

Specifications are written as separation logic triples, both in the case of manually written specifications for concrete programs (such as the macros of Section 3.6.9), and in the case of the "universal specification" which holds of arbitrary code by the Fundamental Theorem (see Section 3.6.8). Figure 3.7 shows specifications for some single machine instructions as well as for a program composed of several instructions (in this case, a simple macro). In a high-level language, a separation logic triple $\{P\}\,e\,\{Q\}$ provides a precondition P and postcondition Q for the execution of the expression e. However, in our setting, there is no direct equivalent of e since code executed by the machine is laid out in memory as mere integers that are then decoded into instructions. Instead, we use triples of the form $\{P\}\,\mu\,\{Q\}$, where μ denotes an *execution mode* as defined in Figure 3.1. Treating execution modes as expressions in this way makes our assembly language fit well into the Iris framework, which is more usually used with lambda calculi. A triple using the SingleStep execution mode specifies the behavior of a single instruction (the one currently pointed to by the

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc},p') \\ \mathrm{ValidSubseg}(p,b,e,z_1,z_2) \qquad \mathrm{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2\end{array}}{\begin{array}{l}\{\mathsf{pc} \mapsto (p_{pc},g_{pc},b_{pc},e_{pc},a_{pc}) \ast a_{pc} \mapsto_{p'} n \ast r \mapsto (p,g,b,e,a)\} \\ \quad \mathsf{SingleStep} \\ \left\{\begin{array}{l} v,\ \ v = \mathsf{Done\ Standby} \ast \mathsf{pc} \mapsto (p_{pc},g_{pc},b_{pc},e_{pc},a_{pc}+1) \\ \quad \ast\, a_{pc} \mapsto_{p'} n \ast r \mapsto (p,g,z_1,z_2,a) \end{array}\right\}\end{array}}$$

$$\frac{\begin{array}{c}\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc},p') \\ \mathrm{ValidStore}(p,b,e,a,p'',w) \qquad \mathrm{decode}(n) = \texttt{store}\ dst\ src\end{array}}{\begin{array}{l}\left\{\begin{array}{l} \mathsf{pc} \mapsto (p_{pc},g_{pc},b_{pc},e_{pc},a_{pc}) \ast a_{pc} \mapsto_{p'} n \ast dst \mapsto (p,g,b,e,a) \\ \ast\, src \mapsto w \ast a \mapsto_{p''} - \end{array}\right\} \\ \quad \mathsf{SingleStep} \\ \left\{\begin{array}{l} v,\ \ v = \mathsf{Done\ Standby} \ast \mathsf{pc} \mapsto (p_{pc},g_{pc},b_{pc},e_{pc},a_{pc}+1) \ast a_{pc} \mapsto_{p'} n \\ \quad \ast\, dst \mapsto (p,g,b,e,a) \ast src \mapsto w \ast a \mapsto_{p''} w \end{array}\right\}\end{array}}$$

$$\frac{\forall i \in [0,n),\ \mathrm{ValidPC}(p,b,e,a_i,p') \qquad n = |\mathrm{rclear\_instrs}\ l|}{\begin{array}{l}\left\{\begin{array}{l} \mathsf{pc} \mapsto (p,g,b,e,a_0) \ast \mathop{\LARGE\ast}\limits_{r \in l} r \mapsto - \ast \mathop{\LARGE\ast}\limits_{i \in [0,n)} a_i \mapsto_{p'} (\mathrm{rclear\_instrs}\ l)[i] \ \ast \\ \left(\begin{array}{l} \mathsf{pc} \mapsto (p,g,b,e,a_n) \ast \mathop{\LARGE\ast}\limits_{r \in l} r \mapsto 0 \ast \mathop{\LARGE\ast}\limits_{i \in [0,n)} a_i \mapsto_{p'} (\mathrm{rclear\_instrs}\ l)[i] \mathrel{-\!\!\ast} \\ \mathsf{wp\ Repeat\ SingleStep}\ \{Q\} \end{array}\right) \end{array}\right\} \\ \quad \mathsf{Repeat\ SingleStep} \\ \{Q\}\end{array}}$$

$$\begin{array}{lcl}
\mathrm{ValidPC}(p_{pc},b_{pc},e_{pc},a_{pc},p') & \triangleq & p_{pc} \preccurlyeq p' \wedge p' \in \{\mathrm{RX}, \mathrm{RWX}, \mathrm{RWLX}\} \\
 & & \wedge b_{pc} \leq a_{pc} < e_{pc} \\
\mathrm{ValidSubseg}(p,b,e,z_1,z_2) & \triangleq & b \leq z_1 \wedge 0 \leq z_2 \leq e \\
\mathrm{ValidStore}(p,b,e,a,p'',w) & \triangleq & \mathrm{RW} \preccurlyeq p \preccurlyeq p'' \wedge b \leq a < e \wedge \\
 & & \text{if } w = (\_, \mathrm{LOCAL}, \_, \_, \_) \\
 & & \quad \text{then } p \in \{\mathrm{RWLX}, \mathrm{RWL}\} \\
\mathrm{rclear\_instrs}\ l & \triangleq & \mathrm{map}\ (\lambda r.\ \mathrm{encode}(\texttt{move}\ r\ 0))\ l
\end{array}$$

Figure 3.7: Separation Logic specifications for the machine instructions `subseg` and `store` and for the `rclear` macro that sets a given list of registers to zero. Changes to the machine state are highlighted in red.

program counter). A triple using the Repeat SingleStep execution mode specifies a complete execution, starting from the instruction currently pointed to by the program counter, and continuing until the machine halts or fails.

We use Iris' standard definition of triples, which correspond to partial correctness: correctness does not entail termination. Finally, note that machine failure (e.g. failure to pass a capability check) is modeled explicitly. A failing program does not get stuck, instead, it reduces to a configuration with the Done Failed execution mode. A postcondition binds the execution mode at the end of the execution, allowing specifications to talk explicitly about failure or success.

As an additional subtlety, note that separation logic triples are not a primitive concept in Iris. Instead, they are defined as syntactic sugar on top of a weakest-precondition combinator

$$\{P\}\,\mu\,\{Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\,\mu\,\{Q\})$$

The triple $\{P\}\,\mu\,\{Q\}$ specifies that owning the resource P is sufficient to run the machine with mode μ and eventually obtain the postcondition Q. Furthermore, this fact is required to hold not only at the current point, but also to remain true indefinitely, using the Iris modality $\Box$ [see, e.g., 20, 77]. This "persistent" modality $\Box$ expresses that the proof of a triple may not rely on assumptions that hold now but may cease to hold in the future ("ephemeral assertions"). Instead, it must only rely on assumptions that remain true at any point in the execution of the system ("persistent assertions"), because we may want to invoke this specification at any later point.

Access to registers and memory is described using two separate points-to assertions. The assertion "$r \mapsto w$" asserts that register r currently contains the machine word w, and provides exclusive ownership over that register. The assertion "$a \mapsto_p w$" asserts that the memory location at address a currently contains the machine word w and provides ownership over that location. Furthermore, access to the location is restricted with permission p: for instance, if p is RO then it is not possible to modify the value stored at that location. More generally, when accessing a memory location with permission p using a capability with permission p', the permission of the capability must be included in the permission for the location, i.e. $p' \preccurlyeq p$.

The first two rules of Figure 3.7 show specifications for the `subseg` and `store` instructions. Their respective preconditions describe the subset of the machine state accessed by the instruction, and the postconditions describe the updated state after executing the instruction. For both specifications, the postcondition asserts that the execution mode after executing the instruction is Done Standby, meaning that the machine instruction always succeeds under the premises of the specification. The first rule states that if the program counter contains a capability pointing to a memory location a_{pc}, if that location contains an integer n which decodes into `subseg` $r\ z_1\ z_2$, if the register r contains a capability, and assuming that the program counter is valid (ValidPC(...)) and that z_1 and z_2 are valid new bounds (ValidSubseg(...)), then the machine successfully increments the program counter and restricts the capability held in register r with new bounds z_1 and z_2. Similarly, the second rule states that successfully executing the store instruction reads a word from the *src* register and writes it into the memory location pointed to by the capability in the *dst* register.

The specifications that appear in Figure 3.7 for subseg and store are in fact not the most general specifications for these instructions. They assume that some side-conditions hold and establish that the execution succeeds, making them useful for reasoning about the correctness of a concrete program. However, there are many ways in which instructions can fail: because of capability checks, but also, for example, because incrementing the program counter or performing address arithmetic can fail since we have finite memory. Our program logic thus also provides rules (not reproduced here) to reason about cases where executing an instruction fails. Furthermore, "most general" specifications covering all cases are also provided; these are useful not only as a proxy for deriving more specific rules, but also directly in the proof of the Fundamental Theorem (Theorem 10), for characterizing the behavior of arbitrary instructions that might or might not fail.

Our machine code does not have primitive mechanisms for structured control flow. Similarly, our program logic does not make assumptions about program control flow. Instead, programs composed of several instructions are specified in continuation-passing style: one proves a specification for a complete execution of the machine, starting at the beginning of the program, by assuming a specification for the continuation of the program, which is reached either through sequential instruction fetch, or through a jmp instruction.

The last rule of Figure 3.7 exemplifies such a specification for a program composed of several instructions; the rclear macro. This macro clears a number of registers by setting their contents to 0. It is parameterized by a list l of register names and its code consists of a sequence of instructions move r 0 for each register name r in l. We state rclear's specification as a triple using the Repeat SingleStep execution mode, meaning that the specification covers a full execution of the machine, and prove that starting before the execution of rclear, to reach any postcondition Q (describing the state of the machine at the very end of the execution) it is enough to prove that one can reach Q from the *continuation*, i.e. after rclear as been executed. In other words, the postcondition of rclear is given as the precondition of its continuation.

Concretely, the specification of rclear assumes that the body of the macro ("rclear_instrs l") is laid out contiguously in memory range $[a_0, a_n)$, while the program counter initially points to a_0. When the program counter eventually points to a_n, the address immediately after the macro instructions, then all the registers in l have been cleared and now contain 0. Importantly, notice that the specification for the continuation of rclear is given not as a separation logic triple, but directly in terms of the weakest-precondition combinator. Unlike triples, this specification is not required to be persistent (note the absence of $\Box$). Indeed, it only makes sense to invoke this specification once, at the point of the execution where the continuation is reached (i.e. when pc reaches (p, g, b, e, a_n)).

3.6 Logical Relation Model

Now that we have this program logic, we can explain the most important contribution of this paper: the formalization and proof of *capability safety*. This is the set of guarantees that the capability machine provides for untrusted code and it includes both general capability safety guarantees and guarantees that are specific to local and uninitialized capabilities.

While our program logic (Section 3.5) provides rules for concrete machine instructions which are useful to verify known concrete code, capability safety provides a universal contract that holds for unknown, arbitrary code. Thanks to the capability checks implemented by the capability machine, an arbitrary piece of code cannot behave completely arbitrarily: it is limited by the set of capabilities it has access to. Our logical relation model thus captures how we can reason about the interaction of known and unknown code, and in particular which guarantees one exactly gets from the revocation mechanism enabled by local and uninitialized capabilities.

For readability, we introduce the required machinery gradually, starting with a simple formulation of capability safety without support for revocation (Section 3.6.1). Next, we provide some intuitions on what needs to change for supporting revocation in Section 3.6.2. This motivates the need for a form of Kripke worlds with public/private transitions, and standard and custom resources, which we explain and apply in Sections 3.6.3 to 3.6.5. In Sections 3.6.6 to 3.6.7, we provide more technical details on how we combine Iris invariants and saved predicates with more traditional Kripke world-indexing. The Fundamental Theorem, which establishes that our machine indeed satisfies the capability safety formalized by the logical relation, is discussed in Section 3.6.8. Finally, we demonstrate reasoning about examples with revocation, by outlining a proof of the classic awkward example in Section 3.6.9.

3.6.1 A Version of the LR without Kripke Worlds/Local Capabilities

We begin with a formulation of capability safety without support for revocation. As such, we model capability safety of capabilities without a locality bit, reminiscent to the Cerise model presented in Chapter 2. Intuitively, the idea is to define the authority represented by a capability. The guarantees provided by the machine then amount to the fact that arbitrary code can never exceed the authority of the capabilities it has access to, or create capabilities with larger authority. To formalize these intuitions, we define a maximum bound on the authority of a capability using a notion of safety with respect to a set of registered invariants. A capability will be considered safe if it cannot be used in any way to break those invariants. This intuition is instantiated differently for different types of capabilities. For example, memory capabilities are safe if they only grant access to memory that is guaranteed to contain safe values by an invariant. Updating the memory with safe values must not break registered invariants. If the capability is executable, jumping to it with safe words in the registers must respect invariants and produce safe result values.

$$\boxed{\mathscr{E}(v)} \triangleq \forall reg,\ \mathscr{R}(reg) * \mathsf{pc} \mapsto v * \mathop{\ast}_{(r,w)\in reg, r\neq \mathsf{pc}} r \mapsto w \mathrel{-\!\!*} \mathsf{wp\ Repeat\ SingleStep} \left\{ \begin{array}{l} v, v = \mathsf{Done\ Halted} \Rightarrow \\ \exists reg', \mathop{\ast}_{(r,w)\in reg'} r \mapsto w \end{array} \right\}$$

$$\boxed{\mathscr{R}(reg)} \triangleq \mathop{\ast}_{(r,w)\in reg, r\neq \mathsf{pc}} \mathscr{V}(w)$$

$$\boxed{\mathscr{V}(w)} \left\{ \begin{array}{ll} \mathscr{V}(z) & \triangleq \top \\ \mathscr{V}(\mathrm{O}, -) & \triangleq \top \\ \mathscr{V}(\mathrm{E}, b, e, a) & \triangleq \square \triangleright \mathscr{E}(\mathrm{RX}, b, e, a) \\ \mathscr{V}(p, b, e, a) & \triangleq \mathop{\ast}_{a'\in[b,e)} \exists p', p \preccurlyeq p' \wedge \boxed{\exists w, a' \mapsto_{p'} w * \mathscr{V}(w)} \end{array} \right.$$

Figure 3.8: Safety without Revocation.

The reader may notice that this intuitive definition of safety is problematically circular. This is commonly referred to as the world circularity problem [6, 21]. Skorstengaard et al. [129] resolve it for their model using step-indexed Kripke logical relations [6, 21]. We define our logical relations model in Iris so that we can (1) replace the manual bookkeeping of step-indices with its built-in support for guarded recursion, (2) use Iris invariants for reasoning about shared state, and (3) take advantage of the Iris implementation in Coq and its associated interactive proof mode [83].

Formally, we define in Figure 3.8 three mutually recursive logical relations. The value relation $\mathscr{V}$: Word $\rightarrow$ *iProp* defines what it means for a word to be safe, the expression relation $\mathscr{E}$: Word $\rightarrow$ *iProp* expresses what it means for a program counter to be safe to execute and the relation $\mathscr{R}$: (RegName $\rightarrow$ Word) $\rightarrow$ *iProp* expresses that a register file is safe if all register values are safe.

We define safety of words $\mathscr{V}$ as a guarded fixed point: each recursive occurrence of $\mathscr{V}$ is either guarded by the so-called "later" modality $\triangleright$ or appears inside an Iris invariant, indicated by the boxed assertion, and thus Iris guarantees that $\mathscr{V}$ is well-defined. For space reasons, we will not explain the later modality or Iris invariants technically; readers who are unfamiliar with them may interpret $\triangleright P$ to mean that P holds after one step of execution and think of an Iris invariant as a property that remains valid at every step of execution.

We define the expression relation $\mathscr{E}$ as a program specification, expressed using the weakest-precondition combinator. Conceptually, the body of $\mathscr{E}$ can be read as a Hoare-triple (see Section 3.5), except that it is not required to be persistent. A word v is in the expression relation—i.e. it is safe to execute—if one can run the machine with v in the pc register, and safe values in the other registers, provided we temporarily give up ownership of the registers but we get it back afterwards. Note that we do not specify what happens if the machine runs into an error, but only the case where the machine halts gracefully. Now, since we are not requiring any interesting property to hold in the postcondition, it might seem like the definition of $\mathscr{E}$ is trivial and always true! This is not the case, however. A weakest-precondition assertion only holds within Iris if all Iris invariants are preserved at every step of the execution. This includes the Iris invariants mentioned in the definition of $\mathscr{V}$, as detailed next.

The value relation, $\mathcal{V}(w)$, defines what it means for a word to be safe. Intuitively, the definition expresses that a word is safe when it cannot be used to violate invariants. There are two modes of usage to consider: (1) read/write authority over an address, and (2) authority to jump to an enter capability. A capability (p, b, e, a) with a permission p other than E or O grants read/write authority over each address a within its range of authority. It is in the value relation, if for each a within $[b, e)$, there exists an Iris invariant (indicated by the boxed assertion) which owns the memory location and guarantees that it will always contain a safe value. Note that the invariant is allowed to hold a stronger permission $p' \succcurlyeq p$ (so that we can easily downgrade capabilities' permissions).

A capability with an enter (E) permission is a special case: it cannot be used directly to read values from memory, so we do not require safety of the values it points to. Instead, its safety only requires that the capability is safe to execute (by the expression relation) after changing its permission to RX (as happens when invoking an enter capability). Since this capability may be jumped to at any point of the execution, this fact needs to hold persistently, hence the "box" modality.

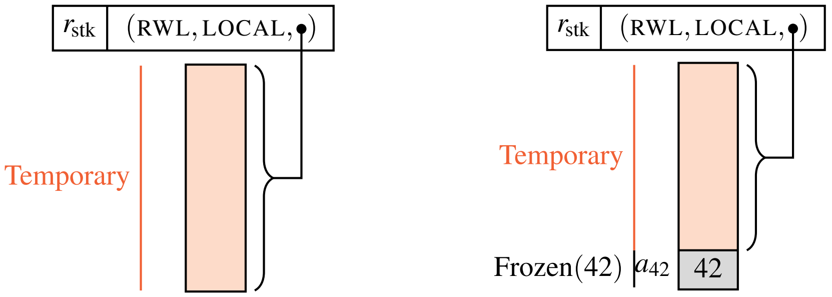
Interestingly, the safety of executable capabilities (RX or RWX) does not require any additional conditions. As we will see in Section 3.6.8, this is because we are formalizing capability safety: a property that holds for arbitrary code. As such, we could in principle allow the adversary to execute any capability it has read access to and in fact, all executable permissions in the lattice of Figure 3.2 also have read permission. In fact, even if we give an adversary read but not execute permission over some memory, we already cannot prevent them from executing the instructions anyway: as soon as they have writable and executable access to any block of memory $[b, e)$ elsewhere, they can simply copy the instructions into the range $[b, e)$ and jump to them there.

3.6.2 Reasoning about Revocation

The logical relation from the previous section is relatively easy to understand, but only captures a basic form of capability safety. In the following sections, we extend it to support local and uninitialized capabilities as well as revocation.

To understand what needs to change, we first take another look at scenario 3 from Section 3.3.3, using the illustrations in Figure 3.9. In this scenario, Alice receives a stack capability c_{stk} from Charlie in some register r_{stk}, as shown in Figure 3.9a. Alice knows that Charlie only has access to safe capabilities, so every address a in the range of c_{stk} must be owned by an invariant $\boxed{\exists w, a \mapsto_{p'} w * \mathcal{V}(w)}$. These invariants are depicted as Temporary in Figure 3.9, a term that we will explain in the next sections. This invariant means that any component in the system is allowed to change the content of the memory cell at a to any safe value w.

However, when Alice invokes Bob, the situation is different. Alice has now stored the value 42 in location a_{42} and expects Bob to not be able to change this value (see Figure 3.9b). To this end, Alice uses local capabilities to revoke Bob's read/write access to part of the stack and only allow him to modify the other parts. In other



(a) The stack Alice receives from Charlie.

(b) The stack Alice passes to Bob.

Figure 3.9: A scenario where a stack capability is passed in a register r_{stk} between different parties.

words, the invariant $\boxed{\exists w, a \mapsto_{p'} w * \mathcal{V}(w)}$ that used to govern the memory location a_{42} should no longer be active. Instead, it should be replaced by a new invariant expressing Alice's intention: the memory location should now be frozen: it should not be modifiable and only be allowed to contain 42, as shown in Figure 3.9b.

Replacing this old invariant with such a frozen invariant also means that capabilities that used to be safe are not safe any more. Specifically, a read/write memory capability c whose range includes a (e.g. Charlie's stack capability) will no longer be safe as the required invariant has been replaced. This observation makes a lot of sense: in this scenario, such a capability is really not safe anymore to pass to Bob, as he could use it to break the new frozen invariant.

In other words, reasoning about local capabilities and revocation requires two things that are impossible in the logical relation from Section 3.6.1. First, general Iris invariants cannot be deactivated (except temporarily during a single atomic step, but that's not what we need). Once they are defined, they remain active during the rest of the execution of the system. Second, the logical relation does not allow a capability to be safe at one moment but become unsafe later (when certain invariants have been revoked): the value relation is simply a predicate on words and if it is true, it remains true forever.

Moreover, we also need to ensure that an adversary does not deactivate an invariant, without reinstating it when they return. In other words, we need a more refined model, where invariants can be in different states and where safety can depend on these states. Moreover, we must be able to track precisely how these states evolve (to ensure that invariants are properly reinstated when necessary). This final point means that to define the refined model it is not enough simply to replace the general Iris invariants with so-called cancellable invariants. Instead we will parameterize our logical relation by an explicit notion of *world*, which will allow fine-grained control over invariant states.

3.6.3 Kripke Worlds to Track the State of Invariants

We change the signature of the value relation as follows: the safety of a word can now depend on a *world* representing the currently active invariants: $\mathcal{V} : \mathsf{WORLD} \rightarrow \mathrm{Word} \rightarrow iProp$.

Some readers may notice that our value relation now has the same signature as a step-indexed Kripke logical relation, but we hasten to point out that our worlds are much simpler than is typical in such settings. In earlier work, e.g., [8, 39, 129, 132], worlds track both invariant states and associated predicates (which are also world-indexed) on memory and are therefore recursively defined. Here instead, worlds track only the states of invariants and in Section 3.6.6, we will discuss how the associated predicates on memory are tracked using an Iris mechanism called saved predicates[75, 77].

Before we move on to the definition of worlds, there is a final important observation to make in the revocation scenario we discussed. As discussed, Alice revokes the old invariant for location a_{42} before invoking Bob and as discussed, this will break the safety of some capabilities. However, not all invariants can be revoked in this way and also, not all capabilities will be made unsafe by revoking an invariant. To understand this, consider that it is easy to control the local capabilities that an adversary has access to: they must reside in the registers or in memory that the adversary has previously received write-local access to. However, the same is not true for global capabilities: the adversary might have previously stored those in arbitrary memory and we have no way to revoke them. Since we can't revoke an adversary's access to global capabilities, it should not be possible to revoke invariants which their safety depends on. Conversely, global capabilities' safety should be able to survive the revocation of invariants like the one for a_{42}.

What this means is that we need to distinguish two kinds of invariants: (1) non-revocable ones, which global capabilities' safety may depend on, and (2) revocable ones, which global capabilities' safety must not depend on. Revoking the latter may affect the safety of local capabilities but not global capabilities. To formalize this, we follow previous work [39, 131] and distinguish public and private world updates. The former are those which cannot break safety of any capabilities (e.g. adding new invariants for previously unused memory) while the latter are updates which may break safety of local capabilities but not global capabilities (e.g. revoking invariants, adding new invariants for unused memory). If a world W' can be reached from W using public transitions alone, we call it a public future world ($W' \sqsupseteq^{pub} W$) and similarly for private transitions and private future worlds ($W' \sqsupseteq^{priv} W$). To securely enforce revocation, a calling convention must satisfy the following; during execution, a function may internally produce private future worlds, however, the overall change observed from the beginning of a call to its return is a public transition.

Our worlds assign to memory locations a logical state belonging to a small protocol tailored to talk about revocation. This "standard" protocol uses four possible states. A location can be either in the Temporary, Frozen, Permanent or Revoked state: The first two are revocable (consequently, global capabilities may not depend on them), the third is not (consequently, global capabilities can depend on them).

- The Temporary state represents the invariant that a location may only contain safe words, including local capabilities. This type of invariant is intended to cover memory locations in the stack as they are passed from caller to callee, which are allowed to contain local capabilities.

- The Permanent state represents the invariant that a location contains safe words, but only those whose safety will survive private updates, i.e., no local capabilities.

- The Revoked state corresponds to the result of revocation: a location that was previously Temporary or Frozen but got revoked. This means we know nothing about the contents of the memory at this location: conceptually, someone has taken control over the location, and needs to do some work to reinstantiate the invariants and restore safety of capabilities for it.

- Finally, the Frozen state asserts that we know the exact (not-necessarily-safe) value stored at the location, and it is not allowed to change. Frozen states are used for two purposes: (1) to keep a local stack frame frozen during a call to an adversary and (2) to freeze the uninitialized part of a capability. Indeed, locations in the uninitialized part of a U- capability will point to the same word right until they are written to. Whenever an uninitialized capability is purposely uninitialized (when passing it to an adversary), the Frozen state will allow us to remember the old, now unsafe value it still contains. If the word is never overwritten, then that knowledge can be used to reinstate the address to its previous Temporary state (see Section 3.6.7 for an example of such an update). Note that the Frozen state is a schema that describes a pattern of states, one for each possible mapping m. The transition from Frozen m_1 to Frozen m_2 must therefore go through the Temporary state, and is thus an overall private transition.

We call these states the *standard* states, StdStates.

Invariants represented by these standard states are collected in W^{std}, the first component of a world W. It is a partial map from addresses to standard states: $W^{std} : \mathrm{Addr} \hookrightarrow \mathrm{EX}(\mathsf{StdStates})$. Here EX refers to the Iris notion of an exclusive resource algebra—readers who are unfamiliar with Iris can ignore it. This map only tracks the states of shared resources, i.e. those that safe capabilities can range over. The shared resources are exactly those that are associated to the standard behaviour. In Section 3.6.5, we will explain a second component of W, which collects other, custom invariants. Such custom invariants are never directly addressable by shared (i.e. safe) capabilities, but they are necessary for modeling advanced examples (closures with non-trivial local state), see, e.g., Section 3.6.9.

$$\boxed{\mathcal{E}(W)(v)} \triangleq \forall reg,\ \mathscr{R}(W)(reg) * \mathit{sharedResources}(W) * \mathit{stsCollection}(W) * \\ \mathsf{pc} \mapsto v * \mathop{\Large *}_{(r,w)\in reg/\mathsf{pc}} r \mapsto w \mathrel{-\!\!*} \\ \mathsf{wp\ Repeat\ SingleStep} \left\{ \begin{array}{l} v, v = \mathsf{Done\ Halted} \rightarrow \\ \exists W'\, reg',\ W' \sqsupseteq^{priv} W \\ * \mathit{sharedResources}(W') \\ * \mathit{stsCollection}(W') \\ * \mathop{\Large *}_{(r,w)\in reg'} r \mapsto w \end{array} \right\}$$

$$\boxed{\mathscr{R}(W)(reg)} \triangleq \mathop{\Large *}_{(r,w)\in reg/\mathsf{pc}} \mathscr{V}(W)(w)$$

$$\boxed{\mathscr{V}(W)(w)} \left\{ \begin{array}{lcl} \mathscr{V}(W)(z), \mathscr{V}(W)(\mathrm{O}, -) & \triangleq & \top \\ \mathscr{V}(W)(\mathrm{E}, g, b, e, a) & \triangleq & \square\, \forall W' \sqsupseteq^{g} W, \\ & & \quad \triangleright\, \mathcal{E}(W')(\mathrm{RX}, g, b, e, a) \\ \mathscr{V}(W)(p, g, b, e, a) & \triangleq & \mathop{\Large *}_{a'\in[b,e)} \exists p', p \preccurlyeq p' \wedge \mathit{rel}(a', p', \mathscr{V}) \\ & & \wedge \left\{ \begin{array}{ll} \mathscr{S}^{u}(W)(a', g, p, a) & \text{if } p = \mathrm{U}\text{-} \\ \mathscr{S}(W)(a', g, p) & \text{otherwise} \end{array} \right. \end{array} \right.$$

$$\boxed{\text{State relation}}$$

$$\mathscr{S}(W)(a, g, p) \triangleq \left\{ \begin{array}{ll} W^{std}(a) \in \{\mathrm{Temporary}, \mathrm{Permanent}\} & \text{if } \neg\text{write-local}(p) \\ & \quad \wedge\ g = \mathrm{LOCAL} \\ W^{std}(a) = \mathrm{Temporary} & \text{if write-local}(p) \\ & \quad \wedge\ g = \mathrm{LOCAL} \\ W^{std}(a) = \mathrm{Permanent} & \text{if } g = \mathrm{GLOBAL} \end{array} \right.$$

$$\mathscr{S}^{u}(W)(a, g, p, mid) \triangleq \left\{ \begin{array}{ll} \mathscr{S}(W)(a, g, p) & \\ \vee\ \exists w, W^{std}(a) = \mathrm{Frozen}\{[a := w]\} & \text{if } \ a \geq mid \\ & \quad \wedge\ g = \mathrm{LOCAL} \\ \mathscr{S}(W)(a, g, p) & \text{otherwise} \end{array} \right.$$

Figure 3.10: Safety with Revocation. Differences with Figure 3.8 are highlighted in blue.

3.6.4 The Logical Relation with Support for Revocation

Let us now take a look at Figure 3.10 and see how the logical relation is updated to use these worlds. The differences with the LR from Section 3.6.1 are highlighted in blue. Apart from the addition of world parameters W, the changes are concentrated around the validity of a read-write capability. Instead of requiring the presence of an Iris invariant, that condition now formalizes the intuitive idea mentioned above: $rel(a, p', \mathcal{V})$ associates a memory invariant (namely $\mathcal{V}$ itself) to address a using saved predicates, whereas $\mathcal{S}$ and $\mathcal{S}^u$ associate the address a to its state. More precisely, the state relation $\mathcal{S}(W)(a, g, p)$ looks at the locality g and the permission p, and requires W to contain the appropriate state in W^{std}. The uninitialized state relation $\mathcal{S}^u(W)(a, g, p, mid)$ does the same but for U- permissions, for which the required state also depends on the boundary mid between the initialized and uninitialized part (i.e. the current address a of the uninitialized capability). The resource $rel(a, p', \mathcal{V})$ will be discussed later in Section 3.6.6.

We highlight what the states are for some interesting cases of safe capabilities:

- A capability with a RWLX permission (which must be itself local) is in $\mathcal{V}(W)$ if each address within its range of authority is in a Temporary state of W^{std} (so the address can be used to store local capabilities).
- A capability in $\mathcal{V}(W)$ with a URWLX permission and LOCAL locality, currently pointing to the address mid, has all addresses $a < mid$ in a Temporary state, whereas it has all addresses $a \geq mid$ either in a Temporary state or Frozen at some hidden word w.
- Global capabilities in $\mathcal{V}(W)$ have all addresses in their range of authority in a Permanent state, regardless of their permission.

In addition, the value relation for enter capabilities now quantify over future worlds $W' \sqsupseteq^{g} W$. For g local resp. global, this means that the execution of the capability must hold in arbitrary public resp. private future worlds. This quantification makes sure that global enter capabilities remain safe when temporary invariants are revoked, and enforces that invariants are properly reinstated. Intuitively, it means that a global enter capability, typically referring to that of a function pointer, can be safely invoked in any private future world (or in other words, at any time during execution), while invoking a local enter capability, i.e. that of a return pointer, indicates the end of a call, and can thus only safely be invoked in a public future world.

Finally, the new *sharedResources* and *stsCollection* assertions in the figure are used to ensure that shared memory actually satisfies the memory invariants, which have been registered using saved predicates, during execution.

Concretely, $sharedResources(W)$ and $stsCollection(W)$ are *non-duplicable* Iris resources that describe the current global state of standard states and memory invariants at world W, contrasted against $\mathcal{V}(W)(w)$ which defines the *persistent* knowledge that w is valid in the world W. All three are parametrized by a world state W. As such,

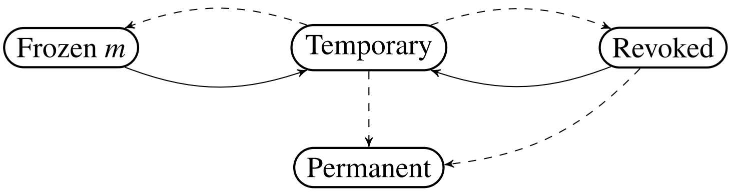


Figure 3.11: Standard State Transition System. Full lines indicate public transitions, dashed lines indicate private transitions. Public transitions are also private.

the following three assertions can hold at the same time:

$$\mathit{sharedResources}(W_1) * \mathit{stsCollection}(W_2) * \mathcal{V}(W_3)(w)$$

where W_1, W_2 and W_3 are different worlds. A recurring proof obligation, is to update the world state of $\mathit{stsCollection}(W)$ and $\mathit{sharedResources}(W)$ (typically as a result of some physical change to the machine), and consequently having to reestablish validity of relevant words, so that it matches the new world state.

The formal details behind *sharedResources* and *stsCollection* are a bit technical and will be discussed further in Section 3.6.6.

3.6.5 World Updates and Monotonicity

Now let us reconsider our worlds and future world relations in more detail. As already mentioned, temporary invariants may be revoked to obtain a private future world and fresh invariants over unused memory may be added to obtain a public future world. Actually, those are not the only types of updates allowed; Figure 3.11 depicts the allowed transitions between standard states. Dashed lines in the figure indicate private updates and full lines indicate public ones. One can observe that making a frozen or revoked location temporary is a public update. Indeed, doing so can never make safe capabilities unsafe; the only frozen locations that a safe capability may depend on, are those that belong to the uninitialized part of a U- capability. These locations can become Temporary without breaking the capability's validity. In contrast, changing the state of a temporary location is a private update, because it may break safety of capabilities depending on it.

We can now give the full definition of WORLD. In addition to the component W^{std} which we have already seen, it contains a second component W^{cus}. This component contains custom state transition systems, whose states can be associated with arbitrary Iris predicates. Such custom invariants are often needed for examples that involve closures with some private state evolving according to a certain ad hoc protocol, like in the example presented in Section 3.6.9. We remark that the definition of the value relation does not depend on the custom states, and only standard states are needed to prove the fundamental theorem of logical relations (Section 3.6.8). However, through its quantification over future worlds, the value relation enforces that custom states evolve according to their public and private future world relations.

W^{cus} refers to a mapping from region names $rn \in \mathsf{RNames}$ to custom state transition systems with private and public transitions, and their current state $\sigma \in \mathsf{CState}$. Formally, this is represented in two maps, W^{cus}_{sta} and W^{cus}_{rel}, where the former maps region names to current states, and the latter maps region names to private and public transitions. A WORLD is simply a pair of W^{std} and W^{cus}, and $\mathit{stsCollection}(W)$ denotes the full ownership, what we call the authoritative view, of W^{std}, W^{cus}_{sta} and W^{cus}_{rel}.

We distinguish between two future world relations. We call W' a public future world of W, $W' \sqsupseteq^{pub} W$, if each state in W' is either fresh, or publicly reachable from its state in W. A state is publicly reachable by a sequence of public transitions. Conversely, we call W' a private future world of W, $W' \sqsupseteq^{priv} W$, if each state in W' is either fresh, or privately reachable from its state in W. A state is privately reachable by a (possibly interwoven) sequence of private and public transitions. Note that a public future world is also a private future world.

Since the validity of global capabilities depend only on Permanent regions, it is easily proved monotone with regards to private future worlds, in which Permanent regions can never be affected.

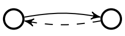
Lemma 10. *Let w be a word that is not Local. Then*

$$W' \sqsupseteq^{priv} W \to \mathcal{V}(W)(w) \mathrel{-\!\!*} \mathcal{V}(W')(w)$$

On the other hand, local capabilities may depend on Temporary regions, and are thus only monotone with regards to public future worlds, giving rise to the following general monotonicity property.

Lemma 11. $W' \sqsupseteq^{pub} W \to \mathcal{V}(W)(w) \mathrel{-\!\!*} \mathcal{V}(W')(w)$

In the remainder of this section, we discuss how individual states of $\mathit{stsCollection}(W)$ may be reasoned about. This part of the model is a bit technical and may be skipped on a first reading. The key idea, is that Iris enables us to reason locally about individual states in the global world state. To facilitate local reasoning about standard states, we introduce *fragmental views* of individual resources, in the form of points-to predicates: $a \overset{std}{\longmapsto} \rho$ denotes the fragmental view of the standard state of address a, and $rn \overset{cus}{\longmapsto} \sigma$ denotes the fragmental view of the state of the custom state transition system rn. These are non-duplicable, while the fragmental view for custom state transitions $rn \overset{rel}{\longmapsto} (R_{pub}, R_{priv})$ is persistent. We occasionally abuse notation and write

$$rn \overset{rel}{\longmapsto}$$ 

to denote the fragmental view of the public and private transitions of a custom state transition system named rn.

Owning both a fragmental and authoritative view implies that the state of the fragment is part of the global state. We write $W(a) = \rho$ as shorthand for $W^{std}(a) = \rho$.

Lemma 12. $\mathit{stsCollection}(W) * a \overset{std}{\longmapsto} \rho \mathrel{-\!\!*} W(a) = \rho$

$$\begin{array}{rcl}
\gamma s_{std}, \gamma s_{cus}, \gamma r_{cus} & \in & \mathsf{GName} \\
W^{std} & : & \mathsf{Addr} \hookrightarrow \textsc{Ex}(\mathsf{StdStates}) \\
W^{cus}_{sta} & : & \mathsf{RNames} \hookrightarrow \textsc{Ex}(\mathsf{CState}) \\
W^{cus}_{rel} & : & \mathsf{RNames} \hookrightarrow \textsc{Ag}(2^{CState} \times 2^{CState}) \\
\\
stsCollection(W^{std}, W^{cus}) & \triangleq & \boxed{\bullet W^{std}}^{\gamma s_{std}} * \boxed{\bullet W^{cus}_{sta}}^{\gamma s_{cus}} * \boxed{\bullet W^{cus}_{rel}}^{\gamma r_{cus}} \\
\\
a \xmapsto{std} \rho & \triangleq & \boxed{\circ [a := \textsc{Ex}(\rho)]}^{\gamma s_{std}} \\
rn \xmapsto{cus} \sigma & \triangleq & \boxed{\circ [rn := \textsc{Ex}(\sigma)]}^{\gamma s_{cus}} \\
rn \xmapsto{rel} (R_{pub}, R_{priv}) & \triangleq & \boxed{\circ [rn := \textsc{Ag}(R_{pub}, R_{priv})]}^{\gamma r_{cus}}
\end{array}$$

Figure 3.12: Collection of State Transition Systems

Likewise, a fragment allows us to update its state in $stsCollection(W)$. We write $W[a := \rho']$ as shorthand for $(W^{std}[a := \rho'], W^{cus})$. Since the following lemma updates Iris ghost state, we use the 'update' modality from iris $\Rrightarrow\!\!*$, readers unfamiliar with Iris may safely ignore it, and regard it as a typical separation logic wand.

Lemma 13. $stsCollection(W) * a \xmapsto{std} \rho \Rrightarrow\!\!* stsCollection(W[a := \rho']) * a \xmapsto{std} \rho'$

Lemma 13 updates $stsCollection(W)$, without updating $sharedResources(W)$ or any previously stated validity assertions. A recurring proof obligation will thus be to reestablish validity in the new updated world state. We will describe how this can be done in Section 3.6.7.

The above lemmas sufficiently display the core features of $stsCollection(W)$ and its fragments. Nevertheless, readers familiar with Iris might be interested in the underlying technical details. Throughout the ensuing explanations, we will first introduce the main ideas through somewhat abstract definitions, and then follow up with the underlying Iris implementation.

When defining $stsCollection(W)$, we introduce three globally named authoritative resource algebras; an exclusive map for standard and custom states respectively, and an agreement map for custom transitions. Figure 3.12 formally defines all the aforementioned resources.

3.6.6 Linking Worlds to Memory

So far, we've defined worlds, explained how the logical relation depends on the invariants in a world and how worlds are allowed to evolve over time. What is still missing in the story is mapping invariants to requirements on memory contents and ensuring that those requirements are satisfied at runtime. This section is quite technical, and makes use of sophisticated Iris machinery like stored predicates.

The core idea is to define an authoritative mapping from addresses to memory invariants, where the authoritative view, $sharedResources(W)$, instantiates each in-

$$\begin{aligned}
\text{monoReq}(W,\phi,v,\sqsupseteq) &\triangleq \Box\, \forall W', W' \sqsupseteq W \to \phi(W,v) \mathrel{-\!\!*} \phi(W',v) \\
\text{permR}(a,p,W,\phi) &\triangleq \exists v, a \mapsto_p v * \triangleright\ \phi(W,v) * \text{monoReq}(W,\phi,v,\sqsupseteq^{\text{priv}}) \\
\text{tempR}(a,p,W,\phi) &\triangleq \exists v, a \mapsto_p v * \triangleright\ \phi(W,v) \\
&\qquad * \begin{cases} \text{monoReq}(W,\phi,v,\sqsupseteq^{\text{pub}}) & \text{if write-local}(p) \\ \text{monoReq}(W,\phi,v,\sqsupseteq^{\text{priv}}) & \text{o/w} \end{cases} \\
\text{frozenR}(a,p,m,M^{std}) &\triangleq a \mapsto_p m(a) * \forall a \in dom(m), M^{std}(a) = \text{Frozen m}
\end{aligned}$$

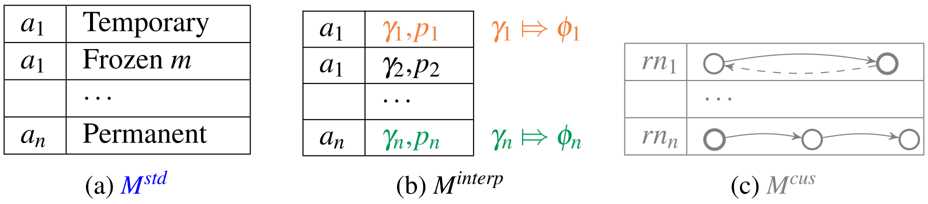
Figure 3.13: Standard Resources.

variant according to some side conditions, while the fragmental view, $rel(a,p,\phi)$ describe knowledge that an address a is associated to a particular invariant ϕ. The standard state of an address, determined via the standard state fragment presented in the previous section, is used to interpret each memory invariant accordingly.

Unlike Iris invariants, which hold at every step of execution, and which can only be invalidated for one atomic step, the authoritative view of the mapping is explicitly carried around, and may be invalidated during multiple steps of execution. Its integrity is instead enforced by the logical relation. Recall that the logical relation is used to reason about unknown code. More precisely, the expression relation yields a weakest precondition statement that describes the safe execution of arbitrary code. However, in order to apply that weakest precondition, certain side conditions must be established, among which are $stsCollection(W)$ and $sharedResources(W)$ for some world W. Likewise, since validity of the return capability is defined in terms of the expression relation, $stsCollection(W')$ and $sharedResources(W')$ is assumed to hold upon invoking the return capability, for some future world W'. Thus, the expression relation is used whenever control is changed between the trusted and untrusted code, and the invariant mapping is guaranteed to hold at the beginning of a function call, and upon the return the its caller. Finally, the postcondition given by the expression relation guarantees that the invariant mapping holds at the final program completion.

We will gradually build up to the full definition of the memory invariant mapping. First, in Figure 3.13, we define the resource interpretation of each standard state: standard shared resource invariants. The role of each interpretation is to map an address to the requirement on the location's contents that the associated invariant in its current state represents.

A *permanent* resource invariant for some address a and permission p contains the ownership of a points-to predicate for the address a. It states that some predicate ϕ holds at the current state of a, say v, and at some world W. Crucially, this ϕ holds *invariantly*, that is in any private future world of W (when applied to that same v). On the other hand, local capabilities are allowed to depend on revocable invariants, so a *temporary* invariant only requires ϕ to be monotone with regards to public future worlds and is not required to be able to survive private world updates.



a_1	Temporary
a_1	Frozen m
	$\cdots$
a_n	Permanent

(a) M^{std}

a_1	γ_1, p_1	$\gamma_1 \Mapsto \phi_1$
a_1	γ_2, p_2	
	$\cdots$	
a_n	γ_n, p_n	$\gamma_n \Mapsto \phi_n$

(b) M^{interp}

(c) M^{cus}

Figure 3.14: Abstract Machine.

$$
\begin{aligned}
srMap(W, M^{interp}, M^{std}) \quad &\triangleq \quad \mathop{\Large\ast}_{(a,(\gamma,p))\in M^{interp}} \exists \rho\ \phi, \gamma \Mapsto \phi * M^{std}(a) = \rho * a \xmapsto{std} \rho \\
&\qquad * \begin{cases} \mathrm{permR}(a, p, W, \phi) & \text{if } \rho = \text{Permanent} \\ \mathrm{tempR}(a, p, W, \phi) & \text{if } \rho = \text{Temporary} \\ \mathrm{frozenR}(a, p, m, \mathrm{M}^{std}) & \text{if } \rho = \text{Frozen } m \\ \top & \text{if } \rho = \text{Revoked} \end{cases} \\[1em]
sharedResources(W) \quad &\triangleq \quad \exists M^{interp}\ M^{std}, \\
&\qquad \cdots * \ \textit{authoritative ownership of } M^{interp} \ * \cdots \\
&\qquad * \ srMap(W, M^{interp}, M^{std})
\end{aligned}
$$

Figure 3.15: Standard Shared Resources Map

Finally, a *frozen* resource invariant Frozen m is parametrized by a memory segment: a partial map $m : \text{Addr} \hookrightarrow \text{Word}$ from addresses to specific words. Note that these words do not need to satisfy any invariant ϕ or be themselves safe in any way. The *Frozen* state imposes two requirements for addresses $a \in \text{dom}(m)$: that they point to the associated word, i.e. they cannot change until the state changes from *Frozen* to *Temporary*, and that each address in the map m are also frozen to the same map m (see below for more details on M^{std}). This additional requirement ensures that if one of the invariants for an address in $\text{dom}(m)$ is revoked, all other ones must be revoked along and it allows us to think of the addresses in m as being frozen as a block, rather than individually.

So, now we have defined maps that keep track of the standard state of shared resources, the region state of custom resources, and their associated state transition systems. We have also defined the meaning of temporary, permanent, and frozen resource invariants as requirements on memory. What remains is to connect the two: mapping a given world to the requirement on memory that its invariants represent. This connection is made using a few non-trivial pieces of Iris machinery, and as such we begin by presenting a rough sketch of what is going on.

Technically, the connection is made in the predicates $rel(a, p, \phi)$, $stsCollection(W)$ and $sharedResources(W)$ that appear in Figure 3.10. These three predicates are all defined as requirements on what we can think of as an *instrumented machine state*. This instrumented machine state consists of three parts, depicted in Figure 3.14. The

$$\begin{array}{rcl} \mathit{sharedResourcesOpen}(W,S) & \triangleq & \exists M^{interp}\, M^{std}, \\ & & \cdots * \textit{ authoritative ownership of } M^{interp} * \cdots \\ & & \quad * \mathit{srMap}(W, M^{interp} \backslash S, M^{std}) \end{array}$$

where S is a set of addresses, and $M^{interp} \backslash S$ removes S from the domain of M^{interp}.

Figure 3.16: Open Standard Shared Resources

most important part is M^{interp}, which associates each address a_i with a predicate ϕ_i and a permission p. To simplify the definition of M^{interp}, we do not associate a_i directly to ϕ_i, but indirectly through an Iris saved predicate $\gamma_i \Mapsto \phi_i$ for some i. The permission represents the permission of the first allocated capability with authority over a_i, in other words an upper bound on the permission of all capabilities that contain a_i in their range of authority. The predicate $\phi_i : \mathsf{WORLD} \times \mathsf{Word} \to \mathit{iProp}$ represents the predicate that is currently enforced on values stored in memory at address a_i. It is interesting to note, that the ϕ_i in question will most often be the value relation $\mathcal{V}$. However, given the circular nature of the value relation, and the invariants imposed over memory governed by a valid capability, the use of saved predicates over a generalized ϕ_i enables a well-formed definition.

Additionally, the instrumented machine state contains two other pieces of logical states, M^{std} and M^{cus}, describing the current world W and its two parts W^{std} and W^{cus}. The predicates $\mathit{stsCollection}(W)$ and $\mathit{sharedResources}(W)$ from Figure 3.10 impose requirements on this authoritative world. As we have seen in Section 3.6.5, $\mathit{stsCollection}(W)$ restricts the authoritative copy of the current world to correspond exactly to W. Meanwhile, $\mathit{sharedResources}(W)$ makes the connection for every address a between three things: the actual word w in the capability machine's memory at a, the predicate ϕ registered for a in M^{interp} and the standard state ρ for a in M^{std}. It requires that ϕ satisfies the word w at the world W, in the appropriate way as defined in Figure 3.13 for the state ρ.

Let's now formally outline how these connections are established. Figure 3.15 abstractly describes the final piece of the instrumented machine state, namely $\mathit{sharedResources}(W)$. It claims ownership over the authoritative view of the existentially quantified M^{interp}, and declares an instance of the srMap predicate. This predicate unfolds M^{interp}, determines the saved predicate of each address a, looks up the appropriate standard state ρ as declared in M^{std}, asserts ownership over the exclusive fragment $a \xmapsto{std} \rho$, and asserts the appropriate standard resources predicate accordingly.

We highlight that the standard resource is instantiated using W, rather than M^{std}. Indeed, W^{std} and M^{std} are not formally required to be equal. As a result, memory invariants are instantiated on a world that is potentially out of sync with the actual collection of standard states. In Section 3.6.7, we will see how this is a useful feature of the instrumented machine state.

The $rel(a,p,\phi)$ predicate, used in Figure 3.10, asserts the persistent knowledge that address a is in the domain of M^{interp}, associated with permission p, and the saved predicate ϕ. The *rel* predicate is, in essence, a fragment of the authoritative view as defined by the *sharedResources* predicate. As such, *rel* can be used to extract resources from the instrumented machine state. Since the extracted resources are non-duplicable, the resulting *sharedResources* predicate is opened (Figure 3.16), resulting in a mapping in which the opened resources are invalidated.

The following two lemmas let us to claim ownership over resources in *sharedResources*, allowing us to potentially invalidate their invariants, and later on yield ownership over those resources by reestablishing their invariants, thus closing *sharedResources*. Let W_1 denote the current standard state mapping, and W_2 the world at which each invariant is instantiated to.

Lemma 14 (Open Region)**.**

$$\begin{array}{l} a \notin S \rightarrow \\ stsCollection(W_1) * sharedResourcesOpen(W_2, S) * rel(a,p,\phi) \mathrel{-\!\!*} \\ \exists \rho, a \overset{std}{\longmapsto} \rho * stsCollection(W_1) * sharedResourcesOpen(W_2, S \uplus \{a\}) \\ \quad * \begin{cases} \mathrm{permR}(a,p,W,\phi) & \textit{if } \rho = \mathrm{Permanent} \\ \mathrm{tempR}(a,p,W,\phi) & \textit{if } \rho = \mathrm{Temporary} \\ \mathrm{frozenR}(a,p,m,W_1^{std}) & \textit{if } \rho = \mathrm{Frozen}\ m \\ \top & \textit{if } \rho = \mathrm{Revoked} \end{cases} \end{array}$$

The assumption that $a \notin S$, and the resulting $sharedResourcesOpen(W_2, S \uplus \{a\})$, guarantees that a cannot be extracted twice. Once extracted, it is possible to invalidate the memory invariant of a. However, in order to get the closed *region* predicate back, it is necessary to reestablish the memory invariants of address a.

Lemma 15 (Close Region)**.**

$$\begin{array}{l} stsCollection(W_1) * sharedResourcesOpen(W_2, S \uplus \{a\}) * rel(a,p,\phi) * a \overset{std}{\longmapsto} \rho \\ \quad * \begin{cases} \mathrm{permR}(a,p,W,\phi) & \textit{if } \rho = \mathrm{Permanent} \\ \mathrm{tempR}(a,p,W,\phi) & \textit{if } \rho = \mathrm{Temporary} \\ \mathrm{frozenR}(a,p,m,W_1^{std}) & \textit{if } \rho = \mathrm{Frozen}\ m \\ \top & \textit{if } \rho = \mathrm{Revoked} \end{cases} \\ \mathrel{-\!\!*} stsCollection(W_1) * sharedResourcesOpen(W_2, S) \end{array}$$

The above lemmas are useful for accessing resource invariants, and use them to reason about physical machine changes whenever those resources include memory points-to predicates. However, they do not describe changes to the instrumented machine state, in other words, changes to W_2 or W_1. In the following section, we investigate the necessary requirements for updating $sharedResources(W_2)$ and $stsCollection(W_1)$, and how such complex manipulations are made more tractable.

$$
\begin{aligned}
\gamma_h &\in \mathsf{GName} \\
M^{interp} &: \mathsf{Addr} \hookrightarrow \mathrm{AG}(\mathsf{GName} \times \mathsf{Perm}) \\
\\
sharedResources(W) &\triangleq \exists M^{interp}\ M^{std}, dom(W^{std}) = dom(M^{interp}) \\
&\qquad\qquad\qquad\qquad\qquad\quad = dom(M^{std}) \\
&\quad * \boxed{\bullet M^{interp}}^{\gamma_h} \\
&\quad * srMap(W, M^{interp}, M^{std}) \\
\\
sharedResourcesOpen(W,S) &\triangleq \exists M^{interp}\ M^{std}, dom(W^{std}) = dom(M^{interp}) \\
&\qquad\qquad\qquad\qquad\qquad\quad = dom(M^{std}) \\
&\quad * \boxed{\bullet M^{interp}}^{\gamma_h} \\
&\quad * srMap(W, M^{interp} \backslash S, M^{std}) \\
\\
rel(a,p,\phi) &\triangleq \exists \gamma, \gamma \Mapsto \phi * \boxed{\circ [a := \mathrm{AG}(\gamma, p)]}^{\gamma_h}
\end{aligned}
$$

Figure 3.17: Standard Shared Resources

As in the previous section, the high level descriptions of *sharedResources* are sufficient for understanding the role and utility of the instrumented machine state, however, the Iris reader may want to see its formal definition. Figure 3.17 describes the global names and resource algebras behind *sharedResources* and *rel*.

3.6.7 Changing the Instrumented Machine State

When reasoning about the execution of a function, the instrumented machine state undergoes changes both public and private. These changes often follow specific patterns related to the intended interpretation of standard states. Figure 3.18 defines a collection of useful functions over worlds: revoke(W^{std}) sets all Temporary states to Revoked, freeze(W^{std}, m) sets all addresses in the domain of m to Frozen m, uninitialize(W^{std}, m) sets all addresses in m to singleton Frozen states, and reinstate(W^{std}, S) sets all Revoked addresses in S to Temporary. The first three define private changes, while the latter defines a public change. Finally, updated(W^{std}, S, ρ) sets all addresses in S to ρ, and is used for specific fine-grained changes that does not fall within the more general transformations.

Updating the instrumented machine state is non-trivial, as it changes the general interpretation of memory invariants. Throughout this section, we will refer to current collection of standard state mapping as W_1, and the world at which each invariant is instantiated to as W_2. On the one hand, updating *stsCollection*(W_1) means the interpretation of a standard resource changes. For instance, making a Revoked state Temporary means we must now establish the tempR standard resource interpretation for that address. On the other hand, updating *sharedResources*(W_2) means that each memory invariant in the standard resources map must now be instantiated to the new

$$\text{revoke}(W^{std}) = \lambda a, \begin{cases} \text{Revoked} & \text{if } W^{std}(a) = \text{Temporary} \\ W^{std}(a) & \text{otherwise} \end{cases}$$

$$\text{freeze}(W^{std}, m) = \lambda a, \begin{cases} \text{Frozen } m & \text{if } a \in dom(m) \\ W^{std}(a) & \text{otherwise} \end{cases}$$

$$\text{uninitialize}(W^{std}, m) = \lambda a, \begin{cases} \mathit{Frozen}\ \{[a := m(a)]\} & \text{if } a \in dom(m) \\ W^{std}(a) & \text{otherwise} \end{cases}$$

$$\text{reinstate}(W^{std}, S) = \lambda a, \begin{cases} \text{Temporary} & \text{if } a \in S \\ W^{std}(a) & \text{otherwise} \end{cases}$$

$$\text{updated}(W^{std}, S, \rho) = \lambda a, \begin{cases} \rho & \text{if } a \in S \\ W^{std}(a) & \text{otherwise} \end{cases}$$

Where m denotes partial maps from addresses to standard states, and S denotes sets of addresses. We will overload the notation and write $\text{revoke}(W)$ *to denote* $(\text{revoke}(W^{std}), W^{cus})$ *(and similarly for remaining functions).*

Below is a list of useful world relation properties of above functions.

$$\text{revoke}(W) \sqsupseteq^{priv} W \tag{3.1}$$

$$\text{freeze}(W, m) \sqsupseteq^{priv} W \tag{3.2}$$

$$\text{uninitialize}(W, m) \sqsupseteq^{priv} W \tag{3.3}$$

$$\text{reinstate}(W, S) \sqsupseteq^{pub} W \tag{3.4}$$

$$W' \sqsupseteq^{priv} W \rightarrow \text{revoke}(W') \sqsupseteq^{priv} \text{revoke}(W) \tag{3.5}$$

Figure 3.18: Some Useful Functions over Worlds

updated world. This is particularly interesting for all the permanent and temporary resources, which each satisfy their own distinct monotonicity property.

Both temporary and permanent standard resources are monotone with regards to public future worlds. As such, public changes to $sharedResources(W_2)$ require no side conditions on the collection of standard states, and we can prove the following public update lemma.

Lemma 16. *If* $dom(W_2^{std}) = dom(W_2'^{std})$ *and* $W_2' \sqsupseteq^{pub} W_2$*, then* $sharedResources(W_2) \mathrel{-\!\!*} sharedResources(W_2')$

Proof. By unfolding $sharedResources(W_2)$, we observe that the changed parameter is restricted to the world at which invariants to the *srMap* are instantiated: neither the standard states in M^{std}, nor the content of M^{interp} are affected.

The proof proceeds pointwise over each a to (p, γ) mapping in M^{interp}. The saved predicate $a \Mapsto \phi$ and standard state predicate $a \overset{std}{\longmapsto} \rho$ are framed away, and we are left with the standard resource of a. We then proceed by cases:

- Case: $\rho = \text{Permanent}$:
  To show: $\text{permR}(a, p, W_2, \phi) \mathrel{-\!\!*} \text{permR}(a, p, W_2', \phi)$.

  Unfolding permR, we frame away the points to predicate, and are left with two proof obligations, namely:

  1. $\triangleright \phi(W_2, w) \mathrel{-\!\!*} \triangleright \phi(W_2', w)$, and that
  2. $\text{monoReq}(\text{W}_2, \phi, \text{w}, \sqsubseteq^{priv}) \mathrel{-\!\!*} \text{monoReq}(\text{W}_2', \phi, \text{w}, \sqsubseteq^{priv})$.

  (1) is shown using the private monotonicity property of (2) (public future worlds are also private future worlds), and (2) is shown by applying transitivity of private and public future worlds (the private future world of a public future world describes an overall private future world).

- Case: $\rho = \text{Temporary}$:
  To show: $\text{tempR}(a, p, W_2, \phi) \mathrel{-\!\!*} \text{tempR}(a, p, W_2', \phi)$.

  Similar to above, by applying the transitivity of public future worlds.

- Cases: $\rho = \text{Frozen } m$ and $\rho = \text{Revoked}$:
  To show: $\text{frozenR}(a, p, m, M^{std}) \mathrel{-\!\!*} \text{frozenR}(a, p, m, M^{std})$
  To show: $\top \mathrel{-\!\!*} \top$

  Cases are trivial.

□

The same cannot be proved for private changes to $\mathit{sharedResources}(W_2)$. Since temporary resources might not be monotone with regards to private future worlds, we must assert that the collection of standard states is free from any Temporary state. This can be done by asserting ownership over $\mathit{stsCollection}(\text{revoke}(W_1))$, giving rise to the following private update lemma.

Lemma 17. *If* $dom(W_2^{std}) = dom(W_2'^{std})$ *and* $W_2' \sqsupseteq^{priv} W_2$*, then*
$\mathit{stsCollection}(\text{revoke}(W_1)) * \mathit{sharedResources}(W_2) \mathrel{-\!\!*}$
$\mathit{stsCollection}(\text{revoke}(W_1)) * \mathit{sharedResources}(W_2')$

Proof. The proof proceeds similarly to Lemma 16, except for the case where ρ is Temporary. A priori, this case cannot be proved since the given invariant is assumed to be monotone with regards to public future worlds, but we now assume a private future world. However, unlike in Lemma 16, we now know more about ρ. Specifically, the ownership of $\mathit{stsCollection}(\text{revoke}(W_1))$ and $a \overset{std}{\longmapsto} \rho$ implies that $\text{revoke}(W_1)(a) = \rho$, from which we conclude that ρ cannot be Temporary, reductio ad absurdum. □

Before being able to do any private changes to $\mathit{sharedResources}(W_2)$, it's thus necessary to revoke all Temporary states from $\mathit{stsCollection}(W_1)$. This amounts to claiming ownership of all temporary resources in the instrumented machine state, and is typically done as the first action in a function's execution.

Lemma 18. *Let S be a set of addresses such that $W_1^{std}(a) = \text{Temporary}$ iff $a \in S$, then* $\mathit{stsCollection}(W_1) * \mathit{sharedResources}(W_2) \Rrightarrow\!\!\ast$
$\mathit{stsCollection}(\text{revoke}(W_1)) * \mathit{sharedResources}(W_2)$
$* \mathop{\Large\ast}_{a \in S} \exists p, \phi, \text{tempR}(a, p, W_2, \phi) * \mathit{rel}(a, p, \phi)$

Proof. Proof proceeds by induction over each a to (p, γ) mapping in M^{interp}, which means $\mathit{stsCollection}(W_1)$ is updated one address at a time. Let $W_{1_{mid}}$ denote the partly updated world state of *stsCollection*, obtained through the induction hypothesis. If the standard state of a is Temporary, $\mathit{stsCollection}(W_{1_{mid}})$ must be updated such that the standard state of a is Revoked.

Invoking Lemma 13 yields $a \overset{std}{\longmapsto} \text{Revoked} * \mathit{stsCollection}(W_{1_{mid}}[a := \text{Revoked}])$. Since $a \in S$, we must establish $\exists p, \phi, \text{tempR}(a, p, W_2, \phi) * \mathit{rel}(a, p, \phi)$, which corresponds exactly to the contents of *srMap* at address a. Finally, since the new standard state is Revoked, the new standard resource interpretation of a is trivially $\top$, and the proof can proceed until $W_{1_{mid}} = \text{revoke}(W_1)$. □

Note that the above lemma revokes *all* temporary resources from the instrumented machine state. Indeed, revocation is an all or nothing update, since one temporary resource might be invalidated by revoking another temporary resource that points to it. In this version, we do not assume knowledge about any temporary resources of addresses in S. However, in some cases, a subset of S will refer to known memory invariants. Hence, it's useful to state the following corollary, in which we additionally assume $\mathit{rel}(a, p, \phi)$ for some known p and ϕ, for all addresses a in that subset.

Corollary 3. *Let S and S' be disjoint sets of addresses such that $W_1^{std}(a) = \text{Temporary}$ $\forall a \in S \uplus S'$, then* $\mathit{stsCollection}(W_1) * \mathit{sharedResources}(W_2) * \mathop{\Large\ast}_{a \in S} \mathit{rel}(a, p, \phi) \Rrightarrow\!\!\ast$
$\mathit{stsCollection}(\text{revoke}(W_1)) * \mathit{sharedResources}(W_2)$
$* \mathop{\Large\ast}_{a \in S} \text{tempR}(a, p, W_2, \phi) * \mathop{\Large\ast}_{a \in S'} \exists p', \phi', \text{tempR}(a, p, W_2, \phi) * \mathit{rel}(a, p', \phi')$

Thus far, we have presented lemmas for applying public and private changes to the world, as well as revocation lemmas that relinquish ownership of all temporary resources of the instrumented machine state. We refer to the Coq mechanization for similar lemmas over the remaining function in Figure 3.18.

An often applied pattern is to revoke $\mathit{stsCollection}(W_1)$ and claim ownership over temporary resources by applying Corollary 3, and syncing up the memory invariants by applying Lemma 17, using Property 3.1. This pattern reflects the changes enacted by the calling convention, when a function is called by an adversary. Indeed, the calling convention generally induces certain patterns of change to the instrumented machine state. In the following section, we sketch out these patterns for each stage of the calling convention.

3.6.7.1 Effects of the Calling Convention on the Instrumented Machine State

In principle, the stack is a shared memory region managed by the calling convention, pointed to by a local URWLX capability. It is characterized in the instrumented

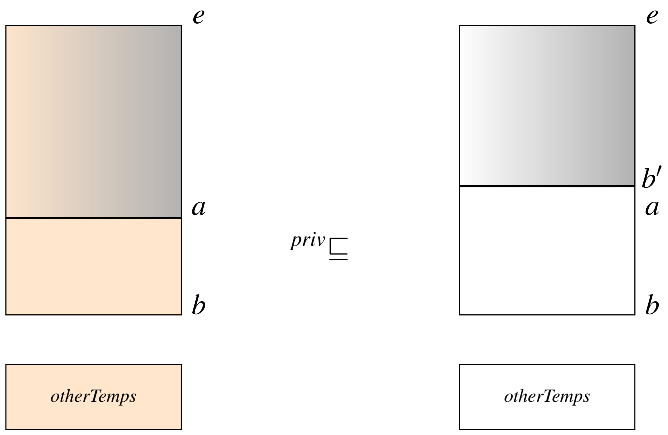


Figure 3.19: World Changes at Program Startup

machine state by Temporary states for the initialized part, and a mix of Temporary and singleton Frozen states for the uninitialized part. During execution, a function may claim ownership of parts of the stack, invalidating the stack as a whole. Upon calling a new function, the current stack frame is Frozen, while the uninitialized part is passed to the callee. All in all, stack regions may take the following states, represented on the left by a color encoding, that we will employ in ensuing diagrams.

▨	Interweaving of Temporary and singleton Frozen states
▧	Temporary states
▤	Interweaving of Revoked and singleton Frozen states
□	Revoked states
■	Cohesive Frozen block

Below are sketches of the general patterns of change performed by the calling convention, and their representations in the instrumented machine state. Let f be the function about to execute. Assume that f is invoked by some other function, and will itself invoke a function during its execution.

When called by a function The calling convention dynamically checks that r_{stk} stores a local URWLX capability, say (LOCAL,URWLX,b,e,a). The currently executing function implicitly claims ownership of their stack frame, whose size is presumed to be known at compile time. Since this invalidates the integrity of the stack as a whole, the instrumented machine state must be fully revoked. The resulting change is depicted in Figure 3.19, where $[b, b')$ describes the local stack frame.

Note that there may be Temporary regions outside the scope of the known stack that get revoked as a result. We will refer to their state as $\mathit{otherTemps} : \mathrm{Addr} \hookrightarrow \mathrm{Word}$.

Since the calling convention does not check the address a of the given stack capability, there is no relation between a and b'. As such, in order to claim ownership

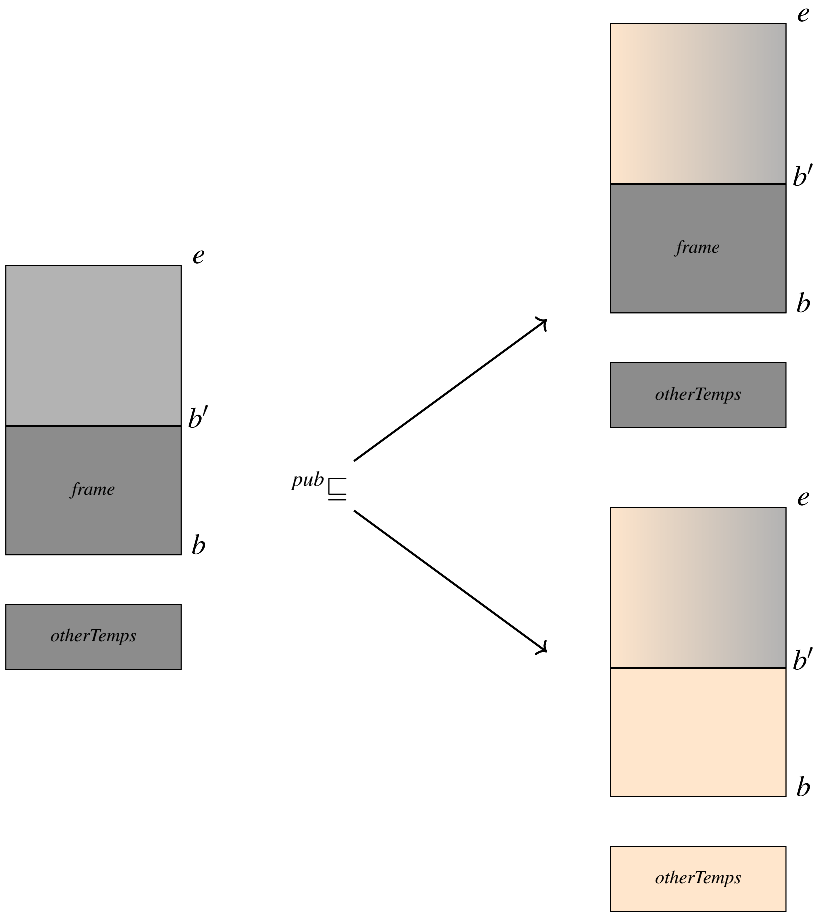


Figure 3.20: World Changes upon Call and at Callback

of the full stack frame, there may be (singleton) frozen states within $[a, b')$ that need to be revoked, in addition to the Temporary states. Let S denote the set of such addresses. Let W be the starting world state depicted on the left. The resulting world is a private future world of W, and can be described as follows (Note that revoked$(\cdot)$ and updated$(\cdot)$ commute):

$$W_f = \text{updated}(\text{revoked}(W), S, \text{Revoked}) \sqsupseteq^{priv} W$$

The updated instrumented machine state is obtained by applying Corollary 3, along a similar lemma for S, and Lemma 17. We will henceforth refer to the resulting world as W_f.

When calling a function and at callback When preparing for a function call, the calling convention stores an activation record on the stack, restricts the stack capability to point to the uninitialized part of the stack, and derives a local E capability pointing to the activation record, to serve as the return capability.

An important function of the calling convention is to encapsulate the local stack frame from the callee, thus guaranteeing its integrity during the callee's execution. This is represented in the instrumented machine state by freezing all the owned Temporary resources, namely the local stack frame and *otherTemps*. This is a private update to W_f, which does not break any invariants in the instrumented machine state, as W_f is fully revoked (Lemma 17).

Meanwhile, the ownership of Temporary resources from the uninitialized part of the stack are relinquished to the callee. Since the restricted stack capability will be fully uninitialized, they do not need to be Temporary; instead they are updated to singleton Frozen resources. Let $\mathit{frame} : \mathrm{Addr} \hookrightarrow \mathrm{Word}$ refer to the state of the local stack frame, and $\mathit{unused} : \mathrm{Addr} \hookrightarrow \mathrm{Word}$ refer to the state of previously Temporary states from $[b', e)$. The new world can then be described as follows (referred to as W_{out}):

$$W_{out} = \mathrm{uninitialize}(\mathrm{freeze}(W_f, \mathit{otherTemps} \uplus \mathit{frame}), \mathit{unused}) \sqsupseteq^{priv} W_f$$

Since the return capability is local, we expect the world at its invocation to be a public future world of W_{out}. Figure 3.20 shows the state of the world as control is passed to the callee (left), and the *two* possible worlds we can expect upon return (right). We will henceforth refer to either of these worlds as W_{ret}. The explanation behind the existence of two cases is subtle, but important in order to understand the degree of stack safety achieved by the new calling convention. The explanation is given in two phases, first purely technical, then how each case can intuitively be understood.

Both possibilities allow for a world where parts of $[b', e)$ may have become Temporary. However, in one version, the cohesive Frozen $\mathit{otherTemps} \uplus \mathit{frame}$ state is unchanged, while in the other it has become *fully* Temporary. From a technical standpoint, the justification is made by observing that in the standard state transition system (Figure 3.11), the transition from Frozen to Temporary is public. In a public future world, any Frozen state in W_{out} is thus either Temporary, or Frozen at the same map. The $\mathit{sharedResources}(W_{ret})$ predicate guarantees the cohesion of the block: either the full map remains Frozen, or it is fully Temporary.

In order to intuitively make sense of the two cases, it is useful to account for the possible situations in which the return capability an be invoked. As expected, the callee can invoke the return capability to return to f (corresponding to the case where the full map remains Frozen). The calling convention guarantees that once invoked, the return capability is in effect revoked from the callee. However, the calling convention makes no similar guarantees with respect to f's caller. While f's local stack frame might be cleared before returning, nothing stops the caller from reading the potentially uncleared contents of higher stack frames. As a result, the callee may use the stack to covertly communicate f's return capability to the caller. The return capability can thus, more surprisingly, also be invoked by f's caller, *after f has returned to it*. Thankfully, this means that f has now cleared its own stack frame, and the invocation does not lead to any unwanted privilege escalation. This second

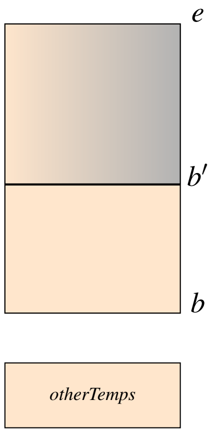


Figure 3.21: World Changes upon Return

situation is reflected in the public future world in which the Frozen state has become Temporary.

We can make some interesting observations based on these two cases. First, it illustrates that a callee may not themselves apply the calling convention, and may in fact leak their own encapsulated capabilities to other functions (such as the caller). In our attacker model, this poses no issue since we assume a simple binary distinction between the trusted code, and the untrusted context. However, the calling convention might not fit an attacker model where multiple mutually distrustful components, with clearly specified privileges, should be prevented from covertly sharing privilege with one another, such as in [3].

Second, we can observe that the calling convention does not prevent a caller from reading the previous contents of now popped stack frames. While the calling convention clears the local stack frame upon return, thus preventing unwanted privilege escalation, one could argue that such a read action ought to have been prevented in the first place. This aspect of stack safety is investigated in [144], and in Chapter 4 of this thesis.

Upon return Upon return, the calling convention clears the local stack frame, and invokes the return capability passed in r_0. Since this signifies the end of f's execution, it is important that no local changes are exposed to the context. As such, the final world should be a public future world of W.

This is achieved by (1) relinquishing ownership of the local stack frame by updating its addresses to Temporary, easily established since its content is cleared prior to returning, and (2) reinstating all the previously Temporary resources that happen to have remained frozen in W_{ret}. The latter is less straightforward, since it involves deriving the relevant memory invariants that are only known to hold at W. The trick is to prove that they hold when instantiated to the *final* world, by applying their public monotonicity property.

Let W_{final} refer to the final world, defined as follows:

$$W_{final} = \text{reinstate}(W_{ret}, dom(frame) \uplus dom(otherTemps) \uplus dom(unused)) \sqsupseteq^{pub} W_{ret}$$

By individually going over the affected states in W (note that we cannot use any transitivity property over the intermediary private future worlds), it is possible to prove that the final world is a public future world of W:

$$W_{final} \sqsupseteq^{pub} W \tag{3.6}$$

This proof is often dependent on the particular behavior of the function f. Figure 3.21 depicts the final world W_{final}. Given Property 3.6, the Frozen resources in *otherTemps* and *unused* can be re-established, and the return capability is provably valid in W_{final}. Each of these patterns will be applied in Section 3.6.9, but first, we need to detail how we prove that it's safe to execute unknown code.

3.6.8 Fundamental Theorem

With the definition of our logical relation in place, we can now state the fundamental theorem of our logical relation (FTLR). In broad terms, the FTLR states that if a range $[b,e)$ is safe to read, then it is safe to execute. The permission of the capability must itself be executable, and in particular if the capability is RWLX, then its locality must be LOCAL.

Theorem 10 (FTLR)**.** *Assume that* $p = \text{RX}$*,* $p = \text{RWX}$ *or* $(p = \text{RWLX} \land g = \text{LOCAL})$*. Assume also that* $\mathcal{V}(W)(p,g,b,e,a)$*. Then we have that* $\mathcal{E}(W)(p,g,b,e,a)$*.*

Proof sketch. We prove the FTLR by Löb induction, i.e. by assuming that the theorem holds later (after one step), we prove that it holds now. In order to take a step in the program, we consider the different possible instructions pointed to by the program counter. For each instruction, we look at all the possible cases: for example we need to distinguish between moving a constant into a register and moving from one register to another. If the instruction fails, we are done since we know the postcondition of the expression relation holds for a failed configuration. If the instruction succeeds, we prove safety of the resulting machine state and the updated program counter capability, and apply the induction hypothesis. Each instruction has many cases, especially when one considers all the possible ways an instruction can fail. To avoid a tedious blow-up of case distinctions, we use a general form of the program logic rules that separate the (interesting) success case from all the (uninteresting) possible failure cases. The `store` and `load` instructions also require us to access the memory invariants of the source and destination addresses. This is done using Lemmas 14 and 15 over the *sharedResources*(W) predicate, knowing that each address is accessed using a safe capability, which means we know its exact standard type in W. A particularly interesting case is that of the `storeU` instruction at offset 0. Let's consider the case where the destination is a valid capability with permission URWLX.

Case: $instr = \texttt{storeU}\ r_{dst}\ r_{srv}\ 0$

We own the following resources:

$$r_{dst} \mapsto (\text{URWLX}, l, b, e, a) \tag{3.7}$$
$$r_{src} \mapsto w_{src} \tag{3.8}$$
$$sharedResources(W) \tag{3.9}$$
$$stsCollection(W) \tag{3.10}$$
$$\mathscr{V}(W)(\text{URWLX}, l, b, e, a) \tag{3.11}$$
$$\mathscr{V}(W)(w_{src}) \tag{3.12}$$

To consider the case where `storeU` succeeds, we assume that $a \in [b,e)$. By definition of $\mathscr{V}$, we can thus extract $rel(a, \text{URWLX}, \mathscr{V})$ from Equation (3.11). Furthermore, we know:

$$\exists w, W(a) = \text{Frozen}(\{[a := w]\}) \lor W(a) = \text{Temporary} \tag{3.13}$$

We open $sharedResources(W)$ by applying Lemma 14, and extract resources for a. Given Equation (3.13), $a \overset{std}{\longmapsto} \rho$ and $stsCollection(W)$, we can apply Lemma 12 and infer that ρ is either Frozen or Temporary. We can thus derive the points-to predicate $a \mapsto v$, for some word v. We use it to apply the relevant weakest precondition rule for `storeU`, updating the points to predicate to $a \mapsto z$, and Equation (3.7) to $r_{dst} \mapsto (\text{URWLX}, l, b, e, a+1)$.

The final step is to close the region, establish $\mathscr{R}$ for the new register state, and conclude by applying the induction hypothesis. Since the state of a in W might have been frozen, changing its value means we must initialize its state to Temporary. In that case, we apply Lemma 13, resulting in $a \overset{std}{\longmapsto} \text{Temporary} * stsCollection(W[a := \text{Temporary}])$ (note that the resulting world is identical to W in the case where $W(a) = \text{Temporary}$).

Since $W[a := \text{Temporary}] \sqsupseteq^{pub} W$, we can apply *Lemma* 16 on the opened region to get $sharedResourcesOpen(W[a := \text{Temporary}], \{\cdots, a\})$. Next, we can close the region by applying Lemma 15, for which we must establish

$$\text{tempR}(a, \text{URWLX}, W[a := \text{Temporary}], \mathscr{V})$$

$\mathscr{V}(W[a := \text{Temporary}])(z)$ holds by Equation (3.12) and Lemma 11, while $\text{monoReq}(W[a := \text{Temporary}], \mathscr{V}, z, \sqsupseteq^{pub})$ can be derived from Lemma 11.

Finally, we use Lemma 11 to show that the register state is valid in the new world, and conclude by applying the induction hypothesis.

□

We use the fundamental theorem whenever we want to reason about unknown adversary code. For instance, if we go back to the third scenario, when Alice returns to Charlie, we can finish the execution simply by knowing that the return capability is

```
g1: malloc r_2 1
    store r_2 0
    move r_3 pc
    lea r_3 offset
    crtcls [(x,r_2)] r_3
    jmp r_0
f1: reqglob r_adv
    prepstack r_stk
    store r_env 0
    scallU r_adv ([],[r_0,r_adv,r_env])
    (continues in next column)
```

```
  (continued from previous column)
  store r_env 1
  scallU r_adv ([],[r_0,r_env])
  load r_adv r_env
  assert r_adv 1
  getb r_1 r_stk
  add r_2 r_1 10
  subseg r_stk r_1 r_2
  mclear r_stk
  rclear RegName\{pc,r_0}
  jmp r_0
end:
```

Figure 3.22: The awkward example using our new calling convention. It relies on local state encapsulation and well-bracketedness as provided by `scall`. `g1` is the entry-point of the program; when executed, it creates a closure (as an E capability) whose body executes `f1`. *offset* is the offset to `f1`. Changes following our new calling convention are highlighted in blue.

safe: if it was an enter capability, we directly apply the execute condition, and if it was an executable capability, we know that its range is safe to read, and thus by the fundamental theorem, safe to execute. Let us see in slightly more detail what kind of properties we can prove about example programs.

3.6.9 A Concrete Scenario: the "Awkward Example"

We demonstrate the use of our logical relation model by verifying the correctness of a tricky example program, in a scenario where known (verified) code calls to and is called by unknown (possibly adversarial) code. The example is a low-level version of the "awkward example" [39]:

$$\text{let}\, x = \text{ref}\, 0\, \text{in}\, \lambda f.\, (x := 0;\; f();\; x := 1;\; f();\; \text{assert}(!x = 1))$$

The correctness of this program—the assert never fails—relies on local state encapsulation (the adversary cannot modify private location x) and well-bracketed control flow (the adversary must return to where he was last invoked). Taking advantage of these properties when they are built into the language is already quite challenging: Dreyer et al. deploy a step-indexed Kripke logical relation with public and private transitions to achieve that task. In subsequent work, Skorstengaard et al. [129] verify (on paper) a low-level version of this example adapted to a capability machine with local capabilities. In that setting, local state encapsulation and well-bracketed control flow are not properties of the language (they do not make sense at the machine-code level), but are instead consequences of the secure calling convention implemented in the example.

In this work, we adapt the machine-code example from Skorstengaard et al. [129] to use our improved calling convention with uninitialized capabilities, and prove the updated code correct using Iris and our logical relation mechanized in Coq. Our code appears in Figure 3.22, with differences highlighted in blue. There are two main changes: first, secure function calls are made through a new `scallU` macro that implements the stack discipline described in Section 3.4.2; second, we now only clear our own stack frame before returning to the adversary instead of the whole stack (here, this means clearing ten memory cells instead of possibly thousands or millions).

We carry out the proof in two main steps. In a first step (Lemma 19), we show that the program entry-point `g1` is safe according to the expression relation $\mathcal{E}$. In a second step (Theorem 11), we use the standard adequacy theorem of Iris, and derive a closed statement for the correctness of our program against the operational semantics of the machine[2].

The bulk of the work consists in proving the first lemma: the proof requires allocating a custom state transition system for the encapsulated reference, stepping through the code of the program using the program logic rules, and using the FTLR (Theorem 10) to reason about calls to unknown code (made by `scallU` and the final `jmp` to an unknown return pointer). What follows is a rough sketch of the proof, focused on the evolution of the instrumented machine state.

Lemma 19. *For any world W, assuming that the memory has been properly initialized[3] in region $[b_{\text{awk}}, e_{\text{awk}})$ with the code of the program and a pointer to the* `malloc` *and* `assert` *subroutines, we have:*

$$\mathcal{E}(W)(\text{RX}, \text{GLOBAL}, b_{\text{awk}}, e_{\text{awk}}, \texttt{g1}).$$

Proof. In an initial phase, we apply relevant weakest precondition rules to step through the instructions in $[\texttt{g1}, \texttt{f1})$. The final instruction `jmp` r_0 jumps to an unknown target, but can be reasoned about given the assumption that it is in the value relation. If the jump succeeds, and the program does not immediately crash, we apply the fundamental theorem (Theorem 10), and assert that the target program is in the expression relation.

Next, in order to apply the weakest precondition granted by the expression relation, we must show that the new register state is in the register relation. In particular, we must show that the closure generated by `crtcls` $[(x, r_2)]$ r_3 is in the value relation. A preliminary step is to allocate a custom state transition system and a related iris invariant, for the local state of that closure. As expected, this state transition system corresponds exactly to the one established in the original proof of the (high level) awkward example [39]. Let $rn_{awk} \in \mathsf{RNames}$ be fresh in W^{cus}, and let W' be the result

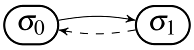
[2] We have also instantiated Theorem 11 with a simple adversarial code that invokes the awkward example with $f = (\lambda().\,())$ and additionally proved that, in that setting, the whole machine runs and gracefully halts.

[3] These assumptions are kept intentionally vague for brevity. Full statements can be found in the Coq formalization.

of allocating the new state transition system (depicted below) in W:

$$stsCollection(W') \tag{3.14}$$

$$\boxed{\begin{array}{l}\exists s, rn_{awk} \overset{cus}{\longmapsto} s \, * \\ \mathit{if}\ s = \sigma_0\ \mathit{then}\ l \mapsto 0\ \mathit{else}\ l \mapsto 1\end{array}}^{\mathcal{N}.awk} \tag{3.15}$$

$rn_{awk} \overset{rel}{\longmapsto}$  (3.16)

Since $W' \sqsupseteq^{pub} W$, we get $sharedResources(W')$ by applying Lemma 16. What remains is to prove the following (where b, e and a are the dynamically allocated addresses storing the closure around $[\mathtt{f1}, \mathtt{end})$ and the dynamically allocated l):

$$\mathcal{V}(W')(\mathrm{E}, \mathrm{GLOBAL}, b, e, a) \quad = \quad \Box \forall W'' \sqsupseteq^{priv} W', \mathcal{E}(W'')(\mathrm{RX}, \mathrm{GLOBAL}, b, e, a)$$

Let W_{start} be a private future world of W. Equation (3.15) and Equation (3.16) are persistent, and thus still within scope after introducing the $\Box$ modality. Unfolding $\mathcal{E}$, we then introduce the following assumptions to our context:

$$stsCollection(W_{start}) \tag{3.17}$$

$$sharedResources(W_{start}) \tag{3.18}$$

$$\mathcal{R}(W_{start})(reg) \tag{3.19}$$

$$\mathsf{pc} \mapsto (\mathrm{RW}, \mathrm{GLOBAL}, b, e, a) * \mathop{\huge *}_{(r,w) \in reg / \mathsf{pc}} r \mapsto w \tag{3.20}$$

Throughout the proof, we will change the instrumented machine state according to the pattern described in Section 3.6.7.1. As such, we start by claiming ownership over the current stack by revoking W_{start}, as well as the addresses that are frozen but within the stack frame bounds (denoted by S).

$$stsCollection(\mathrm{updated}(\mathrm{revoke}(W_{start}), S, \mathrm{Revoked})) \tag{3.21}$$

$$sharedResources(\mathrm{updated}(\mathrm{revoke}(W_{start}), S, \mathrm{Revoked})) \tag{3.22}$$

For clarity, we here omit the Temporary resources that result from applying Corollary 3. Suffices to know, we obtain the relevant points-to predicates for the local stack frame and the unused part of the stack, alongside the remaining unknown Temporary standard resources. We will use the same naming conventions as in Section 3.6.7.1, and use *otherTemps* to refer to the state of unknown Temporary resources, and *unused* to refer to the unused (and Temporary) part of the stack.

Next, we apply weakest precondition rules to step through the program instructions, most interesting of which is `store` r_{env} 0, as it involves storing 0 to the local state l. The necessary points-to predicate is in the iris invariant 3.15. Upon opening the invariant, we gain ownership over the custom state $rn_{awk} \overset{cus}{\longmapsto} s$, and a conditional stating the two cases for l. Regardless of rn_{awk}'s state in W', since W_{start} is a private future world of W', s can be either σ_0 or σ_1. However, once the `store` instruction has executed, we get $l \mapsto 0$, and the state of rn_{awk} must be updated to σ_0, in order to properly close the invariant again.

$$stsCollection(\text{updated}(\text{revoke}(W_{start}), S, \text{Revoked})[rn_{awk} := \sigma_0]) \tag{3.23}$$

$$sharedResources(\text{updated}(\text{revoke}(W_{start}), S, \text{Revoked})) \tag{3.24}$$

As a result, the instrumented machine state is desynchronized. Furthermore, if s happened to equal σ_1, the resulting change is private. We must thus apply Lemma 17 to bring the *sharedResources* up to date.

$$sharedResources(\text{updated}(\text{revoke}(W_{start}), S, \text{Revoked})[rn_{awk} := \sigma_0]) \tag{3.25}$$

We will refer to the above world state as W_1. What follows is the call to the unknown function. We apply the pattern as described in Section 3.6.7.1. Let *frame* be the current state of the local stack frame. Below is the instrumented machine state, upon jumping to the unknown function. We will refer to the new world as W_{out}.

$$stsCollection(\text{uninitialize}(\text{freeze}(W_1, otherTemps \uplus frame), unused)) \tag{3.26}$$

$$sharedResources(\text{uninitialize}(\text{freeze}(W_1, otherTemps \uplus frame), unused)) \tag{3.27}$$

In order to reason about the unknown call, the current register state must be proved valid at W_{out}. The calling convention subroutine clears most registers, except the restricted stack capability, and the return capability. Validity of the stack capability follows from definition of W_{out}. Validity of the return capability, on the other hand, is more interesting, as it requires reasoning about the execution of the callback:

$$\mathscr{V}(W_{out})(\text{E}, \text{LOCAL}, \cdots) \quad = \quad \Box \forall W_{ret} \sqsupseteq^{pub} W_{out}, \mathscr{E}(W_{ret})(\text{RX}, \text{LOCAL}, \cdots)$$

Recall that in any given W_{ret}, we must consider two possible cases; either the Frozen frame remains Frozen at the same state, or it has become fully Temporary. In the latter case, we are not reasoning about the execution of the callback. Instead, we are now executing a list of arbitrary but valid words. In fact, since the return capability has authority exactly over the addresses in *frame*, which have become fully Temporary, it is possible to derive $\mathscr{V}(W_{ret})(\text{RX}, \text{LOCAL}, \cdots)$. In other words, we are reasoning about a range that is safe to read, which is precisely what the fundamental theorem tells us is safe to execute. We conclude the case by applying Theorem 10.

The remainder of the proof proceeds in the same manner; we reason about the callback by stepping through the activation record instructions, perform the typical instrumental machine state changes related to the calling convention, and derive properties about the custom state rn_{awk} whenever needed. More precisely, since we now know that rn_{awk} was in state σ_0 in W_{out}, we can infer that updating it to σ_1 in the first callback yields a public change. Furthermore, during the second callback, we can infer that it is still at the state σ_1, since the given future world in the second callback is again public.

Finally, upon return, we reinstate all the standard resources revoked throughout the proof. Since the final state of rn_{awk} is σ_1, we can conclude that the final world is a public future world of W_{start}. □

In the next step, we use the standard adequacy theorem of Iris, and derive the final closed statement for the correctness of our program.

Theorem 11. *(Correctness of the awkward example) Let reg* $\in$ Reg*,* $m \in$ Mem *and*

$$c_{\text{awk}} \triangleq (\text{RX}, \text{GLOBAL}, \ldots) \quad c_{\text{stk}} \triangleq (\text{URWLX}, \text{LOCAL}, \ldots) \quad c_{\text{adv}} \triangleq (\text{RWX}, \text{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer. Furthermore, assume that:

- *m has been initialized with the code of the program and subroutines (pointed to by c_{awk}), an uninitialized stack (pointed to by c_{stk}), and unknown adversarial code (pointed to by c_{adv});*
- *$reg(\mathsf{pc}) = c_{\text{awk}}$, $reg(r_{\text{stk}}) = c_{\text{stk}}$, $reg(r_0) = c_{\text{adv}}$ and $reg(r) \in \mathbb{Z}$ otherwise;*
- flag *denotes the memory address set to* 1 *by the* `assert` *subroutine in case of failure;*
- *$m(\text{flag}) = 0$.*

If $(\mathsf{Repeat\ SingleStep}, (reg, m)) \rightarrow^* (\mu, (reg', m'))$ *then* $m'(\text{flag}) = 0$.

Theorem 11 states that, starting from a properly initialized machine state, the in-memory flag set by the `assert` routine remains set to 0 at every step of the execution—meaning that the call to `assert` never fails. Obtaining Theorem 11 from Lemma 19 is mostly mechanical: this highlights one of the benefits of using Iris, whose built-in soundness theorem can be leveraged to obtain a program specification stated directly against the operational semantics of the machine.

3.7 Implementation

We have implemented uninitialized capabilities in the CHERI-MIPS ISA for the uncompressed 256-bit capability format (we believe that the implementation should be possible for other capability formats as well[4]). In CHERI-MIPS the stack grows downwards (from higher memory adresses to lower memory addresses) and the implementation of uninitialized capabilities is inverted to reflect the stack growth. Concretely, uninitialized capabilities only allow reading from the range $[a, e]$ and a moves downwards on writes below the current a, just like the stack. Capabilities now have a bit indicating if they are uninitialized or not. Some existing CHERI-MIPS instructions are modified to take the uninitialized permission into account: the load instructions and those that modify a capability's cursor. For experimentation purposes, we have opted to add separate store instructions for uninitialized capabilities, leaving the old store instructions intact. Additionally, we add an instruction to make a regular

[4] Available at https://zenodo.org/record/4067949

capability uninitialized and a new variant of CSetBounds (the CHERI version of subseg) that is needed for technical reasons.

These modifications result in a CHERI-MIPS simulator that supports uninitialized capabilities. We have also added support for the new instructions to the Clang/LLVM assembler for CHERI-MIPS.This allows us to write assembly programs with the new instructions and run them on the simulator. With the simulator and assembler in place, we were able to experiment with the new calling convention by manually modifying assembly programs. The calling convention of Section 3.4.2 is slightly modified for CHERI-MIPS because CHERI uses pairs of sealed capabilities (a code capability and data capability) instead of enter capabilities. This means we do not need to store return closures on the stack (like for StkTokens [132]), but otherwise makes little difference.

Although more investigation is needed, our results suggest that uninitialized capabilities and the calling convention from Section 3.4.2 can be adapted and applied in a CHERI setting.

3.8 Related Work

We already discussed some related work in the introduction, which we briefly recall now. We follow an existing line of work on capability machines [27, 91, 154, 157], and in particular the CHERI family featuring local capabilities [154, 157] that provide a form of revocable capabilities. To our knowledge, uninitialized capabilities and the idea of using them to reduce the cost of local capability revocation are both new.

Other forms of revocation have been proposed in capability machine contexts. A line of work of the CHERI project (CHERI-JNI [29], CHERIvoke [163], Cornucopia [52]) presents a general revocation mechanism for memory managed through a dedicated memory allocator. In that setting, revocation happens by sweeping through the whole memory and clearing obsolete (revoked) pointers. This GC-like approach to revocation is somewhat orthogonal to our stack-based revocation mechanism. The authors mostly focus on practical feasibility, and do not formally state or prove the guarantees provided by their revocation procedure.

Linear capabilities [157] have also been proposed as a lightweight revocation mechanism, both for implementing a secure calling convention [132] providing similar guarantees as ours, and as a secure compilation target for separation logic verified code [147]. However, there are concerns as to whether the atomic store-and-clear operation required by linear capabilities can be realistically implemented in hardware without an important performance penalty and whether they would be easy to support in existing compilers [128, §3.6.2]. We expect uninitialized capabilities to be a more benign extension from a micro-architectural and compiler perspective.

Another category of related work is on formalizing capability safety, i.e. characterizing the guarantees provided by a capability machine or language runtime. In the context of high-level languages with object capabilities, Maffeis et al. [93] define a syntactic notion of capability safety based on reachability between objects. This kind of criterion is however of limited expressive power as it is not directly defined with respect to the actual behaviour of objects. Drossopoulou et al. [42] formalize a form of capability safety in their Chainmail specification language. It can be used to capture properties of object-oriented programs like "An account's balance can be changed only if a client has access to that particular account".

More closely to our current work, Devriese et al. [37] propose a more expressive, semantic definition of capability-safety for object capabilities, based on a Kripke logical relation with public and private transitions which is not unlike ours. Swasey et al. [138] extend this line of work by showing that a similar logical relation can be used to give compositional specifications for the robustness of object capabilities patterns, and formalize their work in Coq using Iris.

Other related work has considered capability safety of (low-level) capability machines. Nienhuis et al. [109] build a formal model of the CHERI ISA, and formally verify a number of architectural security properties using Isabelle/HOL. A key security property they prove is capability monotonicity, meaning that the machine does not allow creating new capabilities out of thin air, and therefore, that an unknown code component can only modify parts of memory it has access to through its reachable capabilities. This is a somewhat syntactic property in nature, and it has an important limitation: it only holds until the code jumps to an enter capability (or sealed capability in the case of CHERI), which purposely gives access to new capabilities in a non-monotonic way. Therefore, their security properties only hold within a single "component". Our definition of capability safety, although more involved, allows reasoning about a complete machine execution, with arbitrary calls between different security domains and dynamic evolution of invariants and boundaries. Akram El-Korashy [9] has studied a formal model of the CHERI capability machine and proved some properties of it. Their main capability safety property captures a whole-system form of capability monotonicity that appears unsuitable for reasoning.

The work by Skorstengaard et al. [129, 131] is probably the most closely related to our own. As discussed before, they define capability safety for a capability machine with local capabilities as a logical relation, and propose a secure calling convention based on local capabilities. Our contributions are the introduction of a more efficient calling convention using uninitialized capabilities and a more expressive model (with the introduction of Frozen regions), as well as our formalization of our work in Coq using Iris. In subsequent work, Skorstengaard et al. [132] verify a secure calling convention based on linear capabilities. They phrase their result as a fully-abstract compilation theorem, rather than by verifying challenging examples, as they did in their previous work, and as we do here. This is an interesting perspective for future work: we believe that we could alternatively prove a similar theorem to characterize the correctness of our secure calling convention.

There are a number of previous work on using logical relations with public/private transitions to account for well-bracketed state changes [37, 39, 129], as well as using Iris to mechanize logical relations using higher-level constructs [61, 142, 143]. Our combination of the two is novel: we use lightweight Kripke worlds and Iris saved predicates to allow for precisely tracking the relationship between intermediate logical states (which would be impossible using Iris invariants), but we avoid solving recursive domain equations or working explicitly with step indices.

A final category of related work is on program logics for low-level code. Our program logic deals with code stored in memory as data, uses continuations to specify sequences of instructions, in combination with step-indexing to deal with unstructured control flow, and uses separation logic to model the resources associated to registers and memory. These features can often be found in a number of previous works [24, 31, 71, 104, 107]. The distinguishing feature of our program logic is that it is built on top of an existing general purpose logic. Consequently, we can (and we do) exploit the powerful features of Iris to reason about the low-level programs that we consider.

3.9 Conclusion

Local capabilities potentially provide an efficient but restricted revocation primitive in capability machines, with many possible applications. We have demonstrated how uninitialized capabilities can make them actually live up to this potential by solving an important performance problem. Moreover, using our novel formalized model of capability safety, we have demonstrated that the combination of local and uninitialized capabilities lends itself to machine-checked reasoning. In particular, we have verified an implementation of a classical example from the literature, which makes advanced use of local capability revocation through our modified calling convention. The example is, by the way, longer than it looks (400 instructions after unfolding macros). This shows the power of local capability revocation using uninitialized capabilities as well as the expressiveness of our reasoning infrastructure. Finally, our initial results suggest that uninitialized capabilities and our new calling convention can be practically applied in a more realistic setting like CHERI. We believe these different results combined make a strong case for the addition of uninitialized capabilities in CHERI and other capability machines.

Acknowledgements We thank the anonymous reviewers for valuable comments and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation; by the Research Foundation - Flanders (FWO) under grant number G0G0519N; and by DFF project grant no. 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU). Thomas Van Strydonck holds a Research Fellowship of the Research Foundation - Flanders (FWO). Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project.

Chapter 4

Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities

This chapter is an extended version of the following conference publication:

> Aïna Linn Georges, Alix Trieu, Lars Birkedal.
> *Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities*
> Proceedings of the ACM on Programming Languages (OOPSLA), 2022, 6.

The extension consists of

- An small addition to the discussion on dangling stack pointers; Section 4.2.2.1
- A proof sketch of the fundamental theorem of logical relations; Section 4.4.2.1
- A proof of an example application of the unary model; Section 4.4.3.1
- Added technical details about the binary model used to reason about confidentiality, including a proof of an example application of the model; Section 4.5
- Formal definitions of various auxiliary definitions used to define contextual equivalence, and the overlay semantics; Section 4.6.2

Abstract

Capability machines are a type of CPUs that support fine-grained privilege separation using *capabilities*, machine words that include forms of authority. Formal models of capability machines and associated calling conventions have so far focused on establishing two forms of stack safety properties, namely local state encapsulation and well-bracketed control flow. We introduce a novel kind of *directed* capabilities and show how to use them to make an earlier suggested calling convention more efficient. In contrast to earlier work on capability machine models we do not only consider integrity properties but also confidentiality properties; we provide a unary logical relation to reason about the former and a binary logical relation to reason about the latter, each expressive enough to reason about temporal stack safety. While the logical relations are useful for reasoning about concrete examples, they do not on their own demonstrate that stack safety holds for a large class of programs. Therefore, we also show full abstraction of a compiler from an overlay semantics that internalizes the calling convention as a single call step and explicitly keeps track of the call stack and frame lifetimes to a base capability machine. All results have been mechanized in Coq.

4.1 Introduction

Lack of memory safety is an important source of security bugs, for instance, 70% of all issues in Microsoft products [141] and in the Google Chrome browser [32] are memory safety related. It is thus not surprising that a large number of software or hardware protection mechanisms such as shadow stacks, stack canaries, address space layout randomization, etc (see [139] for a survey) have been proposed. Capability machines have recently risen as a promising solution to memory safety vulnerabilities; quoting a Microsoft study, “[capability machines] would have deterministically mitigated at least two thirds of all those issues” [73].

Capability machines are a kind of architecture that enable fine-grained memory protection using tagged memory [27, 36, 91] and *capabilities*, a form of unforgeable memory pointers with a certain amount of authority, in the form of a permission, range, etc. Over the last decade, CHERI [158], a family of capability machines, has matured into an extensive design featuring, among other, a full UNIX-style operating system, CheriBSD [154]. Ideas from CHERI are currently being adopted by ARM in their Morello project [12], which is aimed at developing concrete CPU designs and prototypes that could be implemented in future hardware.

One of the promises of capability machines is that they can enforce security properties that we expect from high-level languages, in particular stack safety, *even when machine code is linked with other untrusted and possibly adversarial machine code*. This potential is not yet realized in practice. In particular, while CheriBSD does make use of so-called local capabilities to limit the impact of potential bugs, it does not rely on them for enforcing security properties. This is likely because a secure calling convention based on local capabilities could be too inefficient as it would require a lot of stack clearing. Indeed, this is the case for the first known provably secure calling convention based on local capabilities [131] — this calling convention requires to clear the full stack before and after every call. This has led to research on other calling conventions based on novel forms of capabilities. In particular, Skorstengaard et al. [132] proposed a calling convention based on so-called linear capabilities, which, however, are believed not to be efficiently implementable in hardware. This motivated another proposal by Georges et al. [55] who suggested a secure calling convention based on a combination of so-called uninitialized capabilities and local capabilities, and which only involves a modicum of stack clearing per call, on the order of a single stack frame.

The works cited above on provably secure capability-machine-based calling conventions have all focused on spatial memory safety, in particular local state encapsulation and well-bracketed control flow. In another direction, Tsampas et al. [144] recently proposed a kind of capabilities including "lifetime" information to enforce *temporal* memory safety, e.g., that the content of popped stack frames cannot be accessed. However, one problem with implementing this proposal is that in order to allow for a call depth of size 2^n, n bits would be required in the encoding of the lifetime information for a capability, which renders it impractical.

In this paper, we propose a novel kind of so-called *directed* capabilities and show how they can be used in combination with uninitialized capabilities to realize a new calling convention, which is efficient (it does not involve any stack clearing at all) and which provably enforces both spatial and temporal stack safety properties.

More precisely, we present CERISEM, an extension of the low-level capability machine model of Georges et al. [55], with a novel form of directed capabilities, for which we present a novel stack-based calling convention. We show that it provably guarantees spatial and temporal stack safety. In light of the fact that it is actually quite subtle to capture stack safety properties formally, as also emphasized in a recent paper by Anderson et al. [10], we include a detailed discussion of the stack safety properties we consider and how our novel approach improves over earlier proposals, see Section 4.2. We include a discussion of the impact of stack objects on stack safety properties; prior work on local capability machines have largely ignored stack objects, but they have a significant impact on the guarantees provided by the capability machine. In contrast to the earlier formal models for capability machines mentioned above, we do not only consider integrity properties but also (stack) confidentiality properties.

To formally establish *integrity*, we follow the approach of Skorstengaard et al. [131] and develop a unary Kripke logical relations model, which captures capability machine safety. Our model is an extension of the one by Georges et al. [55]; the novelty consists of an extension to account for temporal safety. There are two facets to this: a simple one, which is to extend the definition to also treat directed capabilities, and a challenging one, which is to extend the model, in particular, the Kripke worlds, to capture the enforcement of temporal properties. The latter means that our model makes use of a novel kind of state transition system for the Kripke worlds.

To formally establish *confidentiality*, we further develop a *binary* logical relations model. We show that the binary logical relation implies contextual refinement so that it can serve as a sound proof method for establishing contextual equivalence and hence confidentiality. To the best of our knowledge, this is the first binary logical relations model for a low-level capability machine model.

We demonstrate that the unary and binary logical relations models can be used to prove stack safety properties for challenging example programs; we focus on examples that have not been considered in the literature before.

To give further evidence for the claim that our novel directed-capability-based calling convention actually does capture stack safety, we follow the approach of Skorstengaard et al. [132] and show full abstraction of a compiler from an overlay semantics that internalizes the calling convention as a single call step and explicitly keeps track of the call stack and frame lifetimes to the base capability machine. The idea is that the overlay semantics clearly enforces stack safety; our overlay semantics is related to the one used in [132] but now accounts for temporal properties by completely removing popped stack frames from the stack once their lifetime is over (technical differences are detailed in Section 4.6).

We have mechanized all of the models and results presented in the paper on top of the mechanization of the Iris program logic [74, 75, 77, 82] in Coq [83, 84]. The Iris-Coq mechanization [56] can be found online at `https://github.com/logsem/cerise-stack-monotone/releases/tag/OOPSLA2022`.

4.2 On the Stack Safety of Capability Machines

In this section, we explore the properties that make up stack safety in the context of capability machines. We follow Anderson et al. [10], who define multiple degrees of stack safety, as various conjunctions of local state encapsulation (LSE) and well-bracketed control flow (WBCF). In particular, our goal is to reach a notion of stack safety that falls within their definition of observational stack safety, which also covers the temporal aspect of LSE. A key takeaway of this section is to highlight how existing calling convetions incur undesired overhead in order to enforce stack safety. Unlike Anderson et al. [10], we consider LSE, WBCF and temporal stack safety for machines with both a stack and a heap. For clarity, we illustrate each property with an example written in a C-like language, though we actually consider the underlying assembly code. Next, we explore two interesting aspects of these properties that are particularly tricky. Finally, we survey previously proposed capability machine calling conventions and show where they fall on the spectrum of stack safety, including the novel calling convention and efficient enforcement mechanism we present in this paper.

4.2.1 A Family of Stack Safety Properties

4.2.1.1 Local State Encapsulation

Local state is a concept that exists in both low and high level languages. In a low-level language with a stack, local state often refers to the encapsulation of local variables in a stack frame. For instance, in Listing 4.1, the local variable `y` is a part of `f`'s local stack frame, and is not shared with the arbitrary adversarial code `adv`; and hence the assert, stating the integrity of `y`, should succeed.

```
void adv(void);
void f(void) {
  int *y; // allocated on
  *y = 2; // the stack
  adv(); assert (*y == 2)}
```

Listing 4.1: Integrity: frame

```
void adv(void);
void f(void) {
  static int x = 2;
  adv(); assert (x == 2)}
```

Listing 4.2: Integrity: environment

Figure 4.1: local state encapsulation: integrity

```
void adv(void);
void f(void) {
 int *y; *y = 2; adv()}
void g(void) {
 int *y; *y = 3; adv()}
```

Listing 4.3: Confidentiality: frame

```
void adv(void);
void f(void) {
 static int x = 2; adv()}
void g(void) {
 static int x = 3; adv()}
```

Listing 4.4: Confidentiality: environment

Figure 4.2: local state encapsulation: confidentiality

In high-level languages, local state may also refer to the state encapsulated within the scope of a closure. Consider Listing 4.2, where function `f` possesses some private state `x` (a `static` variable persists across calls, similarly to local variables in closures). Upon return, the integrity of `x` is tested with an assert statement. If `x` is not properly encapsulated, `adv` may modify `x`, and the assertion fails.

The stack is used to store local variables as well as the local environment to be reclaimed upon return of a call. When discussing LSE, we will refer to the local state being the local stack frame not shared with a callee, as well as the state encapsulated within a closure, which ought to stay encapsulated not just from the callee, but from the caller as well. For instance, an adversarial context may call `f` in Listing 4.2 multiple times, but it should never get access to the private state `x`.

Following Anderson et al. [10], we must distinguish between *local state integrity* and *local state confidentiality*. Local state integrity states that the local stack frame is protected from changes by the callee, local state confidentiality states that the local stack frame cannot be read by the callee and thus influence their behaviour. Hence local state confidentiality is a binary property. For example, Listing 4.3 contains two programs `f` and `g` with *different* local states that should stay hidden from the arbitrary function `adv`. Local state confidentiality guarantees the contextual equivalence of the functions `f` and `g`. Similarly, Listing 4.4 depends on the same local state confidentiality, but for the environment of closures.

Since we also use the stack for the encapsulated environment of closures, our notion of LSE includes the integrity and confidentiality of the environment of a closure against the full context. Indeed, a closure needs to protect its private state against *both* callees and callers.

4.2.1.2 Well-Bracketed Control Flow

```
1  void adv(void);
2  void f(void) {
3    static int x = 0;
4    x = 0; adv();
5    x = 1; adv();
6    assert (x == 1); }
```

Listing 4.5: Awkward Example

```
1   int N, K;
2   void h(int* x) { *x = 0 }
3
4   void g(int* x) {
5     char* t[K]; h(x) }
6
7   void f(int** x) {
8     char* t[N];
9     int z; *x = &z }
10
11  int main(void) {
12    int* x; f(&x);
13    g(x); return 0 }
```

Listing 4.6: Example violating temporal stack safety

Another common property in high-level languages is well-bracketed control flow. For example, consider Listing 4.5, which is a variant of the classical "awkward example" [39]. Here `f` possesses some local state `x` (line 3), which is set to `0` before calling some arbitrary adversarial code `adv()`. After the call returns, `x` is set to `1` before calling the adversary again. Finally, `x` is checked to be still equal to `1` at the end. If WBCF is not enforced, then during the second call to `adv` on line 5, `adv` could store the return pointer to line 6 in its own private state, and call `f`, which would then set `x` to `0` before calling `adv` again who can finally use the return pointer to line 6 it kept and fail the assertion.

It has been shown that both some form of LSE and WBCF can be enforced on capability machines, even in the presence of arbitrary code [55, 129, 132]. We will give more details on how this is enforced in Section 4.3.2.

4.2.1.3 Temporal Stack Safety

In another direction, Tsampas et al. [144] study the issue of temporal safety. Consider the code in Listing 4.6[1], where `&x` on line 12 is a pointer to a location containing another pointer. After the call to `f`, there is now a pointer at `&x` to the location `l` previously occupied by `z` on line 8. The value of `l` depends on a global variable `N`. It should be noted that `l` is *stale* after the return and should not be allowed to be passed down. Nevertheless, `l` is passed to `h` through `g`. For well chosen values of `K` and `N`, it is possible that `l` coincides with where the return pointer of `h` is stored and thus the store at line 2 can lead to the control flow being hijacked. This example shows a temporal stack safety violation that exploits a dangling stack pointer. To address this issue, Tsampas et al. propose that capabilities are extended with "lifetime" information, basically the call depth of the function's stack frame, and that capabilities with longer lifetime may not be used to store a capability with shorter lifetime. This would disallow the store on line 9 in the example. In essence, it disallows dangling

[1] An example adapted from [144].

stack pointers to be stored on the stack, and thus to be passed down the call stack beyond their lifetime.

Also related to temporal stack safety, Anderson et al. [10] find that the lazy tagging and clearing micro-policy of Roessler and DeHon [117] violates the temporal aspect of observable stack safety, and repairs it by generating fresh identifiers for *each* call, requiring an unbounded number of tags.

4.2.2 Two Subtleties of Stack Safety

```
void f(void) {
  static int x = 2;
  int *y;
  *y = &x;
  assert (x == 2); }
```

Listing (4.7) Dangling stack

```
void f(void) {
  int *x; *x = 2; }
void g(void) {
  int *x; *x = 3; }
```

Listing (4.8) Temporal confidentiality

```
int g(char* z, char* in)
int f(char* in) {
  int *y = 2;
  char* z = ...;
  g(z, in);
  assert (y == 2); }
```

Listing (4.9) Integrity with stack objects

We now highlight two subtleties of stack safety which previous works have mostly glossed over in the context of capability machines.

4.2.2.1 Elaborating on Temporal Stack Safety

Stack frame lifetime intuitively dictates that the content of a popped frame should not be read once popped. Tsampas et al. define temporal stack safety as the absence of dangling stack pointers passed down the call stack (cf. Listing 4.6). Here we wish to emphasize that the absence of dangling stack pointers should also mean that no caller should be able to (re)gain access to a dangling stack pointer when they resume. Concretely, consider Listing 4.7, where `f` possesses some local state `x`, initialised to 2, which is copied to the local variable `y`, after which its integrity is tested with an assert statement. This assert statement appears entirely trivial. However, recall that `x` is statically allocated and that `f` may be called multiple times. Each invocation of `f` may therefore leave a copy of `x`'s address on the stack. Subsequently, if a caller can read `f`'s old stack frame then it may break the integrity of `x` in-between calls. This dangling stack attack is an additional threat in low level languages where callers may create activation records containing dangling stack pointers. Furthermore, dangling stack pointers allow callees to use the stack as a covert channel to communicate with callers, which becomes an issue in attacker models with multiple mutually mistrusting components, in which we want to prevent the communication between two different attackers.

We will distinguish the absence of dangling stack pointers property from a slightly different notion of temporal stack safety, which we call *temporal confidentiality*, and which can be thought of as the temporal aspect of local state confidentiality. Consider two programs `f` and `g` whose only difference is to leave different traces on their respective stack frames, e.g., as in Listing 4.8. Then, as long as temporal confidentiality is enforced, no caller should be able to distinguish `f` from `g`. We remark that the complete absence of any dangling stack pointer (passed down or otherwise) implies temporal confidentiality, without having to clear any parts of the stack.

4.2.2.2 Stack Safety in the Presence of Stack Objects

We now explain how *stack objects* may influence stack safety properties. In prior work on local capability machines, stack objects have largely been ignored. However, they have a significant impact on the guarantees provided by the capability machine. Disallowing stack objects altogether is too restrictive as it is a common programming idiom in C-like languages to pass stack references as arguments.

Let us consider what happens to local state integrity in the presence of stack objects. Consider for instance the example in Listing 4.9 in which `f` receives an input from a caller and passes it along with its own stack object to its callee. In this scenario, neither `f`'s caller, nor `g` are trusted. In fact, they may collaborate to break `y`'s integrity. Indeed, if no precaution is taken by `f`, it may be possible that the stack object passed by its caller actually possesses *write* authority on `f`'s stackframe, which could be abused by `g`.

4.2.3 Enforcing Stack Safety in Capability Machines

Let us recap the stack safety properties we have isolated thus far. (1) *Local state integrity* (LSE integrity) guarantees that a callee cannot break the integrity of local stack frames (Listings 4.1 and 4.9), and that neither the callee nor a caller can break the integrity of the private environment associated with a closure (Listing 4.2). (2) *Local state confidentiality* (LSE confidentiality) guarantees that the local stack frame cannot influence the behaviour of a callee (Listing 4.3), and that the private environment of a closure cannot influence the behaviour of a callee or a caller (Listing 4.4). (3) *Well-bracketed control flow* (WBCF) guarantees that a callee returns to its immediate caller in the call stack (Listing 4.5). (4) *Temporal stack safety* according to Tsampas et al. guarantees that a callee cannot return a dangling stack pointer (a property that is violated in Listing 4.6). We expand on that notion and additionally guarantee that no caller can restore a dangling stack pointer upon return either (Listing 4.7). Finally (5) *temporal confidentiality* guarantees that a caller is unable to read popped stack frames, or, in other words, that popped stack frames cannot influence the behaviour of the caller (Listing 4.8). Temporal confidentiality can also be interpreted as the temporal aspect of local state confidentiality.

Each of these properties can be investigated with or without the presence of stack-object parameters. It is in general easier to guarantee these properties by altogether disallowing stack objects. Passing a stack object from a caller to a callee is not safe, unless certain conditions are dynamically checked in between, in case an overlapping stack pointer (potentially breaking integrity) can be reached from that stack object.

Table 4.1: Guarantees granted by the calling convention.

	LOCAL + clear		LOCAL + U		LINEAR		TEMPORAL		DIRECTED + U	
	w\o	w	w\o	w	w\o	w	w\o	w	w\o	w
LSE integrity	✓	✗	✓	✗	✓[2]	✓[2]	N\A	N\A	✓	✗
LSE confidentiality	✓	✗	✓	✗	✓[2]	✓[2]	N\A	N\A	✓	✗
WBCF	✓	✗	✓	✗	✓	✓	N\A	N\A	✓	✗
Temp. confidentiality	✓	✓	✓	✓	✗	✗	N\A	N\A	✓	✓
Dangling stack	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

Together, these properties make up Anderson et al.'s notion of observable stack safety. In order for the calling convention of a capability machine to be fully stack safe, it must enforce each of these properties. Unfortunately, the current state of the art enforce them at a varying degree of efficiency. In particular, enforcing temporal stack safety appears always to come at a significant cost.

Table 4.1 relates previous work on capability machine stack safety and also the work presented in this paper to these properties, each in a situation where stack objects are not passed from caller to callee (w\o), and in a situation stack objects are allowed (w). A ✓ means that the property is guaranteed by the associated calling convention. The different calling conventions have more or less overhead, in terms of the amount of stack clearing required by the calling convention. A ✓ depicts a high overhead, on the order of the full stack size, a ✓ depicts a relatively low overhead, on the order of a single stack frame, and a ✓ depicts a low overhead of constant time. A ✗ means some additional check is needed to guarantee the property (it does not mean it is impossible to guarantee a given property, but rather that it requires some additional mechanism beyond the calling convention). Finally, N\A means that the property is assumed to hold given the granularity of the capability machine language. (The marks come from our understanding of the earlier work, supported by the various examples verified in each model.) The first column outlines the calling convention using LOCAL capabilities and full stack clearing [129]. The second column outlines the calling convention using uninitialized capabilities and partial clearing [55]. The third column outlines the calling convention using LINEAR capabilities [132], and the fourth column outlines a more high level language using TEMPORAL capabilities [144]. The rightmost column gives an overview of the novel calling convention using the DIRECTED capabilities we introduce in this paper.

We remark that the LINEAR column shows a calling convention that checks many of the boxes and, in fact, we conjecture that a (LINEAR + uninitialized)-based calling convention could check all boxes. Thus the reader may wonder why we introduce a new kind of capability and a new calling convention here. There are several reasons: first, linear capabilities can be cumbersome to use, as only the top part of a stack frame can be passed as parameters. Second, exceptions cannot be implemented efficiently.

[2] While we mark the StkTokens calling convention [132] as enforcing LSE as the authors claim, it is actually unclear whether it does protect more than just the local stackframe as done in the other works [55, 129]. Indeed, the calling convention does not seem to prevent one from leaving a capability to some private state on the stack and returning without clearing the stackframe.

Third, and most importantly, LINEAR capabilities are expensive to realize in practice. Moving a LINEAR capability requires an atomic move which is believed by hardware developers to lead to an undesirable overhead in runtime [128, §3.6.2]. Indeed, this was also the reason why Georges et al. [55] considered local and uninitialized capabilities instead.

In fact, in order to reach full stack safety à la [10], there is a cost to each existing calling convention. The excessive stack clearing of [129] was improved upon in [55], however the latter only achieves temporal stack safety by clearing local stack frames upon return. Tsampas et al. [144] propose temporal capabilities as an enforcement mechanism to prevent dangling stack pointers. However, they would require an expensive amount of bits to represent.

In this paper, we propose DIRECTED capabilities as an *efficient* enforcement mechanism (both wrt. space and time complexity) of full stack safety. Section 4.3.3 presents the definition of DIRECTED capabilities, and in Sections 4.4 to 4.6 we show how DIRECTED capabilities can be used to enforce stack safety. Our calling convention does not use any stack clearing at all. Furthermore, DIRECTED capabilities can be efficiently realized in practice, requiring only one additional bit in the representation of capabilities, and with only one additional dynamic bounds check which is similar to existing ones (and hence efficient).

We define a unary model to reason about integrity properties, and a binary model to reason about confidentiality properties (including temporal confidentiality). We use these models to reason about small but challenging examples; we focus on examples that depend on properties not previously considered on a low level capability machine (integrity in the presence of stack objects, and temporal confidentiality). Furthermore, we follow the methodology presented by Skorstengaard et al. [132] and define an overlay semantics that clearly enforces each of the properties in Table 4.1. In Section 4.6.3, we show how our new calling convention is fully abstract with respect to this overlay semantics.[3]

4.3 Capability Machine: Operational Semantics and Calling Convention

In this section we present the operational semantics of the capability machine we consider. Our capability machine is based on the one by Georges et al. [55] and is, transitively, inspired by CHERI [154] and the M-Machine [27].

In Section 4.3.1, we first recall from Georges et al. [55] how the operational semantics for a capability machine with local and uninitialized capabilities is defined.

[3] In light of this full abstraction result, the reader may wonder why we also develop the logical relations models. The reason is that while the overlay semantics makes some properties obvious (e.g., popping stack frames upon return), it is not easy to use the overlay semantics for reasoning about concrete examples. This is not so surprising: even for high-level languages like ML, scientists have had to invent Kripke logical relations (and other kinds of) models to reason about local state encapsulation, e.g., [8, 39, 137].

$$\begin{array}{lclcl}
a & \in & \text{Addr} & \triangleq & [0, \text{AddrMax}] \\
p & \in & \text{Perm} & ::= & \text{O} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
 & \multicolumn{4}{l}{\mid \text{RWL} \mid \text{RWLX} \mid \text{URW} \mid \text{URWL} \mid \text{URWX} \mid \text{URWLX}} \\
g & \in & \text{Locality} & ::= & \text{GLOBAL} \mid \text{LOCAL} \mid \text{DIRECTED} \\
c & \in & \text{Cap} & \triangleq & \{(p,g,b,e,a) \mid b,e,a \in \text{Addr}\} \\
w & \in & \text{Word} & \triangleq & \mathbb{Z} + \text{Cap}
\end{array}$$

$$\begin{array}{lclcl}
r & \in & \text{RegName} & ::= & \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \\
reg & \in & \text{Reg} & \triangleq & \text{RegName} \to \text{Word} \\
m & \in & \text{Mem} & \triangleq & \text{Addr} \to \text{Word} \\
\varphi & \in & \text{ExecConf} & \triangleq & \text{Reg} \times \text{Mem} \\
\delta & \in & \text{ExecMode} & ::= & \mathsf{Executable} \mid \mathsf{Halted} \mid \mathsf{Failed}
\end{array}$$

$$\begin{array}{lcl}
\rho & \in & \mathbb{Z} + \text{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
 & & \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt} \mid \\
 & & \texttt{loadU}\ r\ r\ \rho \mid \texttt{storeU}\ r\ \rho\ \rho \mid \texttt{promoteU}\ r
\end{array}$$

Figure 4.4: Machine words, machine state and instructions.

EXECSTEP

$$(\mathsf{Executable}, \varphi) \to \begin{cases} [\![decode(z)]\!](\varphi) & \text{if } \varphi.\text{reg}(\mathsf{pc}) = (p,g,b,e,a) \\ & \quad \wedge\ b \leq a < e \\ & \quad \wedge\ p \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \\ & \quad \wedge\ \varphi.\text{mem}(a) = z \\ (\mathsf{Failed}, \varphi) & \text{otherwise} \end{cases}$$

Figure 4.5: Operational semantics: reduction steps.

Then, in Section 4.3.2, we further recall how said capabilities can be used to enforce LSE and WBCF via a secure calling convention. We then add support for directed capabilities in Section 4.3.3, and present our new improved calling convention, which can finally *efficiently* guarantee temporal stack safety, in Section 4.3.4. Figures 4.4 to 4.8 summarize the operational semantics; components marked in blue are for the novel directed capabilities and will be detailed in Section 4.3.3.

Figure 4.4 describes the syntax of our capability machine. We model a capability machine with finite memory. The set of addresses Addr is defined as the interval $[0, \text{AddrMax}]$, where AddrMax is the top address and cannot be dereferenced. Memory contains machine words w that are represented by either an (unbounded) integer or a capability. A capability is a quintuple (p, g, b, e, a) representing the authority to exert the permission p over the memory range $[b, e)$ and currently pointing to a.

4.3.1 A Capability Machine with Local and Uninitialized Capabilities

A permission p can either be opaque (O), enter (E), read-only (RO), read-execute (RX), read-write (RW), read-write-execute (RWX), read-write-local (RWL), read-write-local-execute (RWLX), or uninitialized-RW (URW), uninitialized-RWL (URWL), uninitialized-RWX (URWX), uninitialized-RWLX (URWLX). Permissions form a lattice as illustrated in Figure 4.6. The permissions RO, RX, RW, RWX are standard. Permission O provides no authority. Enter (E) capabilities represent opaque closures encapsulating code and data. As such, they cannot be read, written to, executed nor modified. They can only be jumped to, which will load them into the program counter register and unseal them into a RX capability. Their usage will be further illustrated when describing the operational semantics and the calling convention. Locality g is either GLOBAL or LOCAL, and forms a lattice as illustrated in Figure 4.6. LOCAL capabilities are meant to represent stack derived capabilities, while GLOBAL ones represent heap derived ones. They will be described further in Section 4.3.2. Write-local permissions (RWL and RWLX) are similar to their regular counterparts, but additionally provide the authority to write LOCAL capabilities to memory. That is, a regular RW capability cannot be used to write a LOCAL capability to memory, only GLOBAL ones. Finally, uninitialized capabilities Uπ represent a form of use-after-write authority: they provide permission π over the range $[b, a)$ and write permission on range $[a, e)$ — the boundary is automatically increased when the capability is used to write at a.

Machine instructions i operate over registers or constants and their behaviour will be detailed later. A register is either the program counter pc or a general purpose register $\mathsf{r}_0, \mathsf{r}_1, \ldots$. A machine state is composed of a mode describing whether the machine is in an Executable state, or in a terminal Failed or Halted state, and an execution configuration. An execution configuration is a pair of a register file, mapping registers to their values, and a memory state, mapping addresses to their values. The operational semantics of the machine shown in Figure 4.5 is given in smallstep style and has only one rule: if the state is Executable and the program counter contains an in-bounds executable capability pointing to some machine word, then an instruction

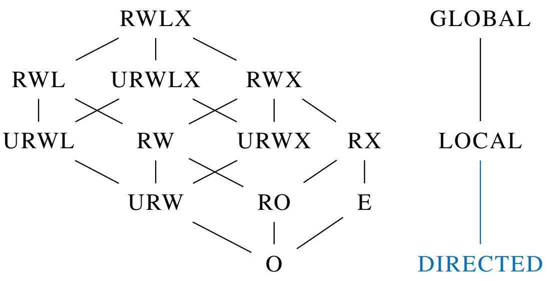


Figure 4.6: Permission and locality hierarchy.

is decoded and executed; otherwise the machine fails. Recall that, in general, failing is considered safe since it crashes the machine before anything unsafe occurs.

The semantics of instructions is given in Figure 4.8, with auxiliary definitions given in Figure 4.7. Most instructions increment the program counter at the end of their execution, except for branching and terminating instructions. This process is defined by updPC, which fails if the program counter does not contain an executable capability. Terminating instructions `fail` and `halt` change the machine state respectively to a Failed and Halted state. The `move` instruction copies a machine word in a register. The `load` instruction reads from the memory into a register, provided that a capability with sufficient authority is given. Similarly, the `store` instruction is used to write to memory. Additionally, if the word that is stored is a local capability, it must be that the capability provided has write-local authority. The `jmp` instruction copies a word to the program counter, additionally, if the word is an enter capability, it is unsealed into a RX one. The instructions `restrict` and `subseg` are used to decrease the authority of a capability. The former decreases the permission and the locality of a capability following the lattice given in Figure 4.6. The latter decreases the range of authority of a capability. The `lea` instruction is used to change the current address a capability points to. As explained earlier, an enter capability cannot be modified, and `subseg` and `lea` will thus fail. Furthermore, as the current address of an uninitialized capability indicates the boundary between where its regular authority applies and where it can only write, it is only safe to decrease the current address using `lea`. The instructions `getp`, `getl`, `getb`, `gete` and `geta` can be used to retrieve respectively the permission, locality, base address, end address and current address of a capability. `loadU` and `storeU` are similar to their regular counterparts, but operate only with uninitialized capabilities. An additional offset parameter is provided in order to be able to access the range that has already been written to. Moreover, if the offset provided is 0, then the boundary of the capability is incremented by `storeU`. The `promoteU` instruction can promote an uninitialized capability to its regular counterpart by discarding the memory range that has not been written to yet.

$$\text{updPC}(\varphi) =$$

$$\begin{cases} (\mathsf{Executable}, \varphi[\text{reg}.\mathsf{pc} \mapsto (p,g,b,e,a+1)]) & \text{if } \varphi.\text{reg}(\mathsf{pc}) = (p,g,b,e,a) \\ & \text{and } \textsc{rx} \not\preccurlyeq p \\ (\mathsf{Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\text{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\text{reg}(\rho) & \text{if } \rho \in \text{RegName} \end{cases}$$

$$\text{canReadUpTo}(w) = \begin{cases} \bot_{\textsc{Addr}} & \text{if } w \in \mathbb{Z} \\ \min(a, e) & \text{if } w = (\textsc{u}\pi, \_, \_, e, a) \\ e & \text{if } w = (\pi, \_, \_, e, \_) \end{cases}$$

Figure 4.7: Operational semantics: auxiliary definitions.

4.3.2 A Secure Calling Convention using Local and Uninitialized Capabilities

We now give an intuitive account of how the calling conventions of Skorstengaard et al. [129, 131] and Georges et al. [55] enforce local state encapsulation and well-bracketed control flow.

The calling convention of Skorstengaard et al. [129] uses local capabilities (not uninitialized capabilities) and requires that a program is initially provided with a stack capability with authority over the whole stack in a register r_{stk} when executed. Assume the following scenario where Alice calls Bob who calls Claire. We explain how Bob can protect himself from both Alice and Claire using the calling convention of Skorstengaard et al. [129].

Bob expects to receive a stack capability from Alice to build his own stack frame. Similarly, when calling Claire, Bob needs to provide the stack capability to her. However, in order to enforce local state encapsulation, it is necessary that Bob does not provide access to his own stackframe to Claire. Thus, when calling Claire, Bob restricts the stack capability to the unused part, using the `subseg` instruction, and then passes it to her. However, Bob needs to be able to restore access to his own stack frame upon return. He can do that using an enter capability: Bob constructs a return capability as an enter capability that restores the local environment when jumped to. This return capability can be safely passed to the callee Claire as its contents cannot be read, but can only be jumped to. This suffices to protect Bob's private state from Claire, but it is not enough to enforce WBCF. Indeed, on its own, this does not prevent the attack explained in Section 4.2.1.2, in which a callee keeps a copy of a previous return capability beyond its "lifetime". To prevent this kind of attack, Skorstengaard et al. use local capabilities: The stack capability is made write-local, executable (RWLX) and LOCAL, and all other (heap) capabilities are non write-local. This guarantees that if the return capability is built on the stack (and therefore LOCAL),

i	$[\![i]\!](\varphi)$	Conditions
`fail`	$(\mathsf{Failed}, \varphi)$	
`halt`	$(\mathsf{Halted}, \varphi)$	
`move` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho)$
`load` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,g,b,e,a)$ and $w = \varphi.\mathrm{mem}(a)$ and $b \leq a < e$ and $p \in \{\mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}, \mathrm{RWL}, \mathrm{RWLX}\}$
`store` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $b \leq a < e$ and $p \in \{\mathrm{RW}, \mathrm{RWX}, \mathrm{RWL}, \mathrm{RWLX}\}$ and $w = \mathrm{getWord}(\varphi, \rho)$ and if $w = (\_, \mathrm{LOCAL}, \_, \_, \_)$, then $p \in \{\mathrm{RWLX}, \mathrm{RWL}\}$ and if $w = (\_, \mathrm{DIRECTED}, \_, \_, \_)$, then $p \in \{\mathrm{RWLX}, \mathrm{RWL}\}$ and $\mathrm{canReadUpTo}(w) \leq a$
`jmp` r	$(\mathsf{Executable},$ $\varphi[\mathrm{reg.pc} \mapsto \mathit{newPc}])$	if $\varphi.\mathrm{reg}(r) = (\mathrm{E},g,b,e,a)$, then $\mathit{newPc} = (\mathrm{RX},g,b,e,a)$ otherwise $\mathit{newPc} = \varphi.\mathrm{reg}(r)$
`restrict` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $(p',g') = \mathrm{decodePermPair}(\mathrm{getWord}(\varphi,\rho))$ and $(p',g') \preccurlyeq (p,g)$ and $w = (p',g',b,e,a)$
`subseg` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi,\rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \mathrm{E}$ and $w = (p,g,z_1,z_2,a)$
`lea` $r\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $z = \mathrm{getWord}(\varphi,\rho)$ and $p \neq \mathrm{E}$ and $w = (p,g,b,e,a+z)$ $p = \mathrm{U}$-, then $z \leq 0$
`geta` $r_1\ r_2$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto a])$	$\varphi.\mathrm{reg}(r_2) = (\_,\_,\_,\_,a)$
`loadU` $r_1\ r_2\ \rho$	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,g,b,e,a)$ and $p = \mathrm{U}$- and $\mathit{off} = \mathrm{getWord}(\varphi,\rho)$ and $b \leq a + \mathit{off} < a \leq e$ and $w = \varphi.\mathrm{mem}(a + \mathit{off})$
`storeU` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\varphi'$ $[\mathrm{mem}.(a+\mathit{off}) \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $p = \mathrm{U}$- and $\mathit{off} = \mathrm{getWord}(\varphi,\rho_1)$ and $w = \mathrm{getWord}(\varphi,\rho_2)$ and if $w = (\_,\ell,\_,\_,\_)$ and $\ell \neq \mathrm{GLOBAL}$ then $p \in \{\mathrm{URWLX}, \mathrm{URWL}\}$ and $b \leq a + \mathit{off} \leq a < e$ and if $\mathit{off} \neq 0$ then $\varphi' = \varphi$ else $\varphi' = \varphi[\mathrm{reg}.r \mapsto (p,g,b,e,a+1)]$ and if $\ell = \mathrm{DIRECTED}$, then $\mathrm{canReadUpTo(w)} \leq a + \mathit{off}$
`promoteU` r	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,g,b,e,a)$ and $p = \mathrm{U}\pi$ and $w = (\pi,g,b,\min(a,e),a)$
$\ldots$		
$\_$	$(\mathsf{Failed}, \varphi)$	otherwise

Figure 4.8: Operational semantics: instruction semantics.

then the only place Claire can keep a copy of a return capability, is on the stack itself. Consequently, *by clearing the stack* before passing it to Claire, Bob can be sure that she will not be able to recover a previously left copy of a return capability. Finally, to protect himself from Alice, Bob also clears the whole stack and the registers before returning, so that Alice cannot access anything. Skorstengaard et al. later point out that it is sufficient for Bob to only clear his own stack frame, as anything that Claire may leave on the stack either originally came from Alice, or is a return capability from Bob, and, as Bob clears his own stack frame, using the return capability will only lead to cleared data. As briefly mentioned in Section 4.2, the original calling convention by Skorstengaard et al. [129] enforces temporal confidentiality with an excessive amount of clearing. With the optimization by Skorstengaard et al. [131], one only clears one's own stack frame on return. Although this improves the efficiency of the calling convention, every secure closure must clear its own stack frame upon return.

Furthermore, even with the optimization mentioned by Skorstengaard et al. [129], Bob still needs to clear the whole stack before being able to call Claire safely, and hence the calling convention is still very inefficient. To address this issue, Georges et al. [55] proposed to make the stack capability into an uninitialized capability. By passing an uninitialized URWLX stack capability to Claire, we are guaranteed that Claire cannot read anything left on the stack, without overwriting it beforehand, and thus there is no need to clear the whole stack before calling Claire. However, Bob still needs to clear his own stack frame before returning, as there is no guarantee that Alice did not keep a "fully initialized" stack capability, which would allow her to read leftover data on the stack. For this calling convention, Georges et al. [55] prove that LSE and WBCF are enforced; however dangling stack pointers are still a possibility, and thus some clearing is still needed in order to guarantee temporal confidentiality.

4.3.3 Directed Capabilities

We now introduce a novel kind of directed capabilities and then explain, in Section 4.3.4, our new improved calling convention, which relies on directed capabilities to efficiently guarantee LSE, WBCF and temporal stack safety (as we prove in later sections).

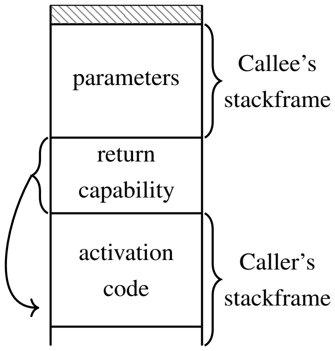
The intention of directed capabilities is to restrict where they can be stored in memory. This is done by adding a new locality DIRECTED, as illustrated in the locality lattice in Figure 4.6. To write a DIRECTED capability to memory it is then necessary to have permit-write-local authority (similarly to writing LOCAL capabilities to memory), as shown in the operational semantics of `store` and `storeU` in Figure 4.8. The distinguishing feature of a directed capability is that it cannot be stored "below" where it can read memory up to. That is, for a directed capability with a *regular* permission (i.e., not uninitialized) with authority over range $[b,e)$, it can only be stored at an address a such that $e \leq a$. For an uninitialized directed capability $(\mathrm{U}\pi, \ell, b, e, a)$, the part $[a,e)$ can only be written to, therefore it can only be stored at an address a' such that $a \leq a'$. The intuition is that, for a stack that grows upwards, the address a stack capability can read up to implicitly approximates the lifetime of the capability. Given two directed capabilities, if the first can read at a lower address than the second, then the first is owned by an "older" stack frame than the second and has thus a longer lifetime.

We remark that, from an hardware implementation point of view, directed capabilities should be quite easy to implement. First, uninitialized directed capabilities require only two additional bits, one for the directed locality, and one indicating whether it is uninitialized. Since increased pointer size can severely affect performance, CHERI Concentrate [162] employs a rigorous compression scheme to achieve realistic performance. Within this scheme, 2 and 7 bits are reserved for future use in the CHERI-64 and CHERI-128 respective compression formats. The two necessary additional bits are thus already available in either format. We can contrast that to temporal capabilities [144], which require n bits to encode the lifetime information for a call depth of size 2^n. The required number of bits is thus unbounded, and it is unclear how to determine the ideal least number of bits. Tsampas et al. discuss this exact point, and propose various workarounds. Directed capabilities, on the other hand, already fit within existing formats.

Second, directed capabilities only change the semantics of `load(U)` and `store(U)` by adding an extra bounds check. The added bounds check is no different from current checks, and we expect existing optimisation patterns, such as parallelisation, to apply.

4.3.4 A New Calling Convention using Directed Capabilities

Let us revisit the example in Section 4.3.2, now assuming that the stack capability is directed instead of local. When Bob is called by Alice, Bob can ensure that Alice did not keep a capability with *read* authority on the unused part of the stack by checking that the return capability that he received is not "above" the stack capability he received. This check can be avoided if the return capability is passed on the stack as part of the calling convention. Indeed, if the return capability is stored at some address a', Bob knows by property of directed capabilities that Alice can only have kept a copy of the stack capability with read authority at most up to a'. Thus, Bob will not have to clear his stackframe on return. On the other hand, if Bob passes some stack references to Claire as parameters, Claire will not be able to store anything from her own stackframe in them, thus avoiding the issues described in Section 4.2.2. In fact, Bob can take advantage of this property to ensure that Claire only returns safe values. By passing a stack capability rather than a dedicated register as the return value destination, Bob knows that any return value cannot grant read authority over popped stack frames. The calling convention assumes this strategy, and clears all general purpose registers upon return.



To sum up, our new calling convention is as follows; see also the figure above, which shows what the stack is expected to look like just after a call.

At program start-up. A directed URWLX stack capability is in register r_{stk}.

When called by an adversary. Check that the received stack capability is a capability of the form
$(\text{URWLX}, \text{DIRECTED}, b, e, a)$, the return capability is expected to be stored at address b.

Before calling an adversary. Push activation record to the stack and create a directed E-capability to use as return capability. Subseg the stack capability to the unused part. Push the return pointer on the stack, as well as all parameters. Clear all registers except r_{stk} and the program counter before calling.

Before returning to an adversary. Clear all general purpose registers.

Let's consider the cost of one secure call. Our new calling convention does not require any memory clearing, and thus incurs a constant overhead, mainly of clearing registers. Register clearing can be done efficiently using the CClearRegs instruction [158]. On the other hand, memory clearing is a costly operation. Previous calling conventions based on LOCAL capabilities all require some amount of stack clearing. Although Georges et al. [55] improves the situation by only clearing the local stack frame, the calling convention still has an undesired linear cost in the amount of stack memory used.

In summary, we present a faster calling convention, which can realistically be implemented in hardware. Moreover, the calling convention enforces all desired stack safety properties as is proved in the next sections.

4.3.5 Discussion

While directed capabilities lead to an efficient calling convention, the questions remains whether their new restrictions render them impractical otherwise. Specifically, do directed capabilities hinder critical C idioms, or compiler optimizations to any significant extent. A full investigation of this question is beyond the scope of this paper. However, we conjecture that directed capabilities are sufficiently permissive in practice, and in some cases more practical than, e.g., linear capabilities.

Some objects cannot be allocated on the stack. For instance, a locally allocated circular linked list breaks the directed property. Similarly, any locally allocated node cannot be added to an ambient heap allocated linked list. Indeed, nor should they: stack objects and heap objects often have different lifetimes and are thus generally incompatible.

We do not expect uninitialized capabilities to have major impact on code generation, though one must be careful to initialize an uninitialized object in "increasing" order. Similarly, it takes linear time to allocate and pass stack objects, while C assumes constant time allocation. However, we hasten to point out that it is in general only safe to pass stack objects for which the previous contents cannot be read. [117] enforce this efficiently using a lazy tagging and clearing scheme; it would interesting to investigate a similar scheme for uninitialized capabilties.

Furthermore, a compiler must also be careful with the order of stack allocations. Consider for instance the following code snippet: **int**∗ x; **int** y; x = &y. In this code, the compiler must reorder the allocations for *x* and *y* to guarantee the directed property. Such considerations must also be taken into account when implementing compiler optimizations.

All in all, directed capabilities are more restrictive (by design) than local capabilities, but we argue they remain more practical than linear capabilities. For instance, it is a common idiom in C to pass pointers as an argument but not return them to the caller. With linear capabilities, a compiler would need to ensure that all linear capabilities be returned when used in this fashion. Furthermore, some library functions such as

```
int memcmp(const void* p1, const void* p2, size_t size)
```

are not implementable in an linearity-friendly way, since p1 and p2 are allowed to be aliases.

We leave a full practical evaluation of directed capabilities to future work.

4.4 A Unary Model for Integrity

In this section, we develop a novel model that captures all the guarantees provided by DIRECTED capabilities and our associated calling convention. The core novelty lies in its ability to express temporal stack safety. The model is made up of two components; a program logic to reason about known and trusted code, and a Kripke logical relation to reason about arbitrary untrusted code. We use the unary model to reason about the *integrity* of example programs.

The program logic is a variant of the one by Georges et al. [55]; the main change is that some proof rules have been updated to account for directed capabilities, following the operational semantics defined in Section 4.3.1. Thus we refrain from describing the program logic in detail.

Here it suffices to know that the program logic is defined using Iris' weakest preconditions [77] (which means that we can re-use the Iris program logic infrastructure to reason formally in Coq) and that the weakest precondition predicate $\mathsf{wp}\,\mathsf{Executable}\,\{v, Q(v)\}$ intuitively means the program pointed to by the program counter can execute without getting stuck, and that if it terminates, then $Q(v)$ is guaranteed to hold for some final mode v, which can be either Halted or Failed.

Thanks to the dynamic checks of the capability machine, the behaviour of a program is limited by the capabilities it has access to. Thus, even completely arbitrary code must adhere to rules imposed by the capability machine, and will satisfy some notion of *capability safety*. The logical relation captures this notion of capability safety, and serves as a contract between trusted and untrusted code. The fundamental theorem of logical relations (see below) means that any word that is safe to read satisfies that contract. As long as arbitrary code is just a list of instructions (and thus does not, e.g., include an embedded capability), a corollary then states that even completely arbitrary code satisfies that contract.

Our logical relation is an extension of the one by Georges et al. [55] and our presentation focuses on the key challenge, which is to extend the model, in particular, the Kripke worlds, to capture the enforcement of temporal properties qua directed capabilities. The execution of a program depends on the physical state of memory. Since we want to reason about stack safety, we are particularly interested in the state of the stack. During execution, different parts of the stack are in different states (e.g. used, unused, etc.). Following Georges et al., we use a Kripke world to model the abstract state of the stack. In essence, the Kripke world is an abstraction of physical memory. Concretely, it tracks which parts are in heap space, which parts are in stack space, and at what particular state a location is. The stack may change in ways that accord with the specific properties we attribute to stack safety. We capture stack-based properties by carefully describing the possible changes to the Kripke world so that they reflect the expected behaviour of the physical stack.

From a technical point of view, a WORLD has two parts: W^{std} maps addresses to so-called *standard states*, governing *shared* regions, such as the stack; and W^{cus} maps a countably infinite set of region names to custom state transition systems, governing *owned* regions, such as the private environment of closures.

The logical relation imposes certain invariants on regions of memory that is shared between trusted and untrusted code. Those invariants depend on the state of the stack, and thus on the Kripke world that represents it. We use Iris ghost state to track not only the physical machine state, but also what we call the *instrumented machine state*, which captures the connection between the physical memory, and the abstract state of memory. It uses an Iris predicate called the *standard resource*. A standard resource has two functions: (1) it associates an address of a shared region of memory, in particular each stack address, to its physical state, and (2) it associates that physical state to an invariant. The invariant may depend on the current state of the Kripke world. Our model extends Georges et al. [55] insofar as it uses the same structure for the instrumented machine state. However, in order to capture temporal properties, we define a *novel* Kripke world, upon which we build new definitions for the standard

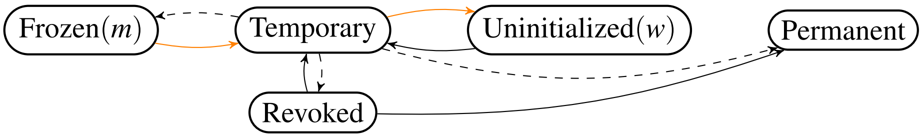


Figure 4.9: Standard State Transition System. Full lines indicate public transitions, dashed lines indicate private transitions, orange lines indicate temporal transitions.

resources. In this paper, we focus a large part of our attention on the new Kripke world.

In the remainder of this section we first describe the standard states we use to capture stack and heap states, and how these states may evolve, such that they can be used to capture the desired stack properties (Section 4.4.1). In Section 4.4.2, we will then present the logical relation itself. Finally, we end this section with two examples highlighting what kind of programs we can now verify using our model (Section 4.4.3).

4.4.1 A New Kripke World

The standard states represent the various states a shared memory address can be in. An address is shared if it lies within the region of authority of a capability that crosses the boundary between known and arbitrary code. The physical state of these addresses will be imposed by invariants, which may in turn depend on the current state of the rest of the machine. For instance, the invariant of a heap region should not be able to depend on the state of the stack and its changes, since locality dictates that the heap is unable to contain stack pointers. Likewise, the invariant of a stack address connected to a lower stack frame will be different from the invariant of the currently live, or popped stack frame. The standard states must therefore also reflect the very specific lifetime properties of a stack frame.

We now explain each of the standard states. The Permanent state represents an allocated heap region. As soon as a heap region is allocated and shared, it becomes permanent. There is no mechanism to free the heap region from its state. The Temporary state represents a live stack frame, i.e., the stack frame owned by the currently executing function. Specifically, it will refer to the *readable* parts of the stack shared between calls. A live stack frame does not need to be Temporary at every step of execution. Rather, the Temporary state is meant to represent the live parts of the stack at the point of change of control. The Uninitialized state represents the unused part of the stack, i.e., a Freed frame or some part of the stack that has never been used. In general, the Uninitialized state will simply refer to the parts of the stack that cannot be read from, only written to. The Frozen(m) state represents a frozen stack frame, i.e., the parts of a frame not shared with a new callee. Here m maps the addresses of that particular frame to their frozen values. The Revoked state represents a part of the stack which is currently owned by an executing function. A region is not part of the shared stack while it is Revoked.

Next we define how a standard state may evolve in order to reflect physical stack changes that accord with the key properties of stack safety (LSE, WBCF, and temporal stack safety). Some stack changes are not observable by any caller or callee, whereas other changes are public and observable by both. As long as temporal stack safety is enforced, some stack changes (such as popping a stack frame) are safe to observe by the caller, but not by higher stack frames. After all, it is not safe to pop a frame if there are still live frames on top of it.

We use three kinds of transitions to reflect these distinctions. A transition is *observable* whenever some entity is oblivious to that transformation. Public transitions (depicted in Figure 4.9 as straight black lines) are those that are observable by all functions. Private transitions (dashed lines) can only be observed by the currently executing function. Finally, temporal transitions (orange lines) are only observable by functions that are still present on the call stack.

Using these transitions, we isolate three future world relations. A world W' is a public future world of W, $W' \sqsupseteq^{pub} W$, if all the states in W' are either connected to a fresh address or region name, or were updated from W through public transitions only. A private future world $W' \sqsupseteq^{priv} W$ allows for public, temporal or private transitions. Finally, the third relation we consider is in fact a family of future world relations, where each relation is indexed by an address. We say that W' is a future world of W relative to address a, written as $W' \sqsupseteq^{a} W$, when the state of all addresses *below* a were updated via public transitions only, while addresses *at or above* a were updated via public or temporal transitions.

For instance, consider an address a' that is Temporary in W. If that address lies below a, it must still be Temporary in W' if $W' \sqsupseteq^{a} W$. However, if that address lies at or above a, it may change to an uninitialized state. Likewise, any address at or above a with an Uninitialized(w) state in W may change to a new Uninitialized(w') state.

A relative future world relation captures the changes to a stack *relative* to a specific stack frame (delimited by its upper bound address). From the point of view of a particular stack frame, a new call will push then pop new stack frames, whereas that stack frame *remains* frozen or initialized. Upon return of a well-bracketed call, a stack frame is safe to pop. In other words, invoking a return capability should be safe to do in a world where the current as well as all higher stack frames are uninitialized. In $W' \sqsupseteq^{a} W$, world W' represents such a world, from the perspective of a stack frame with upper bound a.

The instrumented machine state imposes monotonicity requirements on the invariants associated with shared addresses[4]. If a shared address a is part of the heap, then the associated invariant must be monotone with regards to $\sqsupseteq^{priv}$. On the other hand, if a is part of the (live) stack, then the associated invariant must be monotone with regards to $\sqsupseteq^{a}$.

We finish this section by highlighting some interesting transitions, relating them to the corresponding physical state changes of the stack:

[4] The formal definitions of the instrumented machine state and standard resources are here omitted for brevity, full definitions can be found in the Coq formalisation.

- Temporary $\dashrightarrow$ Frozen(m): local variables freeze when their associated function makes a new call. These local variables are stored in a stack frame. The frozen part of a stack frame cannot be written to while it is frozen. A stack frame cannot stay frozen forever. In particular, a frozen stack frame must thaw when control is returned to its caller. A caller should therefore not be able to observe that the stack frame was at any point frozen during execution, hence the private transition.
- Frozen(m) $\longrightarrow$ Temporary: as indicated in the previous point, a frozen stack frame must thaw before it can be written to again. A frozen stack frame is thawed only after a callee invokes the callback. Temporal stack safety dictates that only higher stack frames can invoke the callback. Invoking the callback effectively pops the callee's stack frame. Thus a local stack frame is thawed once all higher stack frames have been popped, but has no effect on lower stack frames.
- Temporary $\longrightarrow$ Uninitialized(w): Finally, local stack frames are popped upon return. Thanks to temporal stack safety, popping a stack frame has no effect on lower stack frames, since they cannot read its content.

In summary, we have presented the standard states and transitions that make up the new Kripke world used to model LSE, WBCF and now also temporal stack safety. The world differs from [55] in the following way: we distinguish between the Uninitialized state (shared write access but no shared read access) and the Frozen state (no shared write or read access), and we introduce a new kind of transition for defining a relative future world relation, capturing the temporal properties of the stack.

4.4.2 A Unary Logical Relation

Figure 4.10 defines the unary Kripke logical relation with support for temporal stack safety. We depict in blue the parts of the definition that are different from the unary logical relation used by Georges et al. [55].

The value relation $\mathscr{V} : \mathsf{WORLD} \to \mathsf{Word} \to \mathit{iProp}$ defines the validity of a word relative to a world; the register relation $\mathscr{R}$ defines the validity of a register state; and the expression relation $\mathscr{E}$ defines when it is is safe to use a word as the program counter. Note that all relations are defined in the Iris program logic (cf. the type *iProp* for Iris propositions). We now explain the definition, and first consider the value relation. Integers and words with O capability are always valid. A word with a read and/or write permission is valid in a world W only if certain conditions on W are met. A GLOBAL capability with a read and/or write permission imposes a Permanent state on its range of authority. An uninitialized capability with a write-local (i.e., it can be used to store LOCAL and DIRECTED words) permission imposes a Temporary state on its initialized readable range of authority, whereas its uninitialized part may be either Temporary or Uninitialized. The state relations $\mathscr{S}$ and $\mathscr{S}^u$ define the exact conditions on W for regular and uninitialized capabilities respectively.

A valid capability with read and/or write permission grants access to the so-called standard resources alluded to in the beginning of this section: $rel(a, \phi)$ associates the memory predicate $\phi : \mathsf{WORLD} \to \mathsf{Word} \to \mathit{iProp}$ to the address a. It suffices

$$\boxed{\mathscr{E}(W)(v)} \triangleq \forall reg, \mathscr{R}(W)(reg) * sharedResources(W) * stsCollection(W) * \mathsf{pc} \mapsto v * \mathop{\Large *}_{(r,w)\in reg/\mathsf{pc}} r \mapsto w \mathrel{-\!\!*} \mathsf{wp\ Executable} \left\{ \begin{array}{l} v, v = \mathsf{Halted} \rightarrow \exists W'\, reg',\ W' \sqsupseteq^{priv} W \\ * sharedResources(W') \\ * stsCollection(W') \\ \mathop{\Large *}_{(r,w)\in reg'} r \mapsto w \end{array} \right\}$$

$$\boxed{\mathscr{R}(W)(reg)} \triangleq \mathop{\Large *}_{(r,w)\in reg/\mathsf{pc}} \mathscr{V}(W)(w)$$

$$\boxed{\mathscr{V}(W)(w)} \begin{cases} \mathscr{V}(W)(z) & \triangleq \top \\ \mathscr{V}(W)(\mathrm{O}, -) & \triangleq \top \\ \mathscr{V}(W)(p,g,b,e,a) & \triangleq \mathop{\Large *}_{a'\in[b,e)} \begin{cases} \mathscr{S}^u(W)(a',g,p,a) & \text{if } p = \mathrm{U}\text{-} \\ \mathscr{S}(W)(a',g,p) & \text{otherwise} \end{cases} \\ & \quad \wedge \begin{cases} \exists P, rel(a',P) * \mathrm{rcond}(P) & \textit{if } p \in \\ & \{\mathrm{RO}, \mathrm{RX}\} \\ rel(a', \mathscr{V}) & \text{otherwise} \end{cases} \\ \mathscr{V}(W)(\mathrm{E},g,b,e,a) & \triangleq \square\ \forall W' \sqsupseteq^{g} W, \triangleright\ \mathscr{E}(W')(\mathrm{RX},g,b,e,a) \end{cases}$$

$$\boxed{\mathrm{rcond}(P)} \triangleq \triangleright \square \forall W, w, P(W)(w) \mathrel{-\!\!*} \mathscr{V}(W)(w) * \triangleright \square \forall W_1, W_2, z, P(W_1)(z) \mathrel{-\!\!*} P(W_2)(z)$$

State relation

$$\mathscr{S}(W)(a,g,p) \triangleq \begin{cases} W^{std}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \\ & \wedge\ g = \mathrm{DIR} \\ W^{std}(a) = \text{Temporary} & \text{if write-local}(p) \\ & \wedge\ g = \mathrm{DIR} \\ W^{std}(a) = \text{Permanent} & \text{if } g \neq \mathrm{DIR} \end{cases}$$

$$\mathscr{S}^u(W)(a,g,p,mid) \triangleq \begin{cases} \mathscr{S}(W)(a,g,p) & \text{if } a \geq mid \wedge\ g = \mathrm{DIR} \\ \vee\ \exists w, W^{std}(a) = \text{Uninitialized}(w) & \\ \mathscr{S}(W)(a,g,p) & \text{if } a < mid \vee\ g \neq \mathrm{DIR} \end{cases}$$

Figure 4.10: A Logical Relation with Support for Temporal Stack Safety. $\sqsupseteq^g$ equals $\sqsupseteq^{priv}$ whenever g is GLOBAL or LOCAL, and $\sqsupseteq^e$ whenever g is DIRECTED. DIR. stands for DIRECTED

to think of $rel(a, \phi)$ as an invariant, that can be used to access the ghost state of a, while guaranteeing that ϕ holds at the current physical state of a in the current world W. Normally, the predicate we associate with such an address a would be $\mathcal{V}$ itself. However, we distinguish between a read-only and a read-write permission by allowing the associated predicate of an address within a read-only region to be stronger than $\mathcal{V}$. The predicate in question then needs to satisfy the read condition $\mathrm{rcond}(\phi)$, which imposes two restrictions on ϕ. First, it enforces that $\phi(W)(w)$ always implies validity, regardless of W and w. Second, it enforces that $\phi(W)(w)$ never depends on W when w is an integer. In other words, only capabilities can depend on the instrumented machine state. Notice how each condition is guarded by a later ($\triangleright$) modality; this is to guarantee that the definition of $\mathcal{V}$ is well defined (here we use that Iris supports the definition of guarded recursive predicates).

An E capability can only be jumped to, hence its validity is defined in terms of its safe execution. Such a capability can be jumped to at any moment and hence the property should be persistent (i.e., not rely on any ephemeral resources); this is expressed by Iris' persistence modality $\square$. The execution of a capability may depend on the current state of the stack. For instance, a GLOBAL E capability represents a global function closure, and is safe to jump to regardless of the state of the stack. On the other hand, a DIRECTED closure (used for return pointers) is only safe to jump to at the end of a function's execution. This distinction is made by quantifying over the possible future worlds an E capability may be invoked from, see $W' \sqsupseteq^g W$. A GLOBAL or LOCAL closure can be invoked in any private future world of W, whereas a DIRECTED closure can only be invoked in a relative future world $W' \sqsupseteq^e W$, where e will represent the upper bound of the stack frame being returned to.

Finally, the safe execution of a word is defined using the expression relation $\mathcal{E}$. The expression relation is defined in terms of the program logic that the logical relation is built upon. Specifically, $\mathcal{E}(W)(w)$ expresses that given an instrumented machine state beginning at world W, and a safe register state $\mathcal{R}(r)$, the word w is safe to execute. That is, the weakest precondition holds for a configuration in which the `pc` contains w, with a post condition that enforces the instrumented machine state, i.e., all the established invariants hold at some private future world.

4.4.2.1 Fundamental Theorem of Logical Relations

One of the key facts supporting that our logical relation models temporal stack safety is the following monotonicity lemma. It describes the monotonicity of $\mathcal{V}$ relative to a word's current read authority. It entails in particular that a stack capability that is valid in some world W is also valid in a relative-to-its-current-address-future world; and thus it does not depend on the contents of higher stack frames.

Lemma 20 (Address Relative Monotonicity)**.**

$(1)\ p \in \{\text{URWLX}, \text{URWL}, \text{URWX}, \text{URW}\} \wedge W' \sqsupseteq^{a} W \to$
$\quad \mathcal{V}(W)(p,g,b,e,a) \mathrel{-\!\!*} \mathcal{V}(W')(p,g,b,e,a)$
$(2)\ p \notin \{\text{URWLX}, \text{URWL}, \text{URWX}, \text{URW}\} \wedge W' \sqsupseteq^{e} W \to$
$\quad \mathcal{V}(W)(p,g,b,e,a) \mathrel{-\!\!*} \mathcal{V}(W')(p,g,b,e,a)$

Furthermore, the address relative temporal future world relation can be weakened by lowering the associated address, and the value relation is monotone wrt. private future worlds (for all non-directed capabilities).

Lemma 21 (Address Relative Weakening)**.** $a' \leq a \to W' \sqsupseteq^{a} W \to W' \sqsupseteq^{a'} W$

Lemma 22 (Private Monotonicity)**.** *Let w be a word that is not* DIRECTED*. Then* $W' \sqsupseteq^{priv} W \to \mathcal{V}(W)(w) \mathrel{-\!\!*} \mathcal{V}(W')(w)$.

Finally, we state the fundamental theorem of logical relations. We refer to the Coq mechanisation for its full proof, and present here a proof sketch.

Theorem 12 (FTLR)**.** *Assume that* $p = \text{RX}$, $p = \text{RWX}$ *or* $(p = \text{RWLX} \wedge g = \text{DIRECTED})$. *Assume also that* $\mathcal{V}(W)(p,g,b,e,a)$. *Then we have that* $\mathcal{E}(W)(p,g,b,e,a)$.

Proof. Upon unfolding the definition of $\mathcal{E}$, our goal is to prove that the weakest precondition holds for a program pointed to by the program counter (p,g,b,e,a), given the predicates of a safe register state. According to the underlying definition of Iris weakest preconditions, this amounts to showing that the program does not get stuck; either it executes the next instruction and continues, or the program attempts to reach outside its bounds of authority and subsequently crashes into a failed configuration.

The fundamental theorem is proved by Löb induction (the proof principle for guarded recursion), generalized for all W, p, g, b, e and a. The induction hypothesis states that the program will safely execute *later*, in other words after at least one step of execution. The fundamental theorem is thus proved by stepping through the execution of the next instruction, currently pointed to by a, (which may either fail or succeed), and then applying the induction hypothesis *one step later*.

First, we extract the resources needed for executing the next instruction. Since p has at least read authority over its range of authority, the assumption $\mathcal{V}(W)(p,g,b,e,a)$ grants access to the standard resources within the range $[b,e)$. In particular, if the program counter is valid, we can extract the standard resources for a, including the points-to predicate $a \mapsto w$ for some Word w. The proof then proceeds by case analysis on $decode(w)$.

Let us focus on the particularly interesting case `storeU` r_{dst} r_{src} 0 in which the register r_{dst} contains a stack capability, currently pointing to some address a_{stk}. In this case, we will be overriding some (uninitialized) word with a new possibly DIRECTED and valid word w_{src} from r_{src}. Since the offset is 0, the capability in r_{dst} will increment, thus initializing address a_{stk}. We will therefore have to update the standard resource

```
(closure creation around f1)
g1: malloc r2
    store r2 2
    move r3 pc
    lea r3 offset
    crtcls [r2] r3
    jmp r0
f1: prepstack rstk
    loadU r0 rstk (-1)
(intentional leak)
    push renv
    load renv renv
(integrity assertion)
    assert renv 2
    rclear RegName\{pc,r0}
    jmp r0
end:
```

Figure 4.11: Assembly of Listing 4.7

for a_{stk} to an initialized standard resource, now pointing to the word w_{src}. In particular, since this is a stack address, we will have to prove that w_{src} satisfies the conditions of a stack standard resource, namely that w_{src} is directed with regards to $\sqsupseteq^{a_{stk}}$. We first use Lemma 20, which asserts that w_{src} is directed relative to the upper bound of its own read authority. Storing a DIRECTED capability on the stack will dynamically check that this upper bound is *smaller* than the current stack address, in other words below a_{stk}. We can therefore apply Lemma 21 to assert that w_{src} is indeed directed with regards to $\sqsupseteq^{a_{stk}}$.

The remainder of the cases are similar. Instructions that interact with memory require invariants to be opened, whereas instructions that only change the register state will only have to establish the validity of updated registers. □

4.4.3 Examples

We show how to use the unary model to prove safety of two example programs: Listings 4.7 and 4.9 in capability machine code. These two examples illustrate two properties that have not been explored in previous formalizations. Each program uses an assert subroutine that tests the integrity of encapsulated state. Although we will not present it in this paper, we have also proved the safety of the awkward example (Listing 4.5), which can be found in the Coq mechanization.

4.4.3.1 Protection against Dangling Stack Pointers

Figure 4.11 depicts a program with an assertion whose success depends on the absence of dangling stack pointers. `g1` creates a closure around some code `f1` and a dynamically allocated location containing the integer 2. The macro `crtcls` [r_2] r_3 allocates a closure where r_3 points to the closure's code (created using the *offset* from `g1` to `f1`), and r_2 points to the newly allocated environment; the resulting closure is an enter capability. `f1` applies the calling convention from Section 4.3.4: (1) it prepares the stack by checking that the stack has permission URWLX and lowers its address to point to the bottom of its bound, and (2) it loads the return capability parameter which has been passed on the stack itself. Now the idea is that since the parameter was stored on the stack, it must either be a heap closure, or a stack allocated activation record of an *older* stack frame qua temporal stack safety. Thus, when `f1` attempts to leak the private capability of the closure by pushing a copy of the private capability onto the stack, temporal stack safety ought to ensure that the content of the stack frame cannot be read once popped and thus that the leaked capability remains inaccessible from the environment after we return from `f1`. Finally, `f1` clears the registers and returns to the caller by invoking the return capability that was passed on the stack. Note that `f1`'s stack frame is *not* cleared.

Recall that `f1`'s assertion hinges on the fact that any caller to the closure created by `g1` is not able to read the popped stack frame (Section 4.2.2.1). We use the unary model to prove that within any arbitrary context of a certain layout, the assertion flag associated with the assert subroutine stays at 0 at every step of the execution, meaning that the assertion never fails.

We prove this statement in two stages. First, we show that `g1` is safe according to the expression relation $\mathscr{E}$.

Lemma 23. *For any world W, assuming that the memory has been properly initialized in region $[b_{\text{temp}}, e_{\text{temp}})$ with the code of the program and a pointer to the* `malloc` *and* `assert` *subroutines, we have:*

$$\mathscr{E}(W)(\text{E}, \text{GLOBAL}, b_{\text{temp}}, e_{\text{temp}}, \mathtt{g1}).$$

Proof. The proof proceeds by applying the program logic rules, updating the physical machine state, while at the same time change the instrumented machine state when appropriate. We first reason about instructions $[\mathtt{g1}, \mathtt{f1})$. The `crtcls` subroutine creates a closure around the dynamically allocated region (say at address l, pointing to 2), and the code from `f1` to `end`. This closure is pointed to by a global enter capability, say $(\text{E}, \text{GLOBAL}, b, e, b)$, and exposed to the unknown context at `jmp` r_0.

To reason about the unknown continuation, we apply the fundamental theorem of logical relations (Theorem 12) on the jump target. To use the resulting weakest precondition, we must then show that the current register state is valid. More specifically, we must show that the created E-capability is in the value relation.

$$\mathscr{V}(W)(\text{E}, \text{GLOBAL}, b, e, b) \quad \triangleq \quad \square\ \forall W' \sqsupseteq^{priv} W, \triangleright\ \mathscr{E}(W')(\text{RX}, \text{GLOBAL}, b, e, b)$$

Since the goal is guarded by an always modality, we must first allocate invariants for all the resources that the closure depends on. In particular, we must allocate an invariant for the local data of the closure. We know that the code will never mutate its content, so we allocate the following invariant:

$$\boxed{l \mapsto 2}^{\mathcal{N}.l} \tag{4.1}$$

Note that the above invariant cannot be maintained, if the capability pointing to l is leaked.

Once all relevant invariants have been allocated, we introduce the always modality. Let W' be a private future worlds of W. Upon introducing the later modality, our goal becomes $\mathscr{E}(W')(\text{RX}, \text{GLOBAL}, b, e, b)$.

Unfolding $\mathscr{E}$, we introduce the following assumptions into our context.

$$\begin{aligned}
&\mathscr{R}(W')(reg) && (4.2)\\
&* \; sharedResources(W') * stsCollection(W') && (4.3)\\
&* \; \mathsf{pc} \mapsto v_1 && (4.4)\\
&\mathop{\huge *}_{(r,w) \in reg / \mathsf{pc}} \quad r \mapsto w && (4.5)
\end{aligned}$$

The proof proceeds by applying weakest precondition rules for stepping through the program, using the instrumented machine state to access shared stack state.

We begin the proof by stepping through `prepstack`, and inferring properties about the stack region in W'. The subroutine dynamically checks that r_{stk} points to an uninitialized write-local capability, with at least one initialized address. Without loss of generality, let's assume that the stack capability equals $(\text{URWLX}, l, b, e, b+1)$.

Given assumption 4.2, we know that the stack capability is valid. We can thus infer that its locality l is DIRECTED. Furthermore, we can infer the following assumptions over W':

$$W'^{std}(b) = \text{Temporary} \tag{4.6}$$

$$W'^{std}(a) = \text{Temporary} \vee \exists w, W'^{std}(a) = \text{Uninitialized}(w) \quad \forall a \in [b+1, e) \tag{4.7}$$

Since we now own the current stack capability, we can safely uninitialize all stack addresses above b. Formally, this is done by updating *all* Temporary states above b in the instrumented machine state to its appropriate Uninitialized state. This holds, since any invariant associated to addresses at and below b will be monotone with regards to $\sqsupseteq^b$, and will thus still hold at the new world. Let W'_{uninit} be the world which results in uninitializing all Temporary states above b. We can prove the following future world relation:

$$W'_{uninit} \sqsupseteq^b W' \tag{4.8}$$

From which we can derive the new instrumented machine state:

$$sharedResources(W'_{uninit}) * stsCollection(W'_{uninit}) \tag{4.9}$$

Next, we load the continuation from the stack. Since we load from the initialized part of the stack, we know that its state is Temporary in W'_{uninit}, and we can infer that the loaded word is valid. Let w_{ret} denote the loaded word.

$$\mathscr{V}(W'_{uninit})(w_{ret}) \tag{4.10}$$

We then reach the interesting part of the program, in which we intentionally try to leak a secret on the stack, namely the capability to the local state of the closure. We are changing the physical state of the stack, which signifies we must update the instrumented machine state accordingly. Since we have uninitialized all addresses above b, we may take transitions that yield a relative-to-b future world.

We currently know that

$$W'_{uninit}(b+1) = \text{Uninitialized}(w)$$

(for some word w). Looking at Figure 4.9, we can observe that updating the state of $b+1$ to Uninitialized$(\text{RWX}, \text{GLOBAL}, l, l+1, l)$ is a temporal transition (since the Uninitialized state changes, the path of the transition goes through Temporary). By definition of $\sqsupseteq^b$, the resulting world is thus a relative-to-b future world, and we now have the following instrumented machine state:

$$\mathit{sharedResources}(W'_{uninit}[b+1 := (\text{RWX}, \text{GLOBAL}, l, l+1, l)]) \tag{4.11}$$

$$* \ \mathit{stsCollection}(W'_{uninit}[b+1 := (\text{RWX}, \text{GLOBAL}, l, l+1, l)]) \tag{4.12}$$

Finally, we load from $(\text{RWX}, \text{GLOBAL}, l, l+1, l)$, and assert that it points to 2. This is easily proven by opening invariant 4.1.

Once registers are cleared, we reach the end of the closure, and jump to the continuation w_{ret}, which we can reason about by applying the fundamental theorem. However, the instrumented machine state has now changed, and we must thus first reestablish the validity of w_{ret}, at world $W'_{uninit}[b+1 := (\text{RWX}, \text{GLOBAL}, l, l+1, l)]$. However, since w_{ret} was loaded from address b, we know its validity is monotone with regards to $\sqsupseteq^b$. We can thus conclude the proof by applying Lemma 20 on 4.10.

□

Next, we apply the adequacy of weakest preconditions to conclude the following.

Theorem 13. *(Correctness of the temporal stack safety example)* *Let $reg \in$ Reg, $m \in$ Mem and*

$$c_{\text{temp}} \triangleq (\text{RX}, \text{GLOBAL}, \ldots) \qquad c_{\text{stk}} \triangleq (\text{URWLX}, \text{DIRECTED}, \ldots)$$

$$c_{\text{adv}} \triangleq (\text{RWX}, \text{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer. Furthermore, assume that:

- *m has been initialized with the code of the program and subroutines (pointed to by c_{temp}), an uninitialized stack (pointed to by c_{stk}), and unknown adversarial code (pointed to by c_{adv});*
- *$reg(\mathsf{pc}) = c_{\text{temp}}$, $reg(r_{\text{stk}}) = c_{\text{stk}}$, $reg(r_0) = c_{\text{adv}}$ and $reg(r) \in \mathbb{Z}$ otherwise;*
- flag *denotes the assertion flag, initialized to* 0*;*

If $(\mathsf{Executable}, (reg, m)) \rightarrow^* (\mu, (reg', m'))$ *then* $m'(\text{flag}) = 0$.

```
(closure creation around f2)
g2: move r1 pc
    lea r1 offset
    restrict r1 encodePerm(E)
    jmp r0
f2: (the linked code is a heap closure)
    reqglob radv
    (calling convention)
    prepstack rstk
    loadU r1 rstk (-1)
    (integrity protection)
    reqRA r1
    checkintregion r1
    (hidden part of stack)
    push “secret”
    (new stack object)
    createstackobj r2 “obj”
    (call linked code)
    scall radv [r0;renv] [r1;r2]
    lea rstk (-6)
    (load hidden part of stack)
    loadU radv rstk (-2)
    (assert its integrity)
    assert radv “secret”
    (return to caller)
    loadU r1 rstk (-4)
    rclear RegName\{pc,r1}
    jmp r1
```

Figure 4.12: Assembly of Listing 4.9

4.4.3.2 Local State Integrity and Stack Objects

We now consider what happens with local state encapsulation in the presence of stack objects. One might expect that this property is rather straightforward: a shared stack object is not encapsulated, but capability bounds ensure that the other parts of the stack are hidden from the context, and that their integrity is guaranteed. However, subtle issues creep up if one is not careful about the parameters exposed to the context, in particular in the cases where a stack object is passed from the caller to a callee. Consider Figure 4.12, where `g2` creates a closure around some code `f2`, which in turn calls some unknown linked code, to which `f2` passes two parameters: a stack object that was passed to `f2` from its caller, and a new stack object created by `f2`. In the callback, the integrity of the unshared parts of `f2`’s stack frame is tested with an assertion. `f2` begins by checking that the linked unknown code is indeed a heap closure (using a macro `reqglob`). `f2` then applies the calling convention from Section 4.3.4 by checking the permission of the stack capability, and loading a parameter from the stack; here the parameter of interest is the older stack object passed to `f2` by the caller.

Since this stack object was passed through the stack, its *read* authority must be smaller (i.e. lower) than `f2`’s stack frame. However, directed capabilities do not enforce any restrictions on the *write* authority of that stack object. In fact, this passed stack object could in principle be an uninitialized capability with a write authority that overlaps with `f2`’s stack frame, thus presenting a threat to the integrity of the stack frame if passed to some unknown code.

To mitigate that threat, `f2` must dynamically check not only that the stack object itself is fully initialized, but also that it transitively does not provide any write access to `f2`’s stack frame. In this particular example, `f2` expects a stack object containing simply integers; `checkintregion` is a macro for checking this. For other examples, other mitigations could be done to inspect the permission of all reachable capabilities within the stack object.

Theorem 14. *(Correctness of the stack object example)*
Let $reg \in \mathrm{Reg}$, $m \in \mathrm{Mem}$ *and*

$$c_{\mathrm{stkobj}} \triangleq (\mathrm{RX}, \mathrm{GLOBAL}, \ldots) \quad c_{\mathrm{stk}} \triangleq (\mathrm{URWLX}, \mathrm{DIRECTED}, \ldots)$$

$$c_{\mathrm{adv}} \triangleq (\mathrm{RWX}, \mathrm{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer. Furthermore, assume that:

- *m has been initialized with the code of the program and subroutines (pointed to by* c_{stkobj}*), an uninitialized stack (pointed to by* c_{stk}*), and unknown adversarial code (pointed to by* c_{adv}*);*
- $reg(\mathsf{pc}) = c_{\mathrm{stkobj}}$, $reg(r_{\mathrm{stk}}) = c_{\mathrm{stk}}$, $reg(r_0) = c_{\mathrm{adv}}$ *and* $reg(r) \in \mathbb{Z}$ *otherwise;*
- flag *denotes the assertion flag, initialized to* 0*;*

If $(\mathsf{Executable}, (reg, m)) \rightarrow^* (\mu, (reg', m'))$ *then* $m'(\mathrm{flag}) = 0$.

Discussion We emphasize that the dynamic checking of the content of a stack object would always have been necessary, including in calling conventions based on LOCAL capabilities [129]. However, in all prior examples, including the awkward example considered by Georges et al. [55], this subtlety never arose, since they did not consider stack objects at all. For instance, the awkward example only allows *global heap closures* as input. The issue is worse for LOCAL or DIRECTED stack closures, for which no dynamic check can be done. In these cases, the only safe option is to very carefully control what other parameters are passed alongside the closure.

4.5 A Binary Model For Confidentiality

So far, we have shown how to use the unary model to reason about examples that depend on integrity properties. To reason about confidentiality properties we must use a binary model. For the temporal aspect of local state confidentiality, we expect that a popped frame should not be able to influence the caller, and thus that two programs whose only difference is to leave different traces on their stack frames ought to be contextually equivalent. We define a binary logical relation and use it to show the contextual equivalence of assembly versions of `f` and `g` from Listing 4.8. We follow well-known techniques for defining binary logical relations in Iris [54, 83, 85], but apply them here for the first time to a low-level capability machine language. The logical relation is parameterised by a binary version of the world. The key technical aspect of the definition is to allow for the uninitialized part of the stack to be uninitialized at different words. We will return to this key aspect later. First, let's examine some high-level aspects of the definition of the binary logical relation.

The logical relation, presented in Figure 4.13, captures program refinement. We depict in blue the parts of the definition that are different from the unary logical

$$
\begin{aligned}
\boxed{\mathcal{E}(W)(v_1, v_2)} \quad \triangleq\ & \forall \mathit{reg}_1, \mathit{reg}_2, \mathcal{R}(W)(\mathit{reg}_1, \mathit{reg}_2) \\
& * \mathit{sharedResources}(W) * \mathit{stsCollection}(W) \\
& * \Mapsto \mathsf{Executable} \\
& * \mathsf{pc} \mapsto v_1 * \mathsf{pc} \rightarrowtail v_2 \\
& \mathop{\huge *}_{(r,w_1) \in \mathit{reg}_1/\mathsf{pc}\ \wedge\ (r,w_2) \in \mathit{reg}_2/\mathsf{pc}} r \mapsto w_1 * r \rightarrowtail w_2 \mathrel{-\!\!*} \\
& \mathsf{wp}\ \mathsf{Executable} \left\{ \begin{array}{l}
v, v = \mathsf{Halted} \rightarrow \\
\exists W'\ \mathit{reg}'_1\ \mathit{reg}'_2,\ W' \sqsupseteq^{priv} W \\
\ * \Mapsto \mathsf{Halted} \\
\ * \mathit{sharedResources}(W') \\
\ * \mathit{stsCollection}(W') \\
\ \mathop{\huge *}_{(r,w_1) \in \mathit{reg}'_1\ \wedge\ (r,w_2) \in \mathit{reg}'_2} r \mapsto w_1 * r \rightarrowtail w_2
\end{array} \right\}
\end{aligned}
$$

$$
\boxed{\mathcal{R}(W)(\mathit{reg}_1, \mathit{reg}_2)} \quad \triangleq \ \mathop{\huge *}_{(r,w_1) \in \mathit{reg}_1/\mathsf{pc}\ \wedge\ (r,w_2) \in \mathit{reg}_2/\mathsf{pc}} \mathcal{V}(W)(w_1, w_2)
$$

$$
\boxed{\mathcal{V}(W)(w_1, w_2)} \quad
\begin{cases}
\mathcal{V}(W)(z_1, z_2) \triangleq z_1 = z_2 \\
\mathcal{V}(W)((\mathrm{O}, g, b, e, a), w_2) \triangleq (\mathrm{O}, g, b, e, a) = w_2 \\
\mathcal{V}(W)((\mathrm{E}, g, b, e, a), w_2) \triangleq (\mathrm{E}, g, b, e, a) = w_2 \\
\qquad * \square\ \forall W' \sqsupseteq^{g} W, \\
\qquad\qquad \triangleright\ \mathcal{E}(W')\left(\begin{smallmatrix}(\mathrm{RX}, g, b, e, a), \\ (\mathrm{RX}, g, b, e, a)\end{smallmatrix}\right) \\
\mathcal{V}(W)((p, g, b, e, a), w_2) \triangleq (p, g, b, e, a) = w_2 \\
\qquad \mathop{\huge *}_{a' \in [b,e)} \begin{cases} \mathcal{S}^u(W)(a', g, p, a) & \text{if } p = \mathrm{U}\text{-} \\ \mathcal{S}(W)(a', g, p) & \text{otherwise} \end{cases} \\
\qquad\qquad \wedge \begin{cases} \exists P, \mathit{rel}(a', P) * \mathrm{rcond}(P) & \mathit{if}\ p = \mathrm{RO}|\mathrm{RX} \\ \mathit{rel}(a', \mathcal{V}) & \text{otherwise} \end{cases}
\end{cases}
$$

$$
\begin{aligned}
\boxed{\mathrm{rcond}(P)} \quad \triangleq\ & \triangleright \square \forall W, w_1, w_2, P(W)(w_1, w_2) \mathrel{-\!\!*} \mathcal{V}(W)(w_1, w_2) \\
& * \triangleright \square \forall W_1, W_2, z_1, z_2, P(W_1)(z_1, z_2) \mathrel{-\!\!*} P(W_2)(z_1, z_2)
\end{aligned}
$$

$$\boxed{\text{State relation}}$$

$$
\mathcal{S}(W)(a, g, p) \quad \triangleq \begin{cases}
W^{std}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \\
& \wedge\ g = \mathrm{DIR} \\
W^{std}(a) = \text{Temporary} & \text{if write-local}(p) \\
& \wedge\ g = \mathrm{DIR} \\
W^{std}(a) = \text{Permanent} & \text{if } g \neq \mathrm{DIR}
\end{cases}
$$

$$
\mathcal{S}^u(W)(a, g, p, \mathit{mid}) \triangleq \begin{cases}
\mathcal{S}(W)(a, g, p) & \\
\vee\ \exists w_1, w_2, W^{std}(a) = \text{Uninitialized}(w_1, w_2) & \text{if } a \geq \mathit{mid} \\
& \wedge\ g = \mathrm{DIR} \\
\mathcal{S}(W)(a, g, p) & \text{otherwise}
\end{cases}
$$

Figure 4.13: A Binary Logical Relation with Support for Temporal Stack Safety

relation. The expression relation $\mathcal{E}(v_1, v_2)$ describes that the program pointed to by capability v_1, thought of as the *implementation*, refines the program pointed to by capability v_2, thought of as the *specification*. The trick in defining logical refinements in Iris is to use ghost state (separate from the state interpretation) to track the current state and expression of the *specification*.

In our low level capability machine, this means we use ghost state to track the state of specification registers, denoted $r \rightarrowtail w$, the state of specification memory, denoted $a \rightarrowtail w$, and the current execution mode of the specification program, denoted $\Mapsto \mu$. As usual, these ghost state assertions depict fragmental views of the ghost state. We store the full authoritative view in an Iris invariant, henceforth denoted *specCtx*.

The expression relation can roughly be interpreted as follows: given an implementation register state that refines a specification register state, where both the implementation and specification are in Executable mode, if the implementation halts, then the specification must also halt, and all established invariants of the (binary) instrumented machine state hold at some private future world.

Capability machine programs are able to observe and compare words. As a first step towards a value relation, we thus observe that a word w_1 can only refine w_2 if they are equal. However, syntactic equivalence is not enough. Capabilities that grant read authority must themselves point to refined memory fragments. Just as in the unary case, we use the (binary) instrumented machine state to relate the memory fragments within the authority of a valid capability. The binary state relation enforces appropriate standard states on the world. These standard states are as in Figure 4.9, except the Uninitialized state now records two words, one for the specification and one for the implementation. Crucially, since a directed capability only grants write authority to its uninitialized part, the content of the latter cannot affect program execution. The state interpretation thus allows the content of implementation side uninitialized memory to differ from its specification counterpart, as reflected by the standard state Uninitialized(w_1, w_2). This is the key point that allows us to verify the contextual equivalence of programs that depend on the temporal aspect of confidentiality.

With the binary logical relation in place, we prove a similar fundamental theorem of logical relations as Theorem 12.

Theorem 15 (Binary FTLR). *Assume that* $p = \text{RX}$*,* $p = \text{RWX}$ *or* $(p = \text{RWLX} \wedge g = \text{DIRECTED})$*. Assume also that* $\mathcal{V}(W)((p,g,b,e,a),(p,g,b,e,a))$ *and the invariant specCtx. Then we have that* $\mathcal{E}(W)((p,g,b,e,a),(p,g,b,e,a))$.

We use the logical relation to show contextual equivalence of components. Informally, two components are contextually equivalent if no context can distinguish them through termination. A component is either a library or a main component. A component C can be considered a context for component *comp* when their linking $C[comp]$ generates a closed program, that is, there are no imports left to satisfy. We only consider components that are well-formed. A component is well-formed if its memory segment does not overlap with the stack memory, and all capabilities that it contains only address its own memory segment, and are not permit-write-local.

```
f3: prepstack r_stk                    h3: prepstack r_stk
    loadU r_0 r_stk (-1)                   loadU r_0 r_stk (-1)
    push 2                                 push 3
    rclear RegName\{pc,r_0}                rclear RegName\{pc,r_0}
    jmp r_0                                jmp r_0
```

Figure 4.14: Assembly of Listing 4.8

The initial state of a closed program is a state where the register file contains only 0, except for the program counter, which should be initialized to the entry point of the program, and r_{stk}, which should be initialized with the stack capability. The memory is empty except for the part specified by the memory segment of the program, and the stack, which is initialized with arbitrary words. These definitions are formally stated in Section 4.6.2. We use them to formally define contextual equivalence.

$$comp_1 \approx_{ctx} comp_2 \quad \triangleq \quad \forall C, C[comp_1] \Downarrow \iff C[comp_2] \Downarrow$$

We use $c \Downarrow$ to denote that the configuration c terminates in a Halted state.

Let's use the binary model to prove the equivalence of the two programs from Listing 4.8. Figure 4.14 depicts their low level implementation. `f3` pushes 2 onto the stack, clears its registers and jumps to a return capability loaded from the stack. `h3` behaves similarly, except it pushes 3 onto the stack. The two programs leave different traces on their respective stack frames, but if temporal confidentiality is enforced, no caller can distinguish them. We use the binary model to show that the two programs refine each other according to our logical refinement definition.

Lemma 24. *For any world W, assuming that the memory has been properly initialized in the implementation and in the specification each in region $[b_{\text{ni}}, e_{\text{ni}})$ with the code of the respective programs, and given the invariant specCtx :*

$$\mathcal{E}(W)((\text{RX}, \text{GLOBAL}, b_{\text{ni}}, e_{\text{ni}}, \mathit{f3}), (\text{E}, \text{GLOBAL}, b_{\text{ni}}, e_{\text{ni}}, \mathit{h3}))$$

Proof. Unfolding $\mathcal{E}$, we introduce the following assumptions into our context.

$$\mathcal{R}(W)(reg_1, reg_2) \tag{4.13}$$

$$* \; sharedResources(W) \; * \; stsCollection(W) \tag{4.14}$$

$$* \; \Rrightarrow \mathsf{Executable} \tag{4.15}$$

$$* \; \mathsf{pc} \mapsto v_1 \; * \; \mathsf{pc} \rightarrowtail v_2 \tag{4.16}$$

$$\mathop{\Huge *}_{(r,w_1) \in reg_1/\mathsf{pc} \,\wedge\, (r,w_2) \in reg_2/\mathsf{pc}} r \mapsto w_1 \; * \; r \rightarrowtail w_2 \tag{4.17}$$

Where the goal is a weakest precondition, in which the post condition matches that given by the expression relation definition (Figure 4.13). The proof proceeds by applying weakest precondition rules for stepping through the implementation program, while applying respective rules that manipulate $\Rrightarrow \mathsf{Executable}$, $a \rightarrowtail -$ and $r \rightarrowtail -$ for stepping through the specification program.

We begin the proof by stepping through `prepstack`, and inferring properties about the stack region in W. The subroutine dynamically checks that r_{stk} points to an uninitialized write-local capability, with at least one initialized address. Without loss of generality, let's assume that the stack capability in the implementation equals $(\textsc{urwlx}, l, b, e, b+1)$.

We know from assumption 4.13, that the stack capabilities are in the binary value relation. We can thus infer that the specification stack capability also equals $(\textsc{urwlx}, l, b, e, b+1)$, and that its locality l is $\textsc{directed}$. Furthermore, we can infer the following assumptions over W:

$$W^{std}(b) = \text{Temporary} \tag{4.18}$$

$$W^{std}(a) = \text{Temporary} \vee \exists w_1, w_2, W^{std}(a) = \text{Uninitialized}(w_1, w_2) \quad \forall a \in [b+1, e) \tag{4.19}$$

As previous, since we currently have ownership of the stack, we can safely uninitialize all stack addresses above b. Let W_{uninit} be the world which results in uninitializing all Temporary states above b. We can prove the following future world relation:

$$W_{uninit} \sqsupseteq^b W \tag{4.20}$$

From which we can derive the new instrumented machine state:

$$\mathit{sharedResources}(W_{uninit}) \ * \ \mathit{stsCollection}(W_{uninit}) \tag{4.21}$$

Next, we step through the `loadU` instruction, followed by the `push` subroutine. Both use the shared stack, and must be reasoned about in lock-step. Since we are loading from the initialized part of the stack, we know that its state is Temporary (assumption 4.18), and thus that the loaded word is also in the value relation (Note that the value relation is instantiated to the new world W_{uninit}). Let w_{ret} and w'_{ret} be the loaded words.

$$\mathcal{V}(W_{uninit})(w_{ret}, w'_{ret}) \tag{4.22}$$

The next part of the proof is particularly interesting, as the two programs push different values onto the stack. Since we have uninitialized the instrumented machine state for all addresses above b, we know that $W_{uninit}(b+1) = \text{Uninitialized}(w_1, w_2)$, for some w_1 and w_2. However, since `push` updates the state of $b+1$, the instrumented machine state must likewise change. After applying relevant rules to step through the subroutine, we update the instrumented machine state by setting $b+1$ to $\text{Uninitialized}(2,3)$, denoted $W_{uninit}[b+1 := (2,3)]$. We prove the following world relation:

$$W_{uninit}[b+1 := (2,3)] \sqsupseteq^b W_{uninit} \tag{4.23}$$

The update is allowed, since there are no Temporary invariants in W_{uninit} above b, and all remaining Temporary invariants are monotone with regards to $\sqsupseteq^b$. Likewise, we can derive the following from applying Lemma 20 to assumption 4.22.

$$\mathcal{V}(W_{uninit}[b+1 := (2,3)])(w_{ret}, w'_{ret}) \tag{4.24}$$

Finally, we step through the remaining instructions, and reason about the continuations w_{ret} and w'_{ret} by applying the binary fundamental theorem on assumption 4.24.

□

We prove a similar lemma to show the refinement in the other direction.

We then apply the adequacy of weakest preconditions to prove the following contextual equivalence:

Theorem 16. *(Correctness of the temporal confidentiality example) Let $comp_{f3}$ and $comp_{h3}$ be two components containing the programs* `f3` *and* `h3` *respectively, where*

$$comp_{f3}.exports \triangleq \{\mathtt{0} : (\text{E}, \text{GLOBAL}, ..., \mathtt{f3})\}$$

$$comp_{h3}.exports \triangleq \{\mathtt{0} : (\text{E}, \text{GLOBAL}, ..., \mathtt{h3})\}$$

in which the respective exported entry points have an appropriate range of authority and pointer.

Furthermore, assume that contexts are defined as well-formed components with no exports, a single import `0`*, and a memory segment with instructions (integers) only. Then*

$$comp_{f3} \approx_{ctx} comp_{h3}$$

Proof. By applying Iris adequacy on the conclusion from Lemma 24. □

4.6 Characterizing security using a fully abstract overlay semantics

The unary and binary model can be used to prove the integrity and confidentiality of example programs that may depend on any of the five properties presented in Section 4.2.1. While the unary and binary model capture integrity and confidentiality properties, and proving examples increase our confidence in the calling convention, they do not *define* any notion of stack safety. Rather than detailing examples that vaguely cover all properties, we wish to truly capture this notion of stack safety, and prove that our new calling convention enforces it.

To that end, we follow the same approach as Skorstengaard et al. [132] for proving that our calling convention does enforce the security properties we claim. We first start by describing an overlay semantics (Section 4.6.1) whose aim is to clearly capture the properties included in our notion of stack safety. Next, we provide some base definitions for components, linking and contexts (Section 4.6.2). Then, we prove a full abstraction theorem between the overlay semantics and the original base capability machine semantics (Section 4.6.3).

4.6.1 Overlay Semantics

The overlay semantics augments the base semantics of the capability machine with additional structure to model the properties we wish to enforce, these components are indicated in blue.

$$
\begin{aligned}
c \in \mathsf{Cap} &\triangleq \{(p,\ell,b,e,a) \mid b,e,a \in \mathsf{Addr}\} \\
&\quad \cup \{\mathsf{Stk}(d,p,b,e,a) \mid d \in \mathbb{N}, p \in \mathsf{Perm}, b,e,a \in \mathsf{Addr}\} \\
&\quad \cup \{\mathsf{Ret}(b,e,a) \mid b,e,a \in \mathsf{Addr}\} \\
\mathit{instr} &::= \ \dots \mid \mathsf{call}\ r\ \vec{r} \\
\mathit{sf} \in \mathsf{Stackframe} &\triangleq (\mathsf{Reg} \times \mathsf{Mem}) \\
\varphi \in \mathsf{ExecConf} &\triangleq \mathsf{Reg} \times \mathsf{Mem} \times \mathsf{Mem} \times \mathsf{list}\ \mathsf{Stackframe}
\end{aligned}
$$

Syntax Configurations in the overlay semantics now track a list of overlay stack frames, and natively separate the heap and stack memory. Configurations are now quadruples (reg,h,stk,cs) where reg and stk are the current register state and current stack frame. h is the state of the heap, while cs corresponds to the call stack, a list of saved register states and stack frames.

The overlay semantics has two additional kinds of capabilities; stack derived capabilities $\mathsf{Stk}(d,p,b,e,a)$, and return capabilities $\mathsf{Ret}(b,e,a)$. A capability $\mathsf{Stk}(d,p,b,e,a)$ provides access to the d^{th} stack frame with permission p over range $[b,e)$, and currently points to address a. That is, if the current state is (reg,h,stk,cs), then $d=0$ provides access to the *oldest* stack frame (i.e., at the tail of cs), while $d=|cs|$ provides access to the current stack frame stk. For instance, if $|cs|=1$, then it means that the current executing function is at depth 1, its caller is necessarily the main entrypoint function. Return capabilities $\mathsf{Ret}(b,e,a)$ make the overlay semantics return from a call by deallocating the topmost frame, the addresses b,e,a do not matter except for the full abstraction proof which will be explained Section 4.6.3. The regular capabilities (p,ℓ,b,e,a) are now specifically for accessing only the heap in the overlay semantics.

Call The overlay semantics provide a new instruction $\texttt{call}\ r\ \vec{r}_{args}$ which calls the function given in the register r, and passing the arguments in $\vec{r}_{args}$. The operational semantics is given an additional rule for executing calls as follows.

$$\textsc{ExecCall}\quad \frac{\begin{array}{c} {}^{\dagger}\varphi.reg(\mathsf{pc}) = (p,\ell,b,e,a) \qquad {}^{\dagger}p \in \{\text{RX},\text{RWX},\text{RWLX}\} \\ {}^{\dagger}[a, a + |\mathsf{call}\ r\ \vec{r}_{args}|) \subseteq [b,e) \\ {}^{\dagger}\varphi.h([a, a+|\mathsf{call}\ r\ \vec{r}_{args}|)) = [\mathsf{call}_0\ r\ \vec{r}_{args}; \mathsf{call}_1\ r\ \vec{r}_{args}; \ldots] \\ {}^{\P}\forall r, r \in \vec{r}_{args}, \mathsf{safe}(\varphi.reg(r)) \\ {}^{\S}\varphi.reg(r) = (\text{E}, \ell', b', e', a') \qquad {}^{\bigstar}\varphi.reg(r_{\text{stk}}) = \text{Stk}(d, \text{URWLX}, b_{stk}, e_{stk}, a_{stk}) \\ {}^{\bigstar}d = |\varphi.cs| \qquad {}^{\bigstar}[a, a_{stk} + |act_{code}| + 1 + |\vec{r}_{args}|) \subseteq [b_{stk}, e_{stk}) \\ {}^{\bullet}\forall i, \text{canBeStored}(\varphi.reg(r_i), a_{stk}+i) \\ \varphi' = (reg', \varphi.h, stk', (reg^*, stk^*) :: \varphi.cs) \end{array}}{(\mathsf{Executable},\ \varphi) \to (\mathsf{Executable},\ \varphi')}$$

where

- reg' is defined such that $reg'(\mathsf{pc}) = (\text{RX}, \ell', b', e', a')$,
  $reg'(r_{\text{stk}}) = \text{Stk}(d+1, \text{URWLX}, a_{stk} + |act_{code}|, e_{stk}, a_{stk} + |act_{code}| + 1 + |\vec{r}_{args}|)$
  and
  $reg'(r') = \text{if } r' = r \text{ then } reg(r) \text{ else } 0$;
- $stk' = \emptyset[a_{stk} + |act_{code}| \mapsto \text{Ret}(\ldots), \ldots]$;
- $reg^* = reg[\mathsf{pc} \mapsto (p, \ell, b, e, a + |\mathsf{call}\ r\ \vec{r}_{args}|)]$;
- $stk^* = stk[a_{stk} \mapsto \ldots, \ldots, a_{stk} + |act_{code}| - 1 \mapsto \ldots]$.

First, the overlay machine dynamically checks that the current pc contains a valid program counter, pointing to the code pattern implementing call $r\ \vec{r}_{args}$ (indicated with a † in ExecCall). It then checks (¶) the parameters of the call are *safe* and (§) the call target is an enter capability.

Definition 2 (Safe word)**.** *Given an overlay configuration $\varphi = (reg, h, stk, cs)$, a word w is* safe *to pass as argument in that configuration, written safe(φ, w) if one of the following holds.*

- *w is an integer ($w \in \mathbb{Z}$);*
- *w is a heap capability (p, ℓ, b, e, a);*
- *w is a stack derived capability* $\text{Stk}(d, p, b, e, a)$ *such that*
  - *$d < |cs|$ and cs is of the form $(reg_n, stk_n) :: \cdots :: (reg_0, stk_0)$ and $reg_d(r_{\text{stk}}) = \text{Stk}(d, p', b', e', a')$ and $e < a'$ and for all $a'' \in [b, \text{canReadUpTo}(w))$, safe$(\varphi, stk_d(a''))$;*
  - *or $d = |cs|$ and the above conditions hold using $reg_d = reg$ and $stk_d = stk$ instead.*

Intuitively, a word is safe to pass if it cannot tamper the activation code of active return capabilities. As such, integers and heap capabilities are obviously safe. Return capabilities are unsafe to pass as this would break well-bracketed control flow (though allowing to pass them would actually implement a form of `longjmp`). The activation code of return capabilities is stored on the stack, just above the stackframe used by a function, hence the $e < a'$ condition above. We must also ensure that the capability can only give access to safe words itself.

We point out, that while the definition may not seem well-founded, a stack capability can only be safe if its contents are also safe (which may itself contain the original capability), this is actually not the case, as stack capabilities are *directed capabilities*, making this definition provide a terminating algorithm. Indeed, in the recursive check $\mathit{safe}(\varphi, \mathit{stk}_d(a''))$, it is either a word that is immediately safe (i.e., an integer or a heap capability), or immediately unsafe (a return capability), or a stack capability w' such that $\text{canReadUpTo}(w') \leq a''$ (by property of directed capabilities) and $a'' < \text{canReadUpTo}(w)$ (by definition of a''). There is therefore a decreasing measure ensuring termination of the check.

In the Coq formalization, we use a more *restrictive* notion of safe words that instead only allow passing integers and heap capabilities.

(★) It then checks that the provided capability in r_{stk} is actually a capability for the current stack, with enough range to store the activation code and parameters. The local state is stored on the stack, followed by the activation record.

(•) The overlay semantics requires that all of these can be legally stored on the stack. The callee is given a fresh register state (all cleared except pc, r_{stk} and r), and receives a fresh stackframe with the return capability at the bottom, and all parameters above it. Finally, the local state $(\mathit{reg}^*, \mathit{stk}^*)$ of the caller is pushed on the call stack. If a condition is not satisfied, the overlay semantics falls back on the rule EXECSTEP, and simply executes one instruction at a time. The formal rule for call, denoted EXECCALL has been formalized in Coq.

Return A jump using a return capability is interpreted as a return.

$$[\![\mathsf{jmp}\ r]\!](\mathit{reg}, h, \mathit{stk}, \mathit{cs}) = (\mathit{reg}', h, \mathit{stk}', \mathit{cs}') \text{ when } \begin{cases} \mathit{reg}(r) = \mathsf{Ret}(b, e, a) \\ \mathit{cs} = (\mathit{reg}', \mathit{stk}') :: \mathit{cs}' \end{cases}$$

The semantics of return capture temporal stack safety. The topmost stack frame stk is deallocated and the local environment reg' and stk' is restored. By deallocating the topmost stack frame, we capture that the caller loses any read access to its old content. This is in contrast with [132], where the stack frames stk and stk' are instead merged together, giving the caller access to whatever was left on the stack.

Properties of the overlay semantics EXECCALL and jumping using a return capability are the only way in the semantics to push or pop the call stack, it is thus obvious that WBCF is enforced by the overlay semantics. As the topmost stack frame is entirely removed when returning, temporal stack safety is also natively enforced by the semantics. Finally, a stack frame can only be accessed using a corresponding

$$
\begin{aligned}
ms \in \mathsf{MemFrag} &::= \mathsf{Addr} \rightharpoonup \mathsf{Word} \\
\overline{a \mapsfrom s} \in \mathsf{Imports} &::= \overline{\mathsf{Symbols} \times \mathsf{Addr}} \\
\overline{s \mapsto w} \in \mathsf{Exports} &::= \mathsf{Symbols} \rightharpoonup \mathsf{Word} \\
basecomp \in \mathsf{BaseComp} &::= \mathsf{Memfrag} \times \mathsf{Imports} \times \mathsf{Exports} \\
\mathsf{Comp} &::= basecomp \mid (basecomp, c_{main})
\end{aligned}
$$

(a) Components.

$$
\frac{\begin{array}{c}
comp_1 = (ms_1, \overline{a \mapsfrom s}^1, \overline{s \mapsto w}^1) \qquad comp_2 = (ms_2, \overline{a \mapsfrom s}^2, \overline{s \mapsto w}^2) \\
comp = (ms, \overline{a \mapsfrom s}, \overline{s \mapsto w}) \qquad \overline{s \mapsto w} = \overline{s \mapsto w}^1 \cup \overline{s \mapsto w}^2 \\
ms = (ms_1 \uplus ms_2)[a \mapsto w \mid a \mapsfrom s \in (\overline{a \mapsfrom s}^1 \cup \overline{a \mapsfrom s}^2), s \mapsto w \in \overline{s \mapsto w}] \\
\overline{a \mapsfrom s} = (\overline{a \mapsfrom s}^1 \cup \overline{a \mapsfrom s}^2) \setminus \{\_ \mapsfrom s \mid s \mapsto w \in \overline{s \mapsto w}\}
\end{array}}{comp = comp_1 \bowtie comp_2}
$$

(b) Linking.

Figure 4.15: Components and linking.

Stk capability with the right depth. ExecCall is the only rule that creates a Stk capability with an increasing depth and only provides it to the callee, a caller cannot thus access to a callee's stack frame. Conversely, a callee is only given access to its Stk capability and parameters, it thus cannot access its caller's local state, local state is also natively enforced by the overlay semantics.

4.6.2 Components, Linking and Contexts

In Section 4.5, we defined contextual equivalence on top of definitions for components, linking and context. Before we present the full abstraction theorem, which also depends on these, let's first establish them formally.

A closed program is created by linking together multiple components, until all imports are satisfied. A component (defined in Figure 4.15a) is either a base component, representing a library waiting to be linked with other components, or a main component, a base component with a main entrypoint. A base component is defined by a memory fragment representing the part of the memory owned by the component, its content, a list of imports indicating where an import will be stored by the linker, and a list of exports associating words with symbols.

Linking components $comp_1$ and $comp_2$, denoted $comp_1 \bowtie comp_2$, is defined in Figure 4.15b. For linking to succeed, components must have disjoint memories, with at most one main component amongst them. The new component $comp_1 \bowtie comp_2$ is obtained by taking the union of both components' memory fragments, and satisfy all matching imports and exports by storing the export word into the import memory. The satisfied imports are removed from the resulting program, and the components' exports are combined.

We can now define notions of contexts and closed programs.

Definition 3 (Closed programs and contexts)**.** *A main component is a closed program when its imports list is empty.*

A component $\mathscr{C}$ is a context for component comp, written $\mathscr{C}[comp]$ when $comp \bowtie \mathscr{C}$ is a closed program.

A closed program can run once we define its initial state. Specifically, we are interested in running well-formed programs.

Definition 4 (Well-formed components)**.** *A base component $(ms, \overline{a \mapsfrom s}, \overline{s \mapsto w})$ is well-formed when:*

- *import and export symbols are disjoint;*
- *all exported words can only address memory owned by the component, i.e., ms;*
- *all import addresses are part of the memory owned by the component;*
- *ms only contains integers or capabilities that can address memory within ms, are not permit-write-local (i.e., have permission* URWL*,* URWLX*,* RWL *or* RWLX*) and are global;*
- *ms is disjoint from the memory reserved for the call stack.*

A main component is well-formed when its base component is well-formed and its main entrypoint can only address memory it owns.

These properties are necessary in order to satisfy the initial assumptions of a secure capability machine program. For instance, stack safety cannot be enforced if any of the components have direct access to the call stack, or if they can keep a copy of stack derived capabilities in their own memory by owning separate permit-write-local capabilities.

Assuming that $[b_{stk}, e_{stk})$ is the range reserved for the stack, the initial configuration of a closed program $((ms, [], \overline{s \mapsto w}), c_{main})$ is a configuration (reg, m) in the base semantics such that:

- $reg(\mathsf{pc}) = c_{main}$;
- $reg(r_{stk}) = (\text{URWLX}, \mathsf{Directed}, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq \mathsf{pc}$ or $r \neq r_{stk}$;
- $mem(a) = ms(a)$ if $a \in \mathsf{dom}(ms)$, and $mem(a) = 0$ otherwise.

In the overlay semantics, the initial configuration of a closed program $((ms, [], \overline{s \mapsto w}), c_{main})$ is the configuration $(reg, ms, \emptyset, [])$ such that:

- $reg(\mathsf{pc}) = c_{main}$;
- $reg(r_{stk}) = \mathsf{Stk}(0, \text{URWLX}, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq \mathsf{pc}$ or $r \neq r_{stk}$.

With these definitions in place, we can now formally state and prove the full abstraction theorem.

4.6.3 A Full Abstraction Theorem

In order to show that our new calling convention does enforce stack safety, we prove a full abstraction theorem between the overlay semantics and the base capability machine. Full abstraction states that two components are indistinguishable by other components in the overlay semantics if and only if they are indistinguishable by other components in the base semantics. Informally, this shows that adversarial contexts in the base capability machine are not stronger that those in the overlay semantics. The theorem is stated as follows. We use *blue* for the overlay machine, and **red** for the regular capability machine.

Theorem 17. *For well-formed components $comp_1$ and $comp_2$, we have*

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow \mathbf{comp_1} \approx_{ctx} \mathbf{comp_2}$$

The theorem states that contextual equivalences are preserved and reflected, it is proved using a simple simulation argument. A forward simulation is defined as follows.

Definition 5 (Forward simulation)**.** *We say that $\sim$ is a forward simulation between programs p_1 and p_2 when the following holds.*

1. *Let ϕ and ψ be the initial states of p_1 and p_2, then $\phi \sim \psi$.*
2. *Let ϕ and ψ such that $\phi \sim \psi$, then if $\phi \to \phi'$ then there exists ψ' such that $\psi \to^* \psi'$ and $\phi' \sim \psi'$.*
3. *Let ϕ and ψ such that $\phi \sim \psi$, then if ϕ is a final state of p_1, then ψ is a final state of p_2.*

A forward simulation implies preservation of termination [89]:

Lemma 25. *If there exists a forward simulation between programs p_1 and p_2, then $p_1 \Downarrow \Rightarrow p_2 \Downarrow$.*

Before proving Theorem 17, we first show the following lemma.

Lemma 26. *For all well-formed closed programs $\mathscr{P}$ in the overlay semantics, there exists a forward simulation between $\mathscr{P}$ and its counterpart* **P** *in the base semantics.*

Proof. The detailed proof can be found in the accompanying Coq development, we only provide a proof sketch here. The crux of the proof is to build a relation $\sim$ and show it is a forward simulation. We say that $(reg, h, stk, cs) \sim (\mathbf{reg}, \mathbf{mem})$ when reg and $\mathbf{reg}$ contain related words in each register. An integer is only related to itself, while a stack capability $\mathsf{Stk}(d, p, b, e, a)$ is related to a directed capability $(p, \text{DIRECTED}, b, e, a)$. Similarly, return capabilities $\mathsf{Ret}(b, e, a)$ are related to enter directed capabilities $(\text{E}, \text{DIRECTED}, b, e, a)$ and regular capabilities (p, ℓ, b, e, a) are related to themselves. Furthermore, the heap h, the current stack stk and saved stack frames in the call stack cs must have disjoint domains, as well as have related words at corresponding addresses with **mem**.

When this relation is defined, it is relatively straightforward, though tedious, to show that it is indeed a forward simulation. The simulation operates in a “lockstep” fashion, except for the EXECCALL and EXECRETURN steps where one step in the overlay semantics is simulated by multiple ones in the base semantics. □

We can now prove our full-abstraction theorem (Theorem 17).

Proof. By unfolding the definitions, we need to prove the following:

$$(\forall \mathscr{C}, \mathscr{C}[\mathit{comp}_1]\Downarrow \Leftrightarrow \mathscr{C}[\mathit{comp}_2]\Downarrow) \Leftrightarrow (\forall \mathbf{C}, \mathbf{C}[\mathbf{comp_1}]\Downarrow \Leftrightarrow \mathbf{C}[\mathbf{comp_2}]\Downarrow)$$

By combining Lemma 25 and Lemma 26, we know that for all closed programs p, $p\Downarrow \implies \mathbf{p}\Downarrow$. Furthermore, as the base capability machine semantics is deterministic, we can actually build a backward simulation from the forward simulation [89]. We thus actually have that for all closed programs p, $p\Downarrow \Leftrightarrow \mathbf{p}\Downarrow$.

Without loss of generality, we can thus consider the $\Rightarrow$ implication (the other direction is similar). By symmetry it suffices once again to consider only the $\Rightarrow$ direction of $(\forall \mathbf{C}, \mathbf{C}[\mathbf{comp_1}]\Downarrow \Leftrightarrow \mathbf{C}[\mathbf{comp_2}]\Downarrow)$.

We prove this by following the proof structure shown on the left. Let $\mathbf{C}$ be a context such that $\mathbf{C}[\mathbf{comp_1}]\Downarrow$, we need to prove that $\mathbf{C}[\mathbf{comp_2}]\Downarrow$, knowing that $\mathscr{C}[\mathit{comp}_1]\Downarrow \Leftrightarrow \mathscr{C}[\mathit{comp}_2]\Downarrow$.

$$\begin{tikzcd}
\mathscr{C}[\mathit{comp}_1]\Downarrow \arrow[r, Rightarrow, "(2)"'] & \mathscr{C}[\mathit{comp}_2]\Downarrow \arrow[d, Rightarrow, "(3)"] \\
\mathbf{C}[\mathbf{comp_1}]\Downarrow \arrow[u, Rightarrow, "(1)"] \arrow[r, Rightarrow, "?"'] & \mathbf{C}[\mathbf{comp_2}]\Downarrow
\end{tikzcd}$$

The steps described in the figure can then be proved as follows.

(1) We use the backward simulation to prove that $\mathscr{C}[\mathit{comp}_1]\Downarrow$.

(2) By assumption, we have that $\mathscr{C}[\mathit{comp}_1]\Downarrow \Leftrightarrow \mathscr{C}[\mathit{comp}_2]\Downarrow$, and therefore $\mathscr{C}[\mathit{comp}_2]\Downarrow$.

(3) We use the forward simulation to conclude that $\mathbf{C}[\mathbf{comp_2}]\Downarrow$.

□

4.7 Related Work

We now discuss some related work that has not already been discussed in the paper.

In this paper we have emphasized temporal stack safety, but capabilities have also recently been proposed in the CHERI project as a mechanism for ensuring temporal memory safety for the heap. In particular, in CHERIvoke [163] and Cornucopia [52] the authors suggest to use capabilities to efficiently and securely reclaim memory managed by a dedicated memory allocator using a garbage-collector-like approach. In contrast to our work, they do not formally state nor prove the guarantees provided by their mechanism, and it would be interesting to do that in future work.

We have already discussed the most closely related work on formalising capability safety. Other related approaches include the work of Nienhuis et al. [109] and Bauereiss et al. [18], who define a syntactic notion of capability safety as a monotonicity guarantee of reachable objects (the machine does not create new capabilities out of thin air); in contrast to our approach, they do not consider safety across calls to possibly adversarial code, so they only show that security properties hold within a single component. On the other hand, they consider a capability machine model with all of the instructions found on a real machine (Morello in the case of [18]) whereas we consider a core capability machine model. Devriese et al. [37] propose a semantic approach to capturing capability safety for a high-level language with object capabilities using logical relations; this approach was further expanded upon by Swasey et al. [138], who showed the robustness of several object capability patterns. Recently, El-Korashy et al. [44] have studied the formal security guarantees of PAC (pointers-as-capabilities) compilers for partial programs and characterized them via a full abstraction result.

Full abstraction [1] is a well-known property in the field of secure compilation [112] and has been used in recent works to characterize the security properties provided by different capability machines. Our approach follows the one proposed by Skorstengaard et al. [132] who use a fully abstract overlay semantics to define and show that their protection mechanisms enforce WBCF and LSE. Their proof uses a complex cross-language logical relation and is not mechanized, whereas we use a simpler simulation argument. Van Strydonck et al. [148] also use a simulation-based argument for a fully abstract compiler from a statically verified language to an unverified language with support for linear capabilities. Likewise, Tsampas et al. [144] also use a simulation proof for proving full abstraction between an “ideal” semantics with native temporal safety and an imperative language equipped with capabilities. It is interesting to note that their higher-level semantics already assumes well-bracketed control-flow with automatic deallocation of stackframes on return; we provably enforce that using directed capabilities.

While we use an overlay semantics to characterize the notion of stack safety our calling convention guarantees, Anderson et al. [10] define stack safety as a trace property, expressed as the conjunction of LSE and WBCF. Anderson et al. distinguish between the integrity of local state, and the confidentiality of local state. Likewise, we develop a unary model to reason about the integrity of specific examples, and a binary model to reason about confidentiality. Unlike Anderson et al., we consider a machine with both a stack and a heap. Anderson et al. use their definition to validate the stack safety micro-policies proposed by Roessler and DeHon [117], who use a general-purpose tagged architecture to design stack protection security policies. Unlike our capability-based calling convention, their policies do not incur an overhead when passing stack objects, but require more sophisticated tags, and a mechanism of lazy tagging to achieve the low overhead.

Tagged architectures have also been used to enforce more general properties such as information-flow control [15], secure compartmentalization [3], among other micro-policies [16]. However, micro-policies must be expertly designed in order to leverage cache and be efficient.

In a different line of work, software-based fault isolation (SFI) aims to provide process-based isolation by compartmentalizing (sandboxing) processes in different regions of the memory [151]. Our calling convention provides similar guarantees *despite* a shared stack. Recently, Kolosick et al. [81] have used an overlay semantics to characterize sufficient conditions for which so-called "heavyweight transitions" (context switching) can be safely replaced by "near zero cost transitions" akin to regular function calls. They further show that WebAssembly code compiled by a *correct* compiler would satisfy these conditions.

Finally, we remark that our unary and binary models are built on a large body of work on characterizing security through logical relations. We use a logical approach [40, 41, 145] to step-indexed Kripke logical relations [6, 21], mechanized in the Coq implementation of Iris [83]. We use private and public transitions to characterise well-bracketed control flow [39], and a new kind of transition that we call temporal transitions to characterise the temporal aspect of stack safety.

4.8 Conclusion and Future Work

We have demonstrated how directed capabilities can be used to enforce a strong degree of stack safety, including local state encapsulation, well-bracketed control flow, and temporal stack safety, with no stack clearing, and with only one additional bit. We have presented two new logical relations to reason about the integrity and confidentiality of specific examples, and proved a full abstraction result for an overlay semantics that defines our notion of stack safety. Finally, we discussed interesting subtleties of stack safety properties in a capability machine with a stack and a heap, and in the presence of stack objects crossing the boundary from caller to callee.

We have used contextual equivalence to formalize confidentiality, whereas Anderson et al. [10] and Azevedo de Amorim et al. [17] intuitively link confidentiality to a kind of non-interference property. We believe our calling convention based on directed capabilities also guarantees non-interference and in future work, it would be interesting to show this formally. To that end, one would probably have to extend the capability machine language with security labels.

We have shown how our capability machine can implement function calls, as found in higher-level languages, in a secure manner. We believe it is easy to show that it can also implement tail-calls securely and conjecture that it is also possible to implement non-standard control flow such as C-style `setjmp/longjmp` efficiently and securely. Indeed, we may implement `setjmp` by creating a Jmp capability pointing to some activation code, similarly to how call creates a return capability. These capabilities can then be safely passed up the stack to callees. `longjmp` would then be implemented by jumping to such a capability. Such an implemention is efficient as `longjmp` is just a jump, and it is not necessary to unwind the stack. It is also safe in the sense that we can guarantee temporal safety of a `setjmp` environment: a caller will not be able to `longjmp` to a `setjmp` environment set up by one of its descendants (this is similar to how we ensure WBCF and guarantee that a return capability cannot be smuggled away). Temporal confidentiality of stack frames can still be enforced, and stack frames do *not* need to be scrubbed because directed capabilities guarantee that a caller cannot read them. Previously proposed calling conventions for capability machines either cannot provide such guarantees or would require careful unwinding or extensive memory clearing.

Acknowledgements We thank the anonymous reviewers for excellent comments and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. We would also like to thank Dominique Devriese, Thomas Van Strydonck, Amin Timany, Armaël Guéneau and Frank Piessens for invaluable discussions and feedback.

Chapter 5

Iris-Wasm: Robust and Modular Verification of WebAssembly Programs

This chapter is an extended version of the following conference submission, which has been conditionally accepted to the International Conference on Programming Language Design and Implementation (PLDI) 2023:

> Xiaojia Rao, Aïna Linn Georges, Conrad Watt, Maxime Legoupil, Jean Pichon-Pharabod, Philippa Gardner, Lars Birkedal
> *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

The extension consists of

- Added technical details about the logical relation model; Section 5.5.1
- A extended presentation of the fundamental theorem of logical relations, including a proof of multiple key cases; Section 5.5.1.3
- An illustrative example of applying the logical relation model; Section 5.5.2

Abstract

WebAssembly makes it possible to run C/C++ applications on the web with near-native performance. A WebAssembly program is expressed as a collection of higher-order ML-like modules, which are composed together through a system of explicit imports and exports, using a host language, enabling a form of higher-order modular programming. We present Iris-Wasm, a mechanised higher-order separation logic building on a specification of Wasm 1.0 mechanised in Coq and the Iris framework. Using Iris-Wasm, we are able to specify and verify individual modules separately, and then compose them modularly in a simple host language featuring the core operations of the WebAssembly JavaScript Interface. Building on Iris-Wasm, we develop a logical relation that enforces robust safety: unknown, adversarial code can only affect other modules through the functions that they explicitly export. Together, the program logic and the logical relation allow us to formally verify functional correctness of WebAssembly programs, even when they invoke and are invoked by unknown code, thereby demonstrating that WebAssembly enforces strong isolation between modules.

5.1 Introduction

WebAssembly (Wasm) is a new bytecode language, supported by all major Web browsers and designed primarily to be an efficient compilation target for low-level languages such as C/C++ and Rust. It is officially specified using a formal operational semantics in the W3C Wasm 1.0 standard [119]. The formal nature of the official Wasm standard and the existence of a well-exercised language mechanisation give us a standout opportunity to define a higher-order program logic that covers the full definition of an industrial programming language. We introduce Iris-Wasm, a mechanised higher-order separation logic for Wasm 1.0 which builds on the WasmCert-Coq mechanised specification of the Wasm 1.0 language standard [160] and the Iris framework [74, 77]. In Iris-Wasm, we present an interactive formal verification framework that exactly reflects the Wasm semantics. The result is a semantic and compositional characterization of all Wasm definitions, which can be used to prove separation logic assertions about real Wasm programs, and which lays the foundation for rigorous investigations of the Wasm ecosystem.

A Wasm program is expressed as a collection of higher-order ML-like *modules*, which are composed together through a system of explicit imports and exports. This process of composing Wasm modules into a full program is not performed within Wasm itself. Instead, Wasm is embedded within a *host language*, which provides several important capabilities not available to core Wasm code, including a complex, inherently higher-order, *instantiation* operation in which the declared state of a WebAssembly module is allocated, the module's requested imports are satisfied, and the module's declared exports are registered for use in satisfying further imports requested during subsequent instantiations. The Wasm standard defines instantiation in a host-agnostic way, to be then satisfied by the specific host-language instantiation. For example, a typical Wasm program on the web will involve individual Wasm modules which are instantiated and composed together by a top-level JavaScript host script using the functions of the WebAssembly JavaScript Interface [43].

Iris-Wasm is a higher-order mechanised program logic for the W3C Wasm 1.0 industrial standard using the Iris framework, inspired by a previous Isabelle-mechanised first-order program logic for the language draft [159]. Our implementation of the Wasm run-time semantics, with its difficult constructs such as complex control-flow commands, is given directly in Iris, instead of being translated into an existing intermediate Iris language. This choice requires considerable Iris engineering, but does provide more trust in our mechanisation, as it is line-by-line close to the Wasm semantics, and should lead to the mechanisation being comparatively straightforward to extend as the standard expands. We make a minor reformulation of the host function semantics (see §5.2.2.3), so that our core Wasm semantics and program logic are properly separate from the host.

We provide a host-agnostic axiomatic characterisation of Wasm module instantiation by establishing a lemma which lifts the complex W3C Wasm 1.0 instantiation predicate to our Iris-Wasm logic, describing the state before and after instantiation using our logical assertions. We illustrate this instantiation lemma on a simple host language designed to capture the core functionality of the WebAssembly JavaScript Interface [43], and corresponding host program logic, where the soundness of our host instantiation proof rule is established using our instantiation lemma. The Iris-Wasm program logic thus gives a semantic characterization of the host-agnostic instantiation operation.

By capturing the semantics of the full Wasm 1.0 industrial standard directly, Iris-Wasm lays the groundwork for a wide range of future analyses. Iris-Wasm can be used to validate proposed extensions to Wasm such as MSWasm, a memory safe extension of Wasm [95]. It can be used to rigorously investigate compilers that either target Wasm or compile Wasm down to some low-level assembly language. Jacobs et al. [70] demonstrate that Iris can be a useful tool to prove results such as full abstraction. Iris-Wasm sets the groundwork for similar results for realistic compilers involving Wasm.

We demonstrate our compositional higher-order reasoning about Wasm modules in our host language by developing a series of examples. Our main running example is a higher-order stack example comprising a stack module and a client module. The stack module defines and exports stack functions, including a higher-order map function for the stack, and the client module imports these functions and uses some of them, including map, in its main function. Using our Wasm program logic and a program logic for the simple host we introduce, we provide specifications for both modules: the stack module's specification contains specifications for all the stack functions, and the client module's specification depends on the stack module's specification. Finally, we verify a host program which instantiates the two modules in sequence, by modularly combining the proofs for the two module specifications. In addition, we demonstrate how to reason about *reentrancy* between the host and Wasm, by having the client module invoke a host function to modify the function table to provide a different input function for subsequent applications of map. The higher-order reasoning of the Iris framework provides an ideal environment to reason about Wasm modules. Nevertheless, it's a substantial task to apply Iris to a true industrial standard. Our implementation precisely follows the design decisions of the W3C Wasm 1.0 standard, and by using a rich logic such as Iris, we have laid the foundations for deep semantic investigations of WebAssembly and its future iterations.

In a case study, we investigate the intuitive coarse-grained encapsulation property of Wasm modules, stated in the standard: 'code from a module can arbitrarily affect its own state, but can only access the state of another module through the module's exports'. Several systems rely on this important property of Wasm to provide a form of sandboxing: for example, Fastly's 'Compute@Edge' [67] platform and the RLBox tool [105]. Both depend on the encapsulation property of a module, regardless of behaviour of other modules, which are validated but not necessarily trusted. Reasoning about such modules necessarily involves the interaction between the known, verified code of one module against unknown, untrusted, and unverified code from other modules, something that cannot be done with a program logic. Building on top of Iris-Wasm, we define a relational interpretation of WebAssembly types through a unary logical relation, which is then used to verify specific *robust safety* properties of a known module, that hold even when composed with unknown modules. We demonstrate this by proving robust safety properties of our stack module composed with arbitrary clients. Our relational interpretation is entirely host agnostic, and can modularly be applied to any host language.

In summary, our contributions are:

1. Iris-Wasm, a Coq-mechanised higher-order program logic for the Wasm runtime semantics.

2. A host-agnostic module instantiation lemma, and a program logic for a simple example host language with soundness of the specific host instantiation rule proved using our general instantiation lemma.

3. A semantic interpretation of the Wasm type system, defined via a logical relations interpretation using our Wasm program logic.

4. Illustrative examples and case studies that demonstrate the expressiveness of Iris-Wasm; we show that an implementation of a higher-order stack module satisfies a very modular abstract specification; we verify a reentrant module that uses host language features to modify function tables dynamically; and we use Iris-Wasm to define and prove the properties of our logical relation, which we use to verify robust safety of higher-order examples.

All results, including soundness of the program logic and logical relations, have been formalized in Coq – we hope this foundation will prove useful to other researchers when further investigating the WebAssembly ecosystem.

5.1.1 Higher-Order Programming in WebAssembly and Reentrancy

Consider the WebAssembly snippet in Figure 5.1, which contains a module that works as a library implementing a stack of **i32**s (on the left), and a module that works as a client of that library (on the right). The library module, which the host language calls `"stack"` here, uses a **memory** (with initial size 0; some other function is in charge of allocating space for the stack) to implement a stack. The `"stack"` module exports a `"map"` function that maps a function over a stack. However, because WebAssembly is a first-order language, `"map"` does not take the function to map as an argument. Instead, `"map"` takes as argument an index, $i, into a table of 3 functions, `"tab1"`, that this module creates and exports, and calls the function at that index in the table

```
stack_module ≜
(module ;; "stack"
  (type $t1 (func (param i32) (result i32)))
  (table (export "tab1") 3 funcref)
  (memory 0)
  (func (export "map")
        (param $i i32) (param $stk i32)
   ...
   loop
     ...
     local.get $i
     call_indirect  $t1
     ...
   end
   ...)
  ...)
```

```
client_module ≜
(module ;; "client"
  (import "stack" "tab1"
    (table 3 funcref))
  (import "stack" "map"
   (func $map (param i32 i32)))
  (elem (i32.const 0) $f0 $f1 $f2)
  (func $f0 (param $n i32) (result i32)
   ...)
  ...
  (func (export "main")
        (param $stk i32) (result i32)
   i32.const 0
   local.get $stk
   call $map
   ... ;; Rest of the code
    ))
```

Figure 5.1: A module implementing a stack library, and a client module. Module boundaries enforce isolation.
This example uses the Wasm text format; below, we work directly with the AST.

using **call_indirect**. The client module imports the same shared table of functions, and uses the **elem** directive to populate it (from offset 0) with functions it defines: $f0, $f1, and $f2. It also imports the "map" function from the "stack" module as $map, and its "main" function then calls the $map function with function index 0 as argument, which makes it map $f0 on the stack.

In §5.2 we describe our program logic and we show in §5.2.2 how it can be used to give a modular specification of the stack module, and, in particular, in §5.2.3, the "map" function. A proof of the specification of the *instantiation* of the stack module is given at the end of §5.3. We emphasize that our logic supports verification of the client module relative to an abstract logical specification of the stack module; in other words, the encapsulation of the internal representation of the stack module is reflected in its specification.

We now consider a simple extension of this example to demonstrate the need for reasoning about reentrancy between WebAssembly and the host. To this end, we will let the "main" function, after the call to $map, dynamically modify the contents of the table to now contain a new function $f3 at index 0. Dynamic modification of the table cannot be performed in pure WebAssembly, as WebAssembly only has the **elem** directive available to statically provide an initial value for the elements of the table. WebAssembly code can, however, call functions defined by the host, and those may modify the state of the WebAssembly program. Thus we add an import (**import** "host" "mut" (**func** $mut (**param i32 i32**))) to the preamble of the client module and then complete the code of the "main" function with 6 more instructions: **i32.const** 0; **i32.const** $f3; **call** $mut; **i32.const** 0; **local.get** $stk; **call** $map. The first three of these call the host function $mut that we assume will modify the function table at address 0, replacing the previous value ($f0) by $f3. The last three instructions are a call to $map identical to the one at the beginning of the body of "main" function (see Figure 5.1), but this time, when mapping the 0th function from the table onto the stack, it maps function $f3 instead of $f0 like it did during the first call to $map. Thus calling "main" on a value that represents stack $[x_0,\dots,x_n]$ will modify the stack so that the argument value now represents $[f_3(f_0(x_0)),\dots,f_3(f_0(x_n))]$.

This example illustrates how programs may take advantage of the stronger expressive power of the host. In §5.2.2, we show how we deal with calls to host functions in Iris-Wasm, and in §5.3, we introduce a simple host language and a program logic for it and show how it can be used in combination with our WebAssembly program logic to reason about complex interaction between WebAssembly code and the host language code that embeds it, including this example.

5.2 Modular reasoning for WebAssembly modules

In this section, we introduce Iris-Wasm. We present our proof rules for WebAssembly language features, and outline how they are used to prove a specification for the stack module from the Introduction. For reasons of space, we only discuss selected proof rules; we stress that we have proved program logic rules for *all* of WebAssembly and used them to give full formal proofs of examples, including the stack module; see the accompanying Coq formalisation for details. Then, in §5.3, we present the operational semantics and proof rules for our host language, and show how they are used to verify the interaction of a client module with the stack module; we focus on instantiation and reentrancy. Finally, in §5.4 we discuss how our program logic is defined within the Iris program logic framework, we overview some of the generic features and proof rules we inherit from Iris, and we state the soundness and adequacy of Iris-Wasm.

5.2.1 Proof rules for basic WebAssembly stack operations

WebAssembly is a stack language with structured control. Its dynamics is specified by a small-step operational semantics on *configuration tuples* of the form $(S;F;es)$, where es is a hybrid stack of values and instructions,[1] S is the global *store*, and F is the current function *frame*. The store S contains information about the global variables, the tables, the memories and the functions declared in all modules instantiated thus far, and the frame F contains the values of all local variables, as well as an *instance* that handles indirection, as will be explained progressively below. We recall the abstract syntax in Figure 5.2.

Reductions are structural: for any program fragment[2] es that reduces to es', the same reduction can occur under a context; for example, for any lists vs of constants

[1] The standard uses '*' to stand for 'a list of', but we prefer using s as a suffix to avoid confusion with the symbol for separating conjuction, so 'es' is a list of 'e's, 'vs' is a list of 'v's, etc.

[2] For simplicity, in this paper, we conflate what WebAssembly calls 'basic instructions' and 'administrative instructions', see beginning of §5.2.2

$$\begin{aligned}
\text{(value type) } t &::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64}\\
\text{(value) } v &::= t.\mathbf{const}\ c
\end{aligned}$$

$$\begin{aligned}
\text{(function type) } \mathit{ft} &::= \mathit{ts} \to \mathit{ts}\\
\text{(immediate) } i, \mathit{min}, \mathit{max} &::= \mathit{nat}
\end{aligned}$$

$$\begin{aligned}
\text{(instructions) } e ::= {} & v \mid t.\mathbf{add} \mid \mathit{other\ stackops} \mid \mathbf{local}.\{\mathbf{get}/\mathbf{set}\}\ i \mid \mathbf{global}.\{\mathbf{get}/\mathbf{set}\}\ i \mid\\
& t.\mathbf{load}\ \mathit{flags} \mid t.\mathbf{store}\ \mathit{flags} \mid \mathbf{memory}.\mathbf{size} \mid \mathbf{memory}.\mathbf{grow} \mid\\
& \mathbf{block}\ \mathit{ft}\ \mathit{es} \mid \mathbf{loop}\ \mathit{ft}\ \mathit{es} \mid \mathbf{if}\ \mathit{ft}\ \mathit{es}\ \mathit{es} \mid\\
& \mathbf{br}\ i \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ \mathit{is} \mid \mathbf{call}\ i \mid \mathbf{call\_indirect}\ \ i \mid \mathbf{return}
\end{aligned}$$

$$\begin{aligned}
\text{(functions) } \mathit{func} &::= \mathbf{func}\ i\ \mathit{ts}\ \mathit{es}\\
\text{(memories) } \mathit{mem} &::= \mathbf{mem}\ \mathit{min}\ \mathit{max}\\
\text{(elem segments) } \mathit{elem} &::= \mathbf{elem}\ i\ \mathit{es}_{\mathsf{off}}\ \mathit{is}
\end{aligned}$$

$$\begin{aligned}
\text{(tables) } \mathit{tab} &::= \mathbf{tab}\ \mathit{min}\ \mathit{max}\\
\text{(globals) } \mathit{glob} &::= \mathbf{glob}\ \mathsf{mutable}\ t\ \ e_{\mathsf{init}}\\
\text{(data segments) } \mathit{data} &::= \mathbf{data}\ i\ \mathit{es}_{\mathsf{off}}\ \mathit{bytes}
\end{aligned}$$

$$\begin{aligned}
\text{(import descriptions) } \mathit{importdesc} &::= \mathbf{func_i}\ i \mid \mathbf{tab_i}\ \mathit{min}\ \mathit{max} \mid \mathbf{mem_i}\ \mathit{min}\ \mathit{max} \mid\\
&\quad\ \ \mathbf{glob_i}\ \mathsf{mutable}^{?}\ t\\
\text{(imports) } \mathit{import} &::= \mathbf{import}\ \mathit{string}\ \mathit{string}\ \mathit{importdesc}\\
\text{(export descriptions) } \mathit{exportdesc} &::= \mathbf{func_e}\ i \mid \mathbf{tab_e}\ i \mid \mathbf{mem_e}\ i \mid \mathbf{glob_e}\ i\\
\text{(exports) } \mathit{export} &::= \mathbf{export}\ \mathit{string}\ \mathit{exportdesc}\\
\text{(start) } \mathit{start} &::= \mathbf{Some}\ i \mid \mathbf{None}
\end{aligned}$$

$$\begin{aligned}
\text{(function instances) } \mathit{finst} &::= \{(\mathit{inst};\mathit{ts});\mathit{es}\}_{\mathit{tf}}^{\mathsf{NativeCl}} \mid\\
&\quad\ \ \{\mathit{hidx}\}_{\mathit{tf}}^{\mathsf{HostCl}}\\
\text{(table instances) } \mathit{tinst} &::= \{\mathsf{elem} : \mathit{is},\quad \mathsf{max} : \mathit{max}^{?}\}\\
\text{(memory instance) } \mathit{minst} &::= \{\mathsf{data} : \mathit{bytes},\ \mathsf{max} : \mathit{max}^{?}\}\\
\text{(global instance) } \mathit{ginst} &::= \{\mathsf{mut} : \mathit{mutable}^{?},\ \mathsf{value} : v\}
\end{aligned}$$

$$\begin{aligned}
\text{(store) } S &::= \left\{\begin{array}{l} \mathsf{funcs} : \mathit{finsts},\ \ \mathsf{globs} : \mathit{ginsts},\\ \mathsf{mems} : \mathit{minsts},\ \ \mathsf{tabs} : \mathit{tinsts} \end{array}\right\}\\
\text{(frame) } F &::= \{\ \mathsf{locs} : \mathit{vs},\ \ \mathsf{inst} : \mathit{inst}\ \}\\
\text{(module instance) } \mathit{inst} &::= \left\{\begin{array}{l} \mathsf{types} : \mathit{fts},\ \ \mathsf{funcs} : \mathit{is},\ \ \mathsf{globs} : \mathit{is},\\ \mathsf{mems} : \mathit{is},\ \ \mathsf{tabs} : \mathit{is} \end{array}\right\}
\end{aligned}$$

$$\text{(modules) } \ m ::= \left\{\begin{array}{l} \mathsf{types} : \mathit{fts},\ \mathsf{funcs} : \mathit{funcs},\ \mathsf{globs} : \mathit{globs},\ \mathsf{mems} : \mathit{mems},\\ \mathsf{tabs} : \mathit{tabs}, \mathsf{data} : \mathit{datas},\ \mathsf{elem} : \mathit{elems},\ \mathsf{imports} : \mathit{imports},\\ \mathsf{exports} : \mathit{exports},\ \mathsf{start} : \mathit{start} \end{array}\right\}$$

Figure 5.2: WebAssembly 1.0 Abstract Syntax

and es_2 of expressions, $vs \mathbin{++} es \mathbin{++} es_2$ reduces to $vs \mathbin{++} es' \mathbin{++} es_2$. We give the general meaning of contexts in §5.2.2.

The overall structure of the operational semantics is as expected for a stack language; for example, the stack $[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ \mathit{binop}]$ reduces to $[t.\mathbf{const}\ c]$, where c is the result of applying *binop* to c_1 and c_2. Let us introduce the corresponding proof rule in our program logic.

Weakest preconditions Our proof rules are phrased using Iris' *weakest precondition*. Intuitively, $\mathsf{wp}\, es\, \{w, \Phi(w)\}$ states that the program fragment *es* computes safely, and, if it terminates with result w, predicate Φ holds of w (we discuss the formal meta-theory in §5.4). This construct is close to Hoare triples, as we have the following equality in Iris:

$$\{P\}\ es\ \{w, \Phi(w)\} = \Box(P \mathrel{-\!\!*} \mathsf{wp}\, es\, \{w, \Phi(w)\})$$

The persistent modality $\Box$ simply indicates the Hoare triple is a proposition that can be freely duplicated as many times as needed.

Logical values Because we reason about *fragments* of WebAssembly programs, execution does not always terminate with a stack of WebAssembly values, but more generally with a *logical value*:

$$\begin{array}{rcl} \mathit{LogVal} \ni w & ::= & \mathbf{immV}\ vs \mid \mathbf{trapV} \mid \mathbf{brV}\ i\ vh_i \mid \mathbf{retV}\ lh_k \mid \\ & & \mathbf{call\_hostV}\ \mathit{tf}\ \mathit{hidx}\ vs\ \mathit{llh} \end{array}$$

which is one of the following:

- **immV** *vs*, the 'normal' result: a stack of WebAssembly values;
- a trap **trapV**, which represents that the program has encountered an error in its execution;
- a break (or branching) value **brV**, a return value **retV**, or a host call value **call_hostV**, which correspond to program fragments that are stuck as such, but can get unstuck when placed in an appropriate context; we explain their meaning, and the meaning of their arguments, in §5.2.2.

Accordingly, in our proof rules, the postcondition Φ takes a logical value w as an argument.

Proof rule We prove the following Iris-Wasm proof rule for binary operators:

WP_BINOP

$$\frac{[\![t.\mathit{binop}]\!](c_1, c_2) = c \quad * \quad \triangleright\Phi(\mathbf{immV}\,[t.\mathbf{const}\ c]) \quad * \quad \overset{\textsc{Fr}}{\hookrightarrow} F}{\mathsf{wp}\,[t.\mathbf{const}\ c_1; t.\mathbf{const}\ c_2; t.\mathbf{binop}\ \mathit{binop}]\left\{w, \Phi(w) * \overset{\textsc{Fr}}{\hookrightarrow} F\right\}}$$

which states that, with two constants $t.\mathbf{const}\ c_1$ and $t.\mathbf{const}\ c_2$ on the value stack, and any function frame F, if an arbitrary predicate Φ holds *later* of the result c of the binop of type t on c_1 and c_2, then this program fragment executes safely, and if it terminates (which it does in this case), Φ holds of the execution result w, because it will be the value stack $\mathbf{immV}\ [t.\mathbf{const}\ c]$. The frame resource is a special resource which will need to be included in every proof rule where we 'take a reduction step'.

We merely require that Φ holds after one step of execution, as expressed by the later $\triangleright$ modality of Iris [77]. One may choose to ignore this, but it is necessary in the presence of Iris' higher-order features, to avoid cyclicity.

5.2.2 Control and function calls

Control and function calls in WebAssembly are intricate, but still feature locality, as expected; for example, blocks can be reasoned about in isolation, and function scope is still respected. We present an approach that allows us to reason about code fragments without needing knowledge of their environment; it improves over the approach taken in the earlier Wasm program logic [159] which does not scale to higher-order programs. In this section, we show how our rules capture this locality to make reasoning tractable.

5.2.2.1 Administrative instructions

To define reduction of blocks and functions calls, WebAssembly adds an extra layer on top of the surface language, to represent intermediate states by *administrative instructions*, which are defined by the following grammar:

$$\begin{aligned} AI \ ::= \ & \mathbf{basic}\ e \mid \mathbf{trap} \mid \mathbf{invoke}\ i \mid \mathbf{label}_i\{es\}\ es\ \mathbf{end} \mid \mathbf{local}_i\{F\}\ es\ \mathbf{end} \mid \\ & \mathbf{call\_host}\ tf\ hidx\ vs \end{aligned}$$

- A **basic** instruction is a plain WebAssembly expression, as described in Figure 5.2. When clear from the context, we conflate **basic** e and e, for example in weakest preconditions.
- A **trap** represents a program that has encountered an error in its dynamic execution.
- An **invoke** represents an intermediate step when reducing a **call** or **call_indirect** .
- A **label** represents a block or a loop that is being executed.
- A **local** represents a function call that is being executed.
- A **call_host** represents a program that performs a call to a function defined the host language.

We discuss the last four kinds of administrative instructions below, as we describe control flow and function calls in WebAssembly.

5.2.2.2 Blocks, labels, and breaks

WebAssembly is somewhat unusual as an assembly-like language in that it features only structured control, including labeled breaks. We show how we use the higher-order nature of Iris to ease reasoning about the control structure of WebAssembly.

WebAssembly has (aside from function calls) two core constructs for control flow: **block**, and **loop** (and the conditional **if**, which reduces immediately to a **block**). These take as arguments a function type, and a list of expressions constituting the body of the **block** or **loop**. This body will reduce until either it becomes a list of constants and the **block** or **loop** is exited, or a **br** instruction is its first non-constant instruction. In a **block**, the body is then exited, and execution continues with whatever follows the block; and in a **loop**, the full original body of the **loop** is repeated from the beginning. The function type $ts_1 \to ts_2$ describes the $|ts_1|$ values[3] needed to enter the **block** or **loop**, and the $|ts_2|$ values that need to be on the stack if a **br** is encountered.

Because of the similarity between these two constructs, WebAssembly semantics has them both reduce to a **label** administrative instruction. $\mathbf{label}_n\{es_{\text{cont}}\}\ es_{\text{body}}\ \mathbf{end}$ is a label with body es_{body} that will execute continuation expression es_{cont} if it encounters a **br** instruction preceeded by n values. We come back later to the exact semantics of **br**. When preceeded with $|ts_1|$ values vs of the right type, $\mathbf{block}\ (ts_1 \to ts_2)\ es$ reduces to $\mathbf{label}_{|ts_2|}\{[]\}\ vs +\!\!+\ es\ \mathbf{end}$ and $\mathbf{loop}\ (ts_1 \to ts_2)\ es$ reduces to $\mathbf{label}_{|ts_1|}\{[\mathbf{loop}\ (ts_1 \to ts_2)\ es]\}\ vs +\!\!+\ es\ \mathbf{end}$.

Once the **block** or **loop** instruction has been reduced to a **label**, reduction steps can be taken in the body of the **label**. As this may happen under many nested labels, WebAssembly defines evaluation contexts lh_k, which describe stack environments consisting of k nested *labels* surrounding a *hole* $[\_]$ where the next step of execution takes place:

$$lh_0 ::= vs +\!\!+\ [\_] +\!\!+\ es \qquad lh_{k+1} ::= vs +\!\!+\ \mathbf{label}_n\{es_{cont}\}\ lh_k\ \mathbf{end} +\!\!+\ es$$

Note how only (constant) values vs can be on the left of the hole and label instructions: this enforces that we can only ‘zoom in’ on the next expression to reduce.

As expected, steps can be taken under an evaluation context: if es reduces to es', then $lh_k[es]$ reduces to $lh_k[es']$. Taking $k = 0$ yields the expected sequencing rule mentioned at the start of §5.2.1.

Correspondingly, we prove the following Iris-Wasm rule, which reduces reasoning about a program fragment that can be decomposed as $lh_i[es]$ to reasoning about $lh_i[vs]$, that is, the result vs of evaluating the expression to a list of constants, placed in the evaluation context. [4]

$$\frac{\texttt{WP\_CTX\_BIND} \qquad \mathsf{wp}\ es\ \{w, \mathsf{wp}\ lh_i[w]\ \{w', \Phi(w')\}\}}{\mathsf{wp}\ lh_i[es]\ \{w', \Phi(w')\}}$$

[3] In WebAssembly 1.0, ts_1 is always empty.

[4] The version we show here is meant for evaluation contexts with at least one label constructor; in our Coq formalisation, we prove more intricate variations of this rule, to be applied for sequencing, with for instance $lh_i[es]$ replaced with $lh_i[es_1 +\!\!+\ es_2]$.

This rule leverages the fact that in Iris, weakest preconditions are propositions themselves, and can therefore be nested. Notice how we have implicitly cast w, a logical value, into an expression when plugging it into lh_i. This is done in the intuitive way: **immV** *vs* is cast into *vs*, **trapV** is cast into the single administrative instruction [**trap**], etc.

While control flow in WebAssembly is structured, the presence of labelled breaks makes it slightly involved. A break targets a particular level of the evaluation context, and skips the rest. As a result, the default evaluation context rules provided by Iris are inadequate, and we have to build our own reasoning principles for contexts.

The **br** i instruction targets the i^{th} label from the context. Crucially, breaking relies on the instruction **br** i being in an evaluation context lh_k with $i = k$: the break index indicates what context depth is targeted. If $i > k$, the expression $lh_k[\mathbf{br}\ i]$ is stuck and can only reduce if placed in a deeper context. Correspondingly, we introduce a new type of logical values: **brV** $i\ vh_i$, representing the program fragment $vh_i[\mathbf{br}\ i]$. The *breaking context* vh_i is similar to an evaluation context lh_i, except that the meaning of the subscript i is that the context has depth *at most* i, instead of exactly i. If $i < k$, a **br** i nested in context lh_k will only break out of the i first **label**s, and the result will be in the form $lh_{k-i}[vs +\!\!+\ es]$. The break value **brV** allows to bind into any number of **label**s without needing to worry about getting stuck at a **br** i statement: when encountering such a statement, we simply bind back $i+1$ times to get a wp in a form where our rule for **br** can be applied.

5.2.2.3 Functions

There are two ways to call a function in WebAssembly: statically with **call**, or by dynamically fetching a function from a table, with **call_indirect**. We focus on the simpler direct **call** here, and explain **call_indirect** in §5.2.3.

The instruction **call** n calls the nth function declared in the current module. Indexing starts at 0 with the imported functions, followed by the functions defined in the module itself. The store S keeps a list of the *function closures* (which we describe below) of *all* the instantiated modules. This means the nth function in the current module will not always been the nth function in the store: the *instance* in the function frame F is in charge of remembering that indirection. The instance also contains this indirection information for global variables, memories, and tables.

A **call** i retrieves the address *addri* of the relevant closure in the store from the frame's instance, and reduces to **invoke** *addri*. We prove the corresponding Iris-Wasm rule:

$$\texttt{wp\_call}\quad \frac{(F.\mathsf{inst}.\mathsf{funcs}[i] = addri) * \xhookrightarrow{\text{FR}} F * \triangleright\left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\ [\mathbf{invoke}\ addri]\ \{w, \Phi(w)\}\right)}{\mathsf{wp}\ [\mathbf{call}\ i]\ \{w, \Phi(w)\}}$$

which requires ownership of the frame, not only because we are taking a reduction step, but also to know where to look up index *addri*.

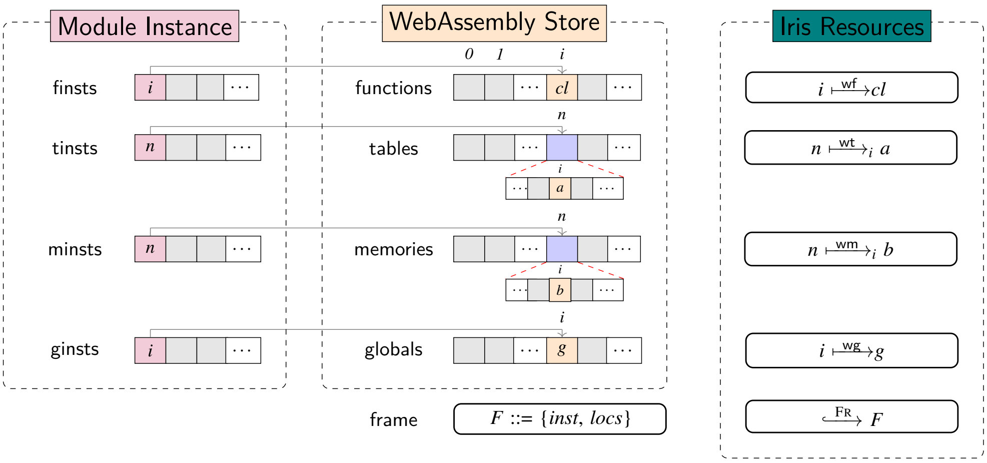

Module Instance
WebAssembly Store
Iris Resources
finsts
tinsts
minsts
ginsts
functions
tables
memories
globals
frame
0
1
i
n
cl
a
b
g
$F ::= \{inst, locs\}$
$i \xrightarrow{\mathsf{wf}} cl$
$n \xrightarrow{\mathsf{wt}}_i a$
$n \xrightarrow{\mathsf{wm}}_i b$
$i \xrightarrow{\mathsf{wg}} g$
$\xhookrightarrow{\mathrm{FR}} F$

Figure 5.3: Points-to predicates for the store and the frame

The function closures cl (also called function instances *finst* in Figure 5.2) stored in the store S are of two kinds: native and host. Let us focus first on native closures, and come back to host closures at the end of this section. The closure $\{(inst;ts);es\}^{\mathsf{NativeCl}}_{ts_1 \to ts_2}$ describes a *native* function that was defined in a WebAssembly module with instance *inst* (this is the environment for the closure), which expects arguments of type ts_1, defines additional local variables of type ts for the computation of its body, yields results of type ts_2, and has body es. When reducing **invoke**, we look up the closure in the store, and check that the stack contains the appropriate number of values to be passed as parameters to the function. If the closure is native, **invoke** is replaced with the body of the function. In order to properly encapsulate the function call, WebAssembly places the function body inside a **local** administrative instruction, and, for technical reasons described later, inside a **block**, as captured by the following Iris-Wasm proof rule:

wp_invoke_native

$$\frac{\begin{array}{c} |vs| = |ts_1| * cl = \{(inst;ts);es\}^{\mathsf{NativeCl}}_{(ts_1 \to ts_2)} * F' = \{\mathsf{locs} := vs +\!\!+ \; \mathbf{zeros}(ts); \mathsf{inst} := inst\} \\ * \; i \xrightarrow{\mathsf{wf}} cl * \xhookrightarrow{\mathrm{FR}} F \\ * \triangleright \left[\begin{array}{c} (i \xrightarrow{\mathsf{wf}} cl * \xhookrightarrow{\mathrm{FR}} F) \twoheadrightarrow \\ \mathsf{wp} \; \left(\mathbf{local}_{|ts_2|}\{F'\} \; (\mathbf{block} \; ([] \to ts_2) \; es) \; \mathbf{end}\right) \; \{w, \Phi(w)\} \end{array} \right] \end{array}}{\mathsf{wp} \; (vs +\!\!+ \; \mathbf{invoke} \; i) \; \{w, \Phi(w)\}}$$

We say more about **local** and the meaning of F' further down.

Unlike for the function frame F, we do not assert ownership of the whole store S. Instead, we rely on points-to predicates to assert ownership of specific components: the predicate $i \xrightarrow{\mathsf{wf}} cl$ asserts ownership of $S.\mathsf{funcs}[i]$ in the store.

In general, we define points-to predicates for each component of the Wasm store. Figure 5.3 illustrates all the points-to predicates used in this paper, and how they relate to the physical Wasm store. Functions and globals are referred to directly via their indices, while function tables and linear memories can be viewed as two dimensional structures, where an index is used to refer to a particular table or memory, and another index is used to refer to a particular cell within that table or memory. For example, $n \overset{\mathsf{wm}}{\longmapsto}_i b$ asserts that the i^{th} byte of memory n is b. The WebAssembly frame F tracks the scope of the currently executing function, namely its enclosing instance and local variables. The enclosing instance collects indices of all the entities of the Wasm store that the module may access, and is crucial for enforcing the encapsulation properties of Wasm modules.

Encapsulation Let us return to why the function body is placed inside a **local** and inside a **block**. The first of these is to provide proper encapsulation, as reduction of an expression nested in a **local** takes place with respect to the nested frame of the **local**: when reducing $[\mathbf{local}_n\{F_1\}\ es\ \mathbf{end}]$, one reduces es with respect to frame F_1 rather than the current function frame F.

For our native invocation, the frame used will be F'. Note that the inst field of F' is the instance that was declared in the closure (to enforce static scoping), and that the local variables in F' are the function parameters from the stack, followed by a list of zeros corresponding to the types of local variables required by the function.

We prove the corresponding proof rule for `local`:

wp_local_bind

$$\frac{\left(\overset{\textsc{Fr}}{\hookrightarrow} F_1 \mathrel{-\!\!*} \mathsf{wp}\, es \left\{ w, \begin{array}{c} \overset{\textsc{Fr}}{\hookrightarrow} F * \\ \exists F_1', \overset{\textsc{Fr}}{\hookrightarrow} F_1' * \\ \left(\overset{\textsc{Fr}}{\hookrightarrow} F \mathrel{-\!\!*} \mathsf{wp}\, [\mathbf{local}_n\{F_1'\}\ w\ \mathbf{end}]\ \{w', \Phi(w')\}\right) \end{array} \right\}\right)}{\mathsf{wp}\, [\mathbf{local}_n\{F_1\}\ es\ \mathbf{end}]\ \{w', \Phi(w')\}}$$

which is reminiscent of wp_ctx_bind; the only reason this rule looks like more of a mouthful, is that the frame changes. As discussed above, this frame change is necessary for proper encapsulation.

Finally, the reason WebAssembly puts the function body in a **block** is to allow the function body to contain a **br** (with the right index) to exit the function-body's execution. Alternatively, a **return** instruction will work like a **br**, but target the closest **local** instruction. The **return** instruction also has an associated logical value **retV** lh_k, representing the expression $lh_k[\mathbf{return}]$. Here, there is no restriction on the depth k, as a **return** is stuck under any amount of **label**s, and can only be unstuck under a **local**. The program logic rules for **return** is as follows:

wp_return

$$\frac{(lh_i[(vs \mathbin{+\!\!+} \mathbf{return})] = les) * (|vs| = n) * \triangleright \left[\mathsf{wp}\,(vs \mathbin{+\!\!+}\ es)\left\{w, \Phi(w) * \overset{\textsc{Fr}}{\hookrightarrow} F\right\}\right]}{\mathsf{wp}\,\mathbf{local}_n\{F_0\}\ les\ \mathbf{end}\left\{w, \Phi(w) * \overset{\textsc{Fr}}{\hookrightarrow} F\right\}}$$

Example Consider the increment function with body

$$es_{incr} = [\mathbf{i32.local.get}\ 0; \mathbf{i32.const}\ 1; \mathbf{i32.add}]$$

of type $[\mathbf{i32}] \to [\mathbf{i32}]$. We show that calling it on input 3 returns 4.

Define es as $[\mathbf{i32.const}\ 3; \mathbf{call}\ \$\mathsf{incr}]$, and assume that $F.\mathsf{inst.funcs}[\$\mathsf{incr}] = i$, we then prove that

$$i \xmapsto{\mathsf{wf}} \{(inst; []); es_{incr}\}^{\mathsf{NativeCl}}_{[\mathbf{i32}] \to [\mathbf{i32}]} * \xhookrightarrow{\mathrm{FR}} F \mathrel{-\!\!*} \mathsf{wp}\, es\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\}$$

Here, the first precondition asserts that we know that function number i in the store is the increment function (we denote by $inst$ the instance of the module where the increment function was defined), and the second precondition is ownership of the frame F.

We first introduce the two preconditions by moving them to a proof environment Γ. For the first step of derivation, we apply the `wp_call` rule. Some structural rules, which we have omitted here for simplicity, allow us to apply it even though the **call** instruction is preceded by a constant. To fulfill the premises of the `wp_call` rule, the resource $\xhookrightarrow{\mathrm{FR}} F$ from Γ is consumed, and it remains to prove its last premise

$$\triangleright(\xhookrightarrow{\mathrm{FR}} F \mathrel{-\!\!*} \mathsf{wp}\, [\mathbf{i32.const}\ 3; \mathbf{invoke}\ i]\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\})$$

Now we introduce the $\triangleright$, move the frame resource back to our proof environment Γ, and are left with a new weakest precondition to prove. This first proof step corresponds to the bottom-most rule of the following simplified proof-tree:

$$\begin{array}{cl}
\Gamma \vdash \mathbf{immV}\ [\mathbf{i32.const}\ 4] = \mathbf{immV}\ [\mathbf{i32.const}\ 4] & \\ \hline
\Gamma \vdash \mathsf{wp}\, [\mathbf{local}_1\{F_1'\}\ [\mathbf{i32.const}\ \mathit{4}]\ \mathbf{end}]\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\} & \texttt{wp\_local\_value} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{label}_1\{[]\}\ [\mathbf{i32.const}\ \mathit{4}]\ \mathbf{end}]\, \{w, \Phi(w)\} & \texttt{wp\_label\_value} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{i32.const}\ 3; \mathbf{i32.const}\ 1; \mathbf{i32.add}]\, \{w, \mathsf{wp}\, [\mathbf{label}_1\{[]\}\ w\ \mathbf{end}]\, \{w', \Phi(w')\}\} & \texttt{wp\_binop} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{label}_1\{[]\}\ [\mathbf{i32.const}\ \mathit{3}; \mathbf{i32.const}\ \mathit{1}; \mathbf{i32.add}]\ \mathbf{end}]\, \{w', \Phi(w')\} & \texttt{wp\_ctx\_bind} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{local.get}\ 0]\, \{w, \mathsf{wp}\, [\mathbf{label}_1\{[]\}\ w \mathbin{+\!\!+} [\mathbf{i32.const}\ \mathit{1}; \mathbf{i32.add}]\ \mathbf{end}]\, \{w', \Phi(w')\}\} & \texttt{wp\_local\_get} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{label}_1\{[]\}\ es_{incr}\ \mathbf{end}]\, \{w, \Phi(w)\} & \texttt{wp\_ctx\_bind} \\ \hline
\Gamma' \vdash \mathsf{wp}\, [\mathbf{block}([] \to [\mathbf{i32}])es_{incr}]\, \{w, \Phi(w)\} & \texttt{wp\_block} \\ \hline
\Gamma \vdash \mathsf{wp}\, [\mathbf{local}_1\{F'\}\ \mathbf{block}([] \to [\mathbf{i32}])es_{incr}\ \mathbf{end}]\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\} & \texttt{wp\_local\_bind} \\ \hline
\Gamma \vdash \mathsf{wp}\, [\mathbf{i32.const}\ 3; \mathbf{invoke}\ i]\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\} & \texttt{wp\_invoke\_native} \\ \hline
\Gamma \vdash \mathsf{wp}\, es\, \{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\} & \texttt{wp\_call}
\end{array}$$

As illustrated, we proceed by applying rule `wp_invoke_native`, leaving us with a new weakest precondition to prove with the same environment Γ. In the figure, F' is defined as $\{\mathsf{locs} := [\mathbf{i32.const}\ 3]; \mathsf{inst} := inst\}$, which is the frame where the call to the increment function needs to be executed in. Next we apply rule `wp_local_bind` in order to bind the contents of the **local**. We give up the $\xhookrightarrow{\mathrm{FR}} F$ resource in order to fulfill one premise. In its last premise, the new frame resource $\xhookrightarrow{\mathrm{FR}} F'$ is introduced back to the context, and will be the frame we use to reason within the call to the increment function. We denote by Γ' this new proof environment where we own frame F' instead of F, and let

$$\begin{aligned}
\Phi(w) = \quad & \exists F_1', \xhookrightarrow{\mathrm{FR}} F_1' \\
& * \left(\xhookrightarrow{\mathrm{FR}} F \mathrel{-\!\!*} \mathsf{wp}\, [\mathbf{local}_1\{F_1'\}\ w\ \mathbf{end}]\, \{w', w' = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\}\right)
\end{aligned}$$

which is the large postcondition in the last premise of the rule `wp_local_bind`.

The next few steps are mechanical, and we omit the details of some rules for brevity. We apply `wp_block` followed by `wp_ctx_bind` to focus on the first instruction of es_{incr}, **local**.**get**. We resolve it by applying rule `wp_local_get`, which inspects the locs field of the frame, and leaves us to prove the post-condition for 3. We apply `wp_ctx_bind` again to bind the binary operation **i32**.**add**, resolve it by applying `wp_binop`[5], and then `wp_label_value` to exit the label. It now remains to show $\Phi(\mathbf{immV}\ \ [\mathbf{i32.const}\ 4])$, which expands to

$$\begin{aligned}&\exists F_1', \xhookrightarrow{\text{FR}} F_1'\\ &* \left(\xhookrightarrow{\text{FR}} F \mathrel{-\!\!*} \mathsf{wp}\,[\mathbf{local}_1\{F_1'\}\ [\mathbf{i32.const}\ 4]\ \mathbf{end}]\,\{w, w = \mathbf{immV}\ [\mathbf{i32.const}\ 4]\}\right)\end{aligned}$$

We satisfy the existential with F', give up the resource $\xhookrightarrow{\text{FR}} F'$ from the context Γ' to satisfy the first part of the separating conjunction, and obtain $\xhookrightarrow{\text{FR}} F$ back, making our proof environment Γ again. We exit the **local** instruction (which is the function call context) by applying `wp_local_value`, and are left with our original post condition to prove, which is now trivial when substituted with the value we obtained inside **local**. This completes the detailed proof.

Example Coming back to the stack module from §5.1.1, we now outline what specifications for functions look like and, how they can be used by client modules. Take any function f from the `"stack"` module. We write its specification in the general form:

$$\Box\exists cl\ P, \forall i\ vs\ xs, \quad \Psi(P, vs, xs) \mathrel{-\!\!*} (i \xmapsto{\mathsf{wf}} cl) \mathrel{-\!\!*} \mathsf{wp}\ vs +\!\!+\ [\mathbf{invoke}\ i]\ \{w, \Phi(P, w, xs)\}$$

with Φ and Ψ some predicates specific to the function f. The persistence modality $\Box$ simply indicates this specification can be duplicated as many times as needed;[6] we omit this modality in every specification that follows, for simplicity. Note the existential quantifiers. The first one, cl, abstracts over the actual closure of function f; because it is hidden behind an existential, it is hidden from clients. The second one, P, allows the specification to reference some abstract representation predicate. In the case of the functions from the `"stack"` module, we will have an existentially quantified predicate **isStack**, which hides the data representation from clients. We put all specifications under one large existential $\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \ldots$ **isStack**, so that all specifications can share the predicate **isStack**.

The specification is thus a weakest precondition[7] on an **invoke**, with some precondition Ψ on the arguments vs given and some postcondition Φ. Both Ψ and Φ can

[5] Formally, to use the rule as it was presented earlier, one must first frame in the resource $\xhookrightarrow{\text{FR}} F'$ in order to have the postcondition be of the right form. This means that, just like for every rule we have applied so far, even though we give up ownership of $\xhookrightarrow{\text{FR}} F'$ to fulfill one premise, we still get to use it to prove the other premise.

[6] As a counterpart, proving this specification cannot rely on usage of any non-duplicable resource.

[7] In practice, we use the host weakest precondition $\mathsf{wp}_{\mathbf{HOST}}\ -\ \{-\}$ that we introduce in §5.3, as to allow functions to interact with the host via host calls. For functions that do not interact with the host, this makes no difference.

mention the existentially quantified predicate P, as well as some universally quantified variables xs. The invocation address i is linked to the function f by the condition $i \xmapsto{\mathsf{wf}} cl$, that asserts that the function body is stored at address i. Let us give the concrete Φ and Ψ used for function "push":

$$\begin{aligned}\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \dots\ \textbf{isStack}, \Big(&\forall i\ v\ x\ s, \textbf{isStack}(v,s) \mathrel{-\!\!*} (i \xmapsto{\mathsf{wf}} cl_{\text{push}}) \mathrel{-\!\!*}\\ &\mathsf{wp}\ [\textbf{i32.const}\ x; v; \textbf{invoke}\ i]\ \{w, w = \textbf{immV}\ [] * \textbf{isStack}(v, x :: s)\}\Big)\\ &* \dots \text{(other specs)}\end{aligned}$$

To present the corresponding Φ and Ψ predicates for the "map" function, we need first to introduce some aspects about higher-order code in WebAssembly, which we do in §5.2.3.

When specifying the functions from the "stack" module, we put all specifications under one large existential $\exists cl_{\text{push}}\ cl_{\text{pop}}\ cl_{\text{map}}\ \dots\ \textbf{isStack}$, so that all specifications can share the predicate **isStack**.

Given a specification written in this form, and given the resource $i \xmapsto{\mathsf{wf}} cl_{\text{map}}$,[8] a client can verify its code in the presence of a call to the imported map function: when arriving at the instruction **call** $map, `wp_call` reduces **call** to **invoke**, and now the specification shown above can be applied.

Host functions WebAssembly is meant to be defined independently of the host language in which it is embedded. However, the way the WebAssembly standard is phrased assumes that it is given some operational semantics of the host language as input, and embeds it in the operational semantics of WebAssembly. This phrasing suffices for defining the semantics of WebAssembly alone, which is what the WebAssembly standard does. However, when providing the first formal integration of WebAssembly with a separately-defined host language, we identified that this phrasing is limiting, because it prevents formally giving the semantics of the combined host and embedded language as the integration of two concrete, separately defined language.

To account for this, we modify the presentation of the WebAssembly semantics (this is our only point of departure from the Coq formalisation of Watt et al. [160]) so that the **invoke** of a host function reduces to a new **call_host** administrative instruction:

$$\texttt{invoke\_host}\quad \frac{\Big(S.\mathsf{funcs}[i] = \{hidx\}^{\mathsf{HostCl}}_{ts_1 \to ts_2}\Big) * (|ts_1| = |vs|)}{(S; F; vs \mathbin{+\!\!+} [\textbf{invoke}\ i]) \hookrightarrow (S; F; [\textbf{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs])}$$

The closure $\{hidx\}^{\mathsf{HostCl}}_{ts_1 \to ts_2}$ represents a *host* function imported from the host language that expects arguments of type ts_1 and yields results of type ts_2. The argument $hidx$ is an identifier that the host will use to determine what the desired function is. The **call_host** instruction remembers the function type tf, the 'host identifier' $hidx$ that

[8] The name of the index i and ownership of this resource are provided by instantiation when the client does the import.

allows the host language to identify which function is being called, and the function arguments vs. A **call_host** is stuck, and can only be unstuck by the host language, which typically replaces it by the return value of the call, possibly changing the frame or the store in doing so. We say more about the host interaction in §5.3.

We prove the following Iris-Wasm proof rule:

`WP_INVOKE_HOST`

$$\frac{\begin{array}{c}|vs| = |ts_1| * cl = \{hidx\}^{\mathsf{HostCl}}_{(ts_1 \to ts_2)} * \\ i \overset{\mathsf{wf}}{\longmapsto} cl * \overset{\mathrm{FR}}{\hookrightarrow} F * \triangleright \left[\begin{array}{c}(i \overset{\mathsf{wf}}{\longmapsto} cl * \overset{\mathrm{FR}}{\hookrightarrow} F) \mathrel{-\!\!*} \\ \mathsf{wp}\,(\textbf{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs)\,\{w, \Phi(w)\}\end{array}\right]\end{array}}{\mathsf{wp}\,(vs +\!\!+\ [\textbf{invoke}\ i])\,\{w, \Phi(w)\}}$$

We introduce the **call_hostV** $tf\ hidx\ vs\ llh$ logical value, representing the stuck value
$llh[\textbf{call\_host}\ tf\ hidx\ vs]$. This allows for seamless binding rules when we introduce the host language's logical rules in §5.3. Since a **call_host** instruction is also stuck if it is under a **local** or under a **label**, we remember the context llh around the **call_host** as the fourth argument of **call_hostV**. This context llh is a generalized version of lh_k, that has a hole in nested **local**s and **label**s. In the rule above, $\mathsf{wp}\,(\textbf{call\_host}\ (ts_1 \to ts_2)\ hidx\ vs)\,\{\Phi\}$ is thus a weakest precondition on a value, and it thus suffices to show that $\Phi(\textbf{call\_hostV}\ (ts_1 \to ts_2)\ hidx\ vs\ [\_])$.

For example, when specifying the `"main"` function of the extended client module from §5.1.1, one intermediate goal, when verifying the part of the code corresponding to the call to the host function `$mut`, would have the form $\mathsf{wp}\, vs +\!\!+$ **call** `$mut` $\{\Phi\}$, where vs represents the constant arguments we have pushed onto the stack prior to making the call. To prove this, one can simply apply rule `wp_call` to reduce **call** to **invoke**, and then rule `wp_invoke_host` to reduce the **invoke** to a **call_hostV** value. The computation is now reduced to a logical value, thus we now must prove that the postcondition Φ holds of the host call value. We cannot carry on to the rest of the code of the reentrant example if we stick at the WebAssembly level; this is in line with the nature of this call: it is a host call and needs interaction with the host to be unstuck. We will see in §5.3 how to reason about interaction with the host to prove the full specification of the reentrant example.

5.2.3 Higher-order code with call_indirect

As explained in §5.1.1, one can use **call_indirect** to implement higher-order functions with the help of the host language. The instruction **call_indirect** i, where i is an index into the types field of the module instance in the function frame, takes one argument k from the stack, and uses it as an index to look up the function to call in the table. The table itself is located in the store. Like for function invocation, the instance in the frame F finds the store-index ta of the correct table (i.e. the one at the head of the tables field). Now the kth element a of the table indexed ta can be looked up, and used as the index in the function closures component of the store, to find the closure cl to execute. As a side condition, the type of the closure must match the one declared by index i (that **call_indirect** takes as an immediate). Finally, [**call_indirect** i] reduces to [**invoke** a], setting cl to be invoked in the next reduction step.

We prove the following program logic rule:

$$\dfrac{\begin{array}{c}\xhookrightarrow{\textsc{Fr}} F * (F.\mathsf{inst.tabs}[0] = ta) * (ta \xmapsto{\mathsf{wt}}_k a) * (a \xmapsto{\mathsf{wf}} cl) * \\ (F.\mathsf{inst.types}[i] = \mathsf{typeof}\ cl) * \\ \triangleright \Big((ta \xmapsto{\mathsf{wt}}_k a) \mathrel{-\!\!*} (ta \xmapsto{\mathsf{wf}} cl) \mathrel{-\!\!*} (\xhookrightarrow{\textsc{Fr}} F) \mathrel{-\!\!*} \mathsf{wp}\ [\mathbf{invoke}\ a]\ \{w, \Phi(w)\} \Big)\end{array}}{\mathsf{wp}\ [\mathbf{i32.const}\ k; \mathbf{call\_indirect}\ \ i]\ \{w, \Phi(w)\}}\ \texttt{WP\_CALL\_INDIRECT\_SUCCESS}$$

Here, we use the points-to predicate for elements of the table: only ownership of the relevant kth element of the table is required. Notice how the rule passes the ownership of all three points-to predicates (frame ownership, table element ownership and function closure ownership) to the continuing weakest precondition.

Example The higher-order "map" function of our stack module in §5.1.1 calls its argument function on each element in the stack by using **call_indirect**. We have now introduced enough logical machinery to present our modular specification of "map":

$$\begin{array}{lr}
\exists cl_{\mathrm{map}}\ \mathbf{isStack}, \forall \Phi\ \Psi\ a\ v\ s\ F\ \ j\ k\ i, & (1) \\
\Box(\forall u.\, \Phi\ u \mathrel{-\!\!*} \ldots \mathrel{-\!\!*} \mathsf{wp}\ (\mathbf{i32.const}\ u; \mathbf{invoke}\ a)\ \{v, \Psi\ u\ v * \ldots\}) \mathrel{-\!\!*} & (2) \\
\quad \mathbf{isStack}\ v\ s \mathrel{-\!\!*} \mathsf{stack\_all}\ s\ \Phi \mathrel{-\!\!*} & (3) \\
\qquad (\xhookrightarrow{\textsc{Fr}} F) \mathrel{-\!\!*} (F.\mathsf{inst.tabs}[0] = j) \mathrel{-\!\!*} (j \xmapsto{\mathsf{wt}}_k a) \mathrel{-\!\!*} & \\
\qquad\qquad\qquad\qquad \ldots \mathrel{-\!\!*} (i \xmapsto{\mathsf{wf}} cl_{\mathrm{map}}) \mathrel{-\!\!*} & (4) \\
\quad\ \mathsf{wp}\ [\mathbf{i32.const}\ k; v; \mathbf{invoke}\ i]\ \{w, \exists s'.\, \mathbf{isStack}\ v\ s' * \mathsf{stack\_all2}\ s\ s'\ \Psi * \ldots\} & (5)
\end{array}$$

Let us describe the specification line by line: 1. As explained in §5.2.2, we existentially quantify over a closure cl_{map} and a predicate **isStack**, to hide our implementation of the stack and the body of the "map" function. We then universally quantify over many variables, including notably Φ and Ψ used in the specification of the mapped function, stressing this specification can be as general as needed 2. The first precondition is a specification for the mapped function; it uses two predicates Φ and Ψ to express that for any **i32** input u that satisfies Φ, the mapped function returns an **i32** result v such that Ψ relates u with v. We have used '. . . ' to elide some predicates, which are simply a copy of some of the resources from line 4, so as to allow usage of those resources (like frame ownership) in the proof of the specification of the mapped function. 3. Next, we describe the argument value v: it must represent a mathematical stack s, all elements of which satisfy Φ. This is captured by the **isStack** $v\ s$ predicate. 4. A points-to predicate for table j links the argument value k to the function index a (from the **invoke** in line 2). For brevity, we elide other side-conditions pertaining to typechecking the mapped function. At the end of the line, we have the function closure points-to predicate that links the index i of the invocation on line 5 to the "map" function closure. 5. After running "map", we have a stack with logical state s' at location v, whose elements are related one-to-one to that of the previous logical state s by Ψ. For readability, we omit the second part of the postcondition, which simply gives back all of the resources from line 4.

To prove the above specification, the `$stack` module, who has access to the actual code of the `"map"` function, simply fills in the existential quantifiers with the actual closure of `"map"` and the definition of **isStack** reflecting the actual implementation. Then all that remains is a weakest precondition to prove, which is done by applying the rules in §5.2.2: `wp_invoke_native` using hypothesis $i \overset{\mathsf{wf}}{\longmapsto} cl_{\text{map}}$, then `wp_local_bind`, to enter the local etc.

Note that we rely on the fact that our ambient logic, Iris, is a higher-order separation logic, in which weakest preconditions are just usual propositions. We stress again that the user of `"map"` does not need to know how **isStack** is defined (and in fact, we hide it with an existential quantifier surrounding the specification of the stack module, again exploiting the higher-order logic of Iris) or the physical state of the stack representation in memory: they only need to reason about the mathematical state, s; for example, stack_all only refers to s.

This example demonstrates that Iris-Wasm can be used to prove specifications for modules that cleanly hide the heavy indirection and low-level details of WebAssembly.[9] The use of **call_indirect** for higher-order programming, to call an arbitrary client function, goes beyond the 'encapsulated' fragment of WebAssembly of Watt et al. [159], and yet is captured modularly in the first line of our specification. Our accompanying Coq formalisation contains a formal proof that a simple implementation of the stack module meets the specification. We can then apply the specification to different clients. In this paper, we focus on the reentrant client introduced in §5.1.1, see §5.3, and a client that applies `"map"` to an unknown and potentially malicious imported function (see §5.5). The code for these examples, and a few more, can be found in our Coq development.

5.3 Host Language and Proof Rules

In this section, we define a minimal host language featuring the core operations of the WebAssembly JavaScript Interface. The host fulfils two important roles; first, it embeds WebAssembly and defines the interoperability between WebAssembly and the host; and, second, it implements *module instantiation*, in which the host language handles the allocation of WebAssembly states. Our minimal host language also has the ability to mutate WebAssembly function tables.

We begin by introducing the syntax of the host language and selected proof rules, with a focus on the interoperability with WebAssembly. We then detail the rules for module instantiation.

The syntax of the host language is shown in Figure 5.4. Host expressions are pairs of WebAssembly expressions and host-specific declarations; host values are pairs of WebAssembly values, and an empty list of declarations. Finally, the host state is

[9] Indeed, the specification shown here is akin to the specification for a stack module implemented in an ML-like programming language in standard Iris [20].

(import variable) $vi ::= nat$ (module variable) $vm ::= nat$ (host action id) $hidx ::= nat$

$$\begin{array}{lrcl}
\text{(declaration)} & \delta & ::= & \textbf{inst\_decl}\ vis\ vm\ vis \mid \textbf{get\_global}\ i \\
\text{(host action)} & a & ::= & \textbf{nop} \mid \textbf{print} \mid \textbf{instantiate}\ \delta \mid \textbf{call\_wasm} \mid \textbf{table}.\textbf{set} \\
\text{(import variable store)} & I & ::= & vi \hookrightarrow export \\
\text{(host state)} & H & ::= & \{\mathsf{store} : S,\ \mathsf{frame} : F,\ \mathsf{imports} : I,\ \mathsf{modules} : ms,\ \mathsf{actions} : as\} \\
\text{(host expression)} & he & ::= & (es; \delta s) \qquad \text{(host value)}\ hw ::= (vs; []) \mid (\textbf{trap}; [])
\end{array}$$

Figure 5.4: Host Syntax (definitions reference the grammar in Figure 5.2)

a record of the WebAssembly store and frame, as well as host-specific state. Host specific state has three components. First, it includes a store of export objects, to store the exports of an instantiated module, and to feed the imports of future instantiations. Note that while we call them import variables, they are used both for imports and exports. Subsequently, an *export* object refers to any object passed from one module to another, either as import or export. Second, it keeps track of a list of WebAssembly modules. Finally, to maintain the generality of host calls, host actions are indirectly referenced by indices into a list of available host actions.

To illustrate the expressive power of a host, our minimal host language includes five different host actions. **nop**, **print** and **instantiate** δ are pure operations that do not depend on host or WebAssembly store. More noteworthy are the **call_wasm** and **table**.**set** operations: **call_wasm** reduces to a WebAssembly call instruction, which opens up the possibility of reentrancy between the host and WebAssembly; **table**.**set** displays the expressive power of the host over the WebAssembly store, by mutating a given function table with a function from the WebAssembly store.

Declarations are either 1. instantiations **inst_decl** *vis vm vis*, which consist of a list of import/export variables to feed into the imports of a module (referenced indirectly by its index into the module store), whose exports are stored in the subsequent list of import/export variables, or 2. load declarations for WebAssembly globals, to load the final output of a Wasm module's main function. The host operational semantics prioritises the reduction of WebAssembly expressions over that of instantiation declarations.

Figure 5.5 defines the operational semantics of the host, where $\hookrightarrow_h$ describes the small step operations of the host language, and $\leadsto_a$ describes the semantics of the host actions. Note that host actions may mutate the WebAssembly store.

In the remainder of this section, we will discuss the proof rules of our new program logic for the host. We define our host logic using a weakest precondition predicate $\mathsf{wp}_{\textbf{HOST}}\ (es; \delta s)\ \{hw, \Phi(hw)\}$, which intuitively means that the host expression $(es; \delta s)$ does not get stuck and, if it terminates with the host value hw, then the predicate Φ holds for hw.

While the host weakest precondition is not to be confused with the Wasm weakest precondition, it shares some similarities in its memory model. The memory model of the host program logic extends the memory model of the Wasm program logic, as it

LIFTING

$$\frac{(S;F;es) \hookrightarrow (S';F';es')}{(S;F;I;ms;as;(es;\delta s)) \hookrightarrow_h (S';F';I;ms;as;(es';\delta s))}$$

HOST ACTION

$$\frac{\begin{array}{c} llh_k[\mathbf{call\_host}\, ft\ hidx\ vs] = lles \quad as[hidx] = a \\ (S;\mathsf{innermostFrame}(F, llh_k);vs;a) \rightsquigarrow_a (S';es) \quad llh_k[es] = lles' \end{array}}{(S;F;I;ms;as;(lles;\delta s)) \hookrightarrow_h (S';F;I;ms;as;(lles',\delta s))}$$

HOST ACTION: INSTANTIATE

$$\frac{llh_k[\mathbf{call\_host}\ ([] \to [])\ hidx\ []] = lles \quad as[hidx] = \mathbf{instantiate}\ \delta \quad llh_k[[]] = lles'}{(S;F;I;ms;as;(lles;\delta s)) \hookrightarrow_h (S;F;I;ms;as;(lles',\delta :: \delta s))}$$

INSTANTIATION

$$\frac{\begin{array}{c} ms[vm] = m \quad I[vi_0] = \mathbf{export}\ s_0\ exportdesc_0 \cdots I[vi_n] = \mathbf{export}\ s_n\ exportdesc_n \\ \mathsf{Instantiate}(S, m, [exportdesc_0;\cdots;exportdesc_n], ((S', inst, exports), start)) \\ I' = I[vi'_0 \leftarrow exports[0]][\cdots][vi'_m \leftarrow exports[m]] \\ es = []\ if\ start = \mathbf{None} \quad es = [\mathbf{invoke}\ i]\ if\ start = \mathbf{Some}\ a \wedge inst.\mathsf{funcs}[a] = i \end{array}}{(S;F;I;ms;as;(vs;\mathbf{inst\_decl}\ [vi_0;\cdots;vi_n]\ vm\ \ [vi'_0;\cdots;vi'_m] :: \delta s)) \hookrightarrow_h (S';F;I';ms;as;(es;\delta s))}$$

GET GLOBAL

$$\frac{F.\mathsf{inst.globs}[i] = addr * S.\mathsf{globs}[addr] = \{mutability;v\}}{(S;F;I;ms;as;(vs;\mathbf{get\_global}\ \ i)) \hookrightarrow_h (S;F;I;ms;as;(v :: vs;[]))}$$

NOP

$$(S;F;[];\mathbf{nop}) \rightsquigarrow_a (S;[])$$

CALL WASM

$$(S;F;[\mathbf{i32.const}\ i];\mathbf{call\_wasm}) \rightsquigarrow_a (S;[\mathbf{call}\ i])$$

PRINT

$$(S;F;[v];\mathbf{print}) \rightsquigarrow_a (S;[])$$

TABLE.SET

$$\frac{F.\mathsf{inst.funcs}[fidx] = func \qquad [1em]S' = S[\mathsf{tabs} \leftarrow S.\mathsf{tabs}[0][tidx \leftarrow func]]}{(S;F;[\mathbf{i32.const}\ tidx;\mathbf{i32.const}\ fidx];\mathbf{table.set}) \rightsquigarrow_a (S';[])}$$

Figure 5.5: Host operational semantics. $\mathsf{innermostFrame}(F, llh_k)$ looks through llh_k to find the innermost frame, and returns F if there is none.

includes the Wasm store. We reason about the host-specific part of the host state using three new predicates: 1. $vi \xmapsto{\mathsf{vis}} export$: a points-to predicate for the export object store; 2. $vm \xhookrightarrow{\mathsf{mod}} m$: a points-to predicate for the module store; 3. $hidx \xhookrightarrow{\mathsf{ha}} a$: a points-to predicate for the host action store. We present the host program logic in two parts: first we discuss the rules that implement interoperability between WebAssembly and the host, and second we discuss module instantiation.

Interoperability The first key to WebAssembly and host interoperability is the WebAssembly lifting step. Any reduction in the WebAssembly part of a host expression corresponds to a step in the host expression, as captured by the following bind rule:

$$\texttt{WP\_LIFT\_WASM}\quad \frac{\mathsf{wp}\, es\, \{w, \mathsf{wp}_{\mathbf{HOST}}\,(w; \delta s)\,\{hw, \Phi(hw)\}\}}{\mathsf{wp}_{\mathbf{HOST}}\,(es; \delta s)\,\{hw, \Phi(hw)\}}$$

Note that w may be a logical value, in particular a suspended host call from Wasm to the host, which can now be resolved via the host proof rules for **call_host**. Recall the definition of a stuck host call: the **call_host** *tf hidx vs* administrative instruction is considered stuck in any nested WebAssembly context *llh*, and is interpreted as the logical value **call_hostV** *tf hidx vs llh*, in which *hidx* refers to the host action identifier which is storing the executing host action, *tf* refers to its type, and *vs* refers to the parameters of the invocation. Each host action is resolved via a different proof rule.

In particular, one such host action is a call in the other direction, from the host to Wasm. In that case, the inner **call_wasm** action, performed by the host function *hidx*, reduces to the WebAssembly instruction **call** as follows.

$$\texttt{WP\_HOST\_ACTION\_CALL\_WASM}\quad \frac{\begin{array}{c} hidx \xhookrightarrow{\mathsf{ha}} \mathbf{call\_wasm} \ast \\ \triangleright\,(hidx \xhookrightarrow{\mathsf{ha}} \mathbf{call\_wasm} \mathrel{-\!\!*} \mathsf{wp}_{\mathbf{HOST}}\,(llh[\mathbf{call}\ i]; \delta s)\,\{hw, \Phi(hw)\}) \end{array}}{\mathsf{wp}_{\mathbf{HOST}}\,(llh[\mathbf{call\_host}\ tf\ hidx\ [\mathbf{i32.const}\ i]]; \delta s)\,\{hw, \Phi(hw)\}}$$

Reentrant example We now have all we need to prove a specification for the extended (reentrant) client introduced in §5.1.1. This specification will be parametrized with specifications for all the functions from the stack module (and thus with all the existentials of those specifications, most importantly the **isStack** predicate), and can be modularely combined with a specification for the stack module.

Our specification could look like this:

$$\begin{array}{c} \exists cl_{\mathrm{main}}, \forall v\ x_1\ \ldots\ x_n\ i\ hidx, \quad \mathbf{isStack}\ v\ [x_1, \ldots, x_n] \mathrel{-\!\!*} i \xmapsto{\mathsf{wf}} cl_{\mathrm{main}} \mathrel{-\!\!*} \\ \mathbf{OwnClosures}([\mathsf{\$f0}; \mathsf{\$f3}; \mathsf{\$map}]) \mathrel{-\!\!*} \\ \mathsf{\$mut} \xmapsto{\mathsf{wf}} \{hidx\}^{\mathsf{HostCl}}_{[\mathbf{i32};\mathbf{i32}] \to []} \mathrel{-\!\!*} hidx \xhookrightarrow{\mathsf{ha}} \mathbf{table.set} \mathrel{-\!\!*} \\ \mathsf{wp}_{\mathbf{HOST}}\,([\mathbf{i32.const}\ v; \mathbf{invoke}\ i], [])\,\{hw, \mathbf{isStack}\ hw\ [f_3(f_0(x_1)), \ldots, f_3(f_0(x_n))] \ast \ldots\} \end{array}$$

The elided postconditions give back all the preconditions; **OwnClosures**(fs) asserts ownership, for all functions $f \in fs$, of a closure cl_f. For the function $map imported from the stack module, the closure is the one referenced in the specification of the stack module. In order to carry out our proof, we assume we are given specifications for functions $f0 and $f3 that reference cl_{f_0} and cl_{f_3}.

To prove this specification, we fill in the existential quantifier for cl_{main} with the actual code of the "main" function. Now we apply `wp_lift_wasm` to bring ourselves to proving a WebAssembly weakest preconditon: the postcondition now becomes $w, \mathsf{wp}_{\mathbf{HOST}}\, w\, \{hw, \Phi(hw)\}$ where Φ is the postcondition in the weakest precondition shown above. We can now begin the proof just like we proved all the specifications for the functions in the stack module: we apply `wp_invoke_native`, then `wp_local_bind`, etc.

As showcased in §5.2.2, the WebAssembly weakest precondition gets stuck on a value when it arrives at the host call: we now need to show that the postcondition holds of the **call_hostV** value, i.e. that

$$\mathsf{wp}_{\mathbf{HOST}}\, llh[\mathbf{call\_host}\ tf\ hidx\ vs]\, \{hw, \Phi(hw)\}$$

where llh is the context in which the host call was, containing for instance all the code that follows the host call. To prove this, we have a rule `wp_host_action_table_set` similar to rule `wp_host_action_call_wasm` shown above, that, given our knowledge of $n \xmapsto{\mathsf{wt}}_0$ $f0, gives back $n \xmapsto{\mathsf{wt}}_0$ $f3, and brings us to prove a (host) weakest precondition statement on the code that follows the host call, with this new function at the 0th place in the table. We can prove this by lifting to WebAssembly and carrying out the proof in the WebAssembly program logic until the end.

Module instantiation While WebAssembly 1.0 does not depend on any particular host language, it does define a *specification* for module instantiation. Any host language is tasked with implementing instantiation according to that specification. We thus conceptually distinguish between the parts of module instantiation pertaining to the official WebAssembly specification, and the parts that deal with the host language. $\mathsf{Instantiate}(S, m, exportdescs, ((S', inst, exports), start))$ defines the specification for module instantiation. The full definition is quite elaborate; we refer to the Coq mechanisation for all details, and provide an intuitive overview here. In essence, it states that $inst$ is the result of instantiating module m while importing $exportdescs$, $exports$ are the resulting exports, and S' is the resulting WebAssembly store, in which all the relevant state has been allocated.

The specification enforces various side conditions. First, the module must be well typed according to a list of relevant import and export types. Next, it asserts the necessary operational conditions on the allocated state and created instance; that all the fields of the instance are properly initialised (e.g. any function table is initialised with the proper elements as defined by the module), that all the initialised values are within the bounds of the initialised object, and finally that the start function is either empty, or refers to a function of the module of type $[] \to []$.

The instantiation specification specifies the outcome of module instantiation on the WebAssembly store. Note that the specification is host language agnostic. The semantic outcome of instantiation on the WebAssembly store ought likewise to be independent of the host language that implements it. The following lemma captures the effects of instantiation on the interpretation of the WebAssembly store as Iris resources, according to the host agnostic instantiation specification. The lemma is thus independent of any host language definition.

Lemma 27 (Module Instantiation Resource Allocation)**.**

$$\begin{array}{ll} \textit{If} & \vdash m : \mathit{timps} \to \mathit{texps} \;\wedge\; \mathsf{constInits}(m) \\ \textit{and} & \mathsf{Instantiate}(S, m, \mathit{imports}, ((S', \mathit{inst}, \mathit{exports}), \mathit{start})) \\ \textit{then} & \mathsf{resourcesImports}(m, \mathit{imports}, \mathit{timps}, \mathit{wfs}, \mathit{wts}, \mathit{wms}, \mathit{wgs}) * \mathsf{stateInterp}(S) \\ & \Rrightarrow\!\!* \mathsf{resources}(m, \mathit{imports}, \mathit{timps}, \mathit{wfs}, \mathit{wts}, \mathit{wms}, \mathit{wgs}, \mathit{start}, \mathit{inst}) * \mathsf{stateInterp}(S') \end{array}$$

For readability, we omit the technical details behind some of the above predicates. It suffices to know the following: $\mathsf{constInits}$ limits the global initialisers and offsets to be constants, $\mathsf{resourcesImports}$ defines the points-to predicate associated with each import in $\mathsf{imports}$, and $\mathsf{resources}$ defines all the points-to predicates associated to the created instance *inst*, including those that were previously imported. The variables *wfs*, *wts*, *wms* and *wgs* are maps that summarise the values of functions, tables, memories and globals of the created instance. (The $\Rrightarrow\!\!*$ modality is used in Iris to update ghost resources [77].) Using Lemma 27, we can then prove a host weakest precondition rule for host instantiation, that we will refer to as `wp_host_instantiate`.

Example The complete stack module is an instantiation declaration, which exports closures for push, pop, new_stack, is_empty, is_full, stack_length and map, as well as the function table invoked by map. We recall that exports are passed via indices into the import variable store.

$$\mathit{vm} \triangleq \{0 \mapsto \mathsf{stack\_module}\} \quad \mathit{host\_program} \triangleq ([], [\textbf{inst\_decl}\;[]\;0\;[0,1,2,3,4,5,6,7]])$$

The Iris-Wasm specification of the complete stack module from §5.1.1 is as follows (we elide the exporting of the table, for simplicity):

$$\begin{array}{l} \exists \mathsf{stack\_module}, \forall i\; js, (i \xhookrightarrow{\mathsf{mod}} \mathsf{stack\_module}) \mathrel{-\!\!*} \left(\mathop{\huge *}_{j \in js} j \xmapsto{\mathsf{vis}} - \right) \mathrel{-\!\!*} \\ \quad \mathsf{wp}_{\mathbf{HOST}}\,([]; [\textbf{inst\_decl}\;[]\;\mathsf{i}\;js]) \left\{ \begin{array}{l} \exists cl_{\text{push}}\; cl_{\text{pop}}\; \ldots\; cl_{\text{map}}, \textbf{isStack}, \text{spec\_push} * \\ \quad \text{spec\_pop} * \ldots * \text{spec\_map} * \\ \quad \mathop{\huge *}_{j \in js} j \xmapsto{\mathsf{vis}} \mathsf{function\_export}\; cl_j \end{array} \right\} \end{array}$$

spec_push is the specification of the `"push"` method shown earlier. Likewise for the other specifications mentioned in the postcondition. Both the contents of the `$stack` module and the implementations of the stack operations are hidden from clients because of the existential quantifiers.

Figure 5.6: Lines of code of the Iris development, as given by cloc

helpers	language	rules	instantiation	host	examples	logrel	stack
11836	3685	7123	6828	2339	2754	8145	8787
total = 51497							

This stack module specification is proven by applying rule `wp_host_instantiate`, which populates the value import stores and gives ownership of all the resources necessary for the stack module operations, and then we apply the specifications for the stack operations shown in §5.2.3.

With this specification for the stack module and a similar one for the client module (parametrised by the specification of the stack), we verify the complete stack program (a sequence of instantiations) in our Coq formalisation.

5.4 Mechanization in the Iris Framework

We implement and prove the Iris-Wasm proof rules in this paper in the Iris framework in the Coq proof assistant. Iris was originally developed to reason about programs with complex concurrency; however, the same mechanisms have proven useful to reason about complex sequential programs such as the awkward example, as demonstrated for example by Georges et al. [55]. In this paper, we focus our presentation on the novel, language-specific proof rules we introduce and prove, but our program logic also inherits many other logical constructs and proof rules from Iris which we make use of in our development. We have already mentioned the 'later' modality, $\triangleright$, which avoids circularities in the presence of the higher-order features of Iris, and which can be used to define guarded recursive predicates in Iris, as well as the 'persistence' $\square$ modality, the 'update' $\dot{|}\!\!\Rrightarrow$ modality, and its wand version $\Rrightarrow\!\!\ast$. Other features we use include the frame rule, non-atomic invariants, ghost state, and other proof rules like Löb induction; for a thorough introduction to those, see Jung et al. [77].

We prove all our proof rules in Iris, with respect to the default definition of the weakest precondition predicate (with an extra requirement that the frame resource holds for every step of reduction; alternatively, we could have made the frame be part of the Iris 'state') instantiated to refer to the Coq formalisation of the official WebAssembly 1.0 operational semantics by Watt et al. [160].

The adequacy theorem of Iris [77, §6.4] then yields the final desired soundness theorem, which intuitively says that if a weakest precondition for a WebAssembly or host program has been proved in Iris-Wasm, then it does indeed mean that the program runs safely, according to the official WebAssembly 1.0 operational semantics, or the host language that embeds it. An example of the latter can be found in the Coq mechanisation.

The size of the full Iris development is summarized in Figure 5.6. The logrel folder contains files for a case study presented in the following section, while stack contains the full stack module and associated clients.

The stack module, with a binary size of 637 bytes, is defined in around 200 lines code in Coq, with the module type checking done in 300 lines of code using the type checker from Watt et al. [160]. The module specification is fully verified using the Iris-Wasm logic in around 3800 lines of code in Coq, where 2100 lines are used to verify each of the module function specifications, and the remaining code is used to prove the top level instantiation specification and auxiliary lemmas. Such a ratio between program and proof size may hint at a substantial verification effort. However, it's important to note that it reflects a version of Iris-Wasm without a bespoke proof mode; an interesting line of future work is to extend Iris-Wasm with various automation techniques, such as Mulder et al. [103], and use it to prove specifications of large real-world programs.

5.5 Case Study

We showcase the utility of our program logic through a case study[10]. The goal is to leverage the coarse-grained encapsulation guarantees of WebAssembly modules to prove robust safety of two scenarios involving some interaction between a known module and an unknown, potentially malicious, module. While the coarse-grained encapsulation properties granted by modules are relatively shallow (one module cannot interact with the internals of another), the reasoning principles are not: not only are we reasoning about unknown code, the desired robust safety property can be subtle, and highly specific to the particular implementation of a robustly safe module. We emphasize that we do not seek to either define or prove encapsulation as a meta-property, rather, we define and apply a methodology to prove robust safety of specific modules.

WebAssembly's modules are designed to allow trusted code to encapsulate its local state (e.g. variables and memory), by limiting what is shared with untrusted modules via imports and exports. This encapsulation is meant to hold no matter what other modules do, either by accident or by malice, and thus does not rely on compliance. Modules can take advantage of this encapsulation to guarantee various safety properties. To prove those properties formally, we may need to reason about the interaction between known, trusted code and unknown, untrusted code. We have thus far presented a program logic to reason about known code only. In this case study, we use the program logic to build a method to reason about the instantiation of unknown code, and use it to prove the *robust* safety of known code, that is, safety even when composed with adversarial code.

[10] Our Coq mechanization also includes another case study of a program that uses recursion through the store, by applying a host call to mutate the function table, known as Landin's Knot.

$m_{client} \triangleq$

```
(module ;; Another Stack Client
(import "adv" "f" (func $f (param i32) (result i32)))
(import "stack" "map" (func $map (param i32 i32)))
  ... ;; import global g and the remaining stack module
(elem (i32.const 0) $f) ;; populate table with imported function
(func $main (local $i i32)
  call $new_stack; ... ; const 4; call $push;
  local.get $i; const 2; call $push;
  local.get $i; call $map;
  local.get $i; call $stack_length; global.set $g))
```

```
stack_client ≜
inst_decl [] "stack" ["tab";...;"pop"]
inst_decl [] "adv" ["f"]
inst_decl ["f";"g";"tab";...;"pop"] "client" []
```

Figure 5.7: Robust safety example: applying map on an imported function

5.5.1 A Relational Interpretation of WebAssembly Types

The methodology is based on a relational interpretation of WebAssembly types, built on top of our Iris-Wasm program logic, by defining logical relations for each WebAssembly type. The key idea is to interpret the types of primitives, functions, etc., all the way to module types, as propositions in Iris-Wasm. The methodology of defining logical relations in Iris is well known [55, 76, 83, 138], but here it is for the first time applied to the type system of a full industrial standard, namely the WebAssembly type system. We define semantic interpretations for all WebAssembly types. That includes all the internals of a module, and in particular it includes the types of exports and imports. We say that an import object is safe to share, or valid, if it is in the appropriate relation. All the results in this section have been formally proved in Coq. We give an overview here, and refer the reader to the accompanying Coq code for the full definition of the relational interpretation of WebAssembly types.

5.5.1.1 The WebAssembly Type System

WebAssembly comes equipped with two core typing judgments, (1) a typing judgment for instruction sequences, and (2) a typing judgment for modules.

(1) $C \vdash es : ts \rightarrow ts'$

(2) $\vdash m : timps \rightarrow texps$

The typing judgment for instructions assigns a function type to every sequence of instructions *es*, under some *typing context* *C*. The typing judgment for modules holds when all its individual components are well typed. A module is a closed definition, and as such does not require a context for validation. Here, following the WebAssembly formalization in Coq [160], we use the module typing judgment to explicitly specify the types of imports and exports.

The typing of instructions In the typing of instructions, the assigned function type $ts \to ts'$ specifies the required type of the input stack ts, and the type of the resulting output stack ts'. Many simple instructions do not refer to the typing context. For example, the rule for binary operations is defined as follows, where *binop* stands for any WebAssembly binary operation:

$$\frac{}{C \vdash t.\mathit{binop} : [t;t] \to [t]}$$

However, other instructions refer to the ambient state of the surrounding module. For instance, **store** and **load** instructions refer to the current memory state, and are valid only such a memory exists within the current scope.

$$\frac{C.\mathsf{memory} \neq [] \qquad \mathit{side\ conditions\ on\ flags}}{C \vdash t.\mathbf{load}\,\mathit{flags} : [\mathbf{i32}] \to [t]}$$

Under this interpretation, the typing context reflects the type of the enclosing module instance. Indeed, its definition is a record that resembles the definition of a module:

$$C \triangleq \left\{ \begin{array}{l} \mathsf{types} : ts, \ \ \mathsf{func} : \mathit{fts}, \\ \mathsf{global} : \mathit{gts}, \ \ \mathsf{table} : \mathit{tabs}, \\ \mathsf{memory} : \mathit{mems}, \\ \mathsf{locals} : ts, \ \ \mathsf{labels} : \mathit{tss}, \ \ \mathsf{return} : ts^{?} \end{array} \right\}$$

The typing context specifies the list of declared types, the type of each function within scope, the type of each global within scope, and the size limits of accessible function tables and memories. In each of these cases, an empty list indicates that no object of that category is within scope.

Beyond the type of the enclosing module, the typing context additionally tracks the runtime information pertaining to the enclosing frame and evaluation context. Recall that the frame tracks the local state of the currently executing function. When invoking a new function closure, the frame is updated to store the enclosed local state, and the body of the function is placed inside a **local** wrapper, to which the body can then optionally return to via the **return** instruction, provided the value stack has the proper return type. The typing context tracks the existence of such a return target in the return field, where **None** indicates there is no valid return target, while **Some** *ts* indicates that there is a return target, and that the expected return type is *ts*.

$$\frac{C.\mathsf{return} = \mathbf{Some}\ ts}{C \vdash \mathbf{return} : ts_1 +\!\!+\ ts \to ts_2}$$

Note that the typing succeeds for any final output type ts_2, since there are no restrictions to what gets executed after returning.

Similarly, the labels field tracks all the valid break targets, and their associated block types. Each new **block** generates a new break target at the head of the list.

$$\frac{C[\mathsf{labels} = ts_2 :: C.\mathsf{labels}] \vdash es : ts_1 \to ts_2}{C \vdash \mathbf{block}\ (ts_1 \to ts_2)\ es : ts_1 \to ts_2}$$

Thus, the i^{th} element of the labels list corresponds to the expected value stack type upon executing a **br** *i* instruction.

$$\frac{C.\mathsf{labels}[i] = ts}{C \vdash \mathbf{br}\ i : ts_1 +\!\!+\ ts \to ts_2}$$

Finally, the **locals** field tracks the types of the local variables of the current frame, which is relevant for any local **get** or **set** instruction.

$$\frac{C.\mathsf{locals}[i] = t}{C \vdash \mathbf{local.get}\ i : [] \to [t]} \qquad \frac{C.\mathsf{locals}[i] = t}{C \vdash \mathbf{local.set}\ i : [t] \to []}$$

The typing of modules WebAssembly modules are valid if each of their components are valid. Functions are valid if their body is well typed, globals are valid if their initial value is well typed, while memories and tables are valid when their size fits the optional maximum allowed sizes and when their initialization is well-formed. In particular, that means tables are valid if they are initialized to valid functions (as determined by the elem field). Imports are unknown, and thus statically unrestricted, while exports must be differently named, and the declared export type must match the type of the exported object.

The typing of functions and globals depend on a typing context *C*, which is statically derived from a module definition *m* as follows (in which *fts* and *gts* are lists of function types, that respectively correspond to the declared functions and globals types):

$$\begin{aligned}
impts_{func} &= \textit{type of all the function import descriptions} \\
impts_{tab} &= \textit{size limits of all the table import descriptions} \\
impts_{mem} &= \textit{size limits of all the memory import descriptions} \\
impts_{glob} &= \textit{type of all the global import descriptions} \\
C &\triangleq \left\{ \begin{array}{l}
\mathsf{types} = m.\mathsf{types}, \\
\mathsf{func} = impts_{func} +\!\!+ fts, \mathsf{global} = impts_{glob} +\!\!+ gts, \\
\mathsf{table} = impts_{tab} +\!\!+ m.\mathsf{tabs}, \\
\mathsf{memory} = impts_{mem} +\!\!+ m.\mathsf{mems}, \\
\mathsf{locals} = [], \mathsf{labels} = [], \mathsf{return} = \mathbf{None}
\end{array} \right\}
\end{aligned}$$

Note that each function is validated against a context that contains all available function types, thus allowing functions to be mutually recursive. Recall that in the instantiation of *m*, all functions are instantiated as function closures. When validating the type of a declared function **func** *i ts es*, we thus want to validate not just the type

$$\boxed{\mathscr{V}[\![ts]\!] : \mathit{LogVal} \to \mathit{iProp}}$$

$$\begin{array}{rcl} \mathscr{V}_0[\![t]\!](v) & \triangleq & \exists c, v = t.\textbf{const}\ c \\ \mathscr{V}[\![[t_1, \cdots, t_n]]\!](w) & \triangleq & w = \textbf{trapV} \vee \\ & & \exists v_1, \cdots, v_n, w = \textbf{immV}\ \ [v_1, \cdots, v_n] \\ & & \wedge \mathscr{V}_0[\![t_1]\!](v_1) \wedge \cdots \wedge \mathscr{V}_0[\![t_n]\!](v_n) \end{array}$$

$$\boxed{\mathscr{F}\!\mathit{rame}[\![ts]\!]_{inst} : \mathit{Frame} \to \mathit{iProp}}$$

$$\begin{array}{rcl} \mathscr{F}\!\mathit{rame}[\![ts]\!]_{inst}(F) & \triangleq & [\text{NaInv} : \top] * \xhookrightarrow{\text{FR}} F \\ & & * \exists vs, F = \{inst; vs\} * \ \mathscr{V}[\![ts]\!](\textbf{immV}\ vs) \end{array}$$

$$\boxed{\mathscr{E}_0[\![ts]\!]_* : \mathit{Expr} \to \mathit{iProp}} \qquad \boxed{\mathscr{E}[\![ts]\!]^*_* : \mathit{Lholed} \times \mathit{Expr} \to \mathit{iProp}}$$

$$\mathscr{E}_0[\![ts]\!]_{(F,hfs)}(es) \ \triangleq \ \mathsf{wp}\, es \left\{ w, \begin{array}{l} (\mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]_{hfs}(w)) \\ *[\text{NaInv} : \top] * \xhookrightarrow{\text{FR}} F \end{array} \right\}$$

$$\mathscr{E}[\![ts]\!]^{\tau_{\mathsf{lbs}}, \tau_{\mathsf{ret}}}_{(\tau l, inst, hfs)}(lh, es) \ \triangleq \ \mathsf{wp}\, es \left\{ w, \begin{array}{l} \left(\begin{array}{l} \mathscr{V}[\![ts]\!](w) \vee \mathscr{H}[\![ts]\!]^{\tau_{\mathsf{lbs}}, \tau_{\mathsf{ret}}}_{(\tau l, inst, hfs)}(w) \\ \vee \mathscr{B}r[\![\tau_{\mathsf{lbs}}]\!]^{\tau_{\mathsf{ret}}}_{(\tau l, inst, hfs)}(w, lh) \\ \vee \mathscr{R}et[\![\tau_{\mathsf{ret}}]\!]_{(\tau l, inst)}(w) \end{array} \right) \\ * \exists F, \mathscr{F}\!\mathit{rame}[\![\tau l]\!]_{inst}(F) \end{array} \right\}$$

Figure 5.8: Logical relations for values, frames and expressions

of *es*, but that of a local and labeled block enclosing *es*. As such, we update the typing context C to reflect the generated **local** and **label** wrappers, as well as the type of its locals (including both the input parameter type and the declared locals type). Finally, its type must be the i^{th} type of the module (which must correspond to the type declared in *fts*). Let $m.\mathsf{types}[i] = ts_1 \to ts_2$. The function is then validated as follows:

$$C[\mathsf{locals} = ts_1 +\!\!+\, ts, \mathsf{labels} = [\ [ts_2]\ ], \mathsf{return} = \textbf{Some}\ ts_2] \vdash es : ts_1 \to ts_2$$

5.5.1.2 A Logical Relation for the WebAssembly Type System

The ultimate goal is to define relational interpretations for each WebAssembly primitive, including that of a module, so we can define a *semantic* typing judgment, that we can use to reason about the execution of unknown WebAssembly code. Note that one particular goal is to keep all definitions host agnostic. To facilitate this goal, many of the ensuing definitions will be parameterized by a list of possible host call targets. This list is determined upon instantiation, from the list of imported functions.

We begin with the relational interpretation of value and function types, as unary logical relations built on top of Iris-Wasm. Due to the intrinsic connection between the execution of a function and its frame, the latter will depend on the relational interpretation of the local value types of a particular function, which we interpret in the *frame relation*. Henceforth, we will refer to a primitive as *valid* when it inhabits the appropriate logical relation.

Figure 5.8 defines the unary logical relation for values, frames, and expressions. The value relation $\mathscr{V}[\![ts]\!] : \mathit{LogVal} \to \mathit{iProp}$ is inhabited either by values of type *ts*, or by a trap expression (it is always valid for a program to trap). Recall that *LogVal* is the logical values resulting from execution of WebAssembly program fragments, and includes immediates, traps, and stuck break and return expressions. Such stuck expressions are purposefully excluded from the value relation.

WebAssembly programs have local access to their particular frame. In the typing judgment, the local variables in a frame are given a specific type. Well-typed programs preserve these types. We define a frame relation $\mathscr{F}\!\mathit{rame}[\![ts]\!]_{inst} : \mathit{Frame} \to \mathit{iProp}$ to capture all the non-persistent resources that are *owned* by the currently executing program, and the invariants they must adhere to. This includes the key to open non atomic invariants, $[\mathrm{NaInv} : \top]$, the frame resource $\xhookrightarrow{\mathrm{FR}} F$, and the guarantee that the local variables in F satisfy $\mathscr{V}[\![ts]\!]$. The frame relation is parametrised by an instance *inst*, to enforce that F acts as the environment of that particular instance.

We present two expression relations;

$$\mathscr{E}_0[\![ts]\!]_{(F,hfs)} : \mathit{Expr} \to \mathit{iProp} \quad \text{and} \quad \mathscr{E}[\![ts]\!]^{\tau_{\mathsf{lbs}},\tau_{\mathsf{ret}}}_{(\tau l,inst,hfs)} : \mathit{Lholed} \times \mathit{Expr} \to \mathit{iProp}.$$

Both are parametrised by a list of possible host call identifiers and host function type pairs, *hfs*. The former defines the valid execution of a closed WebAssembly program, while the latter defines the valid execution of a WebAssembly program in some nested evaluation context *lh*. In other words, the latter may contain a break or return instruction that targets *lh*, while the former either loops, or reduces to an immediate, a trap, or a suspended host call. Each is defined in terms of the weakest precondition from Section 5.2, and differ merely in that the latter has cases for break and return values in the postcondition. The relation for closed programs remembers the previous frame, and reinstates it in the postcondition.

A suspended host call is handled by the host relation $\mathscr{H}[\![ts]\!]_{hfs}$. Since our goal is to keep the logical relation host-agnostic, the host relation uses the type of the host call to impose a valid continuation of the suspended call. More precisely, a suspended host call value $\mathbf{call\_host}(ts \to ts', h, vs, llh)$ is valid when: $(h, ts \to ts') \in hfs$, $\mathscr{V}[\![ts]\!](vs)$, and for any value of type ts', the expression relation holds for that value, plugged into evaluation context *llh*. Similar to the expression relation, there is a host relation both for open and closed WebAssembly programs.

The break relation $\mathscr{B}r[\![\tau_{\mathsf{lbs}}]\!]^{\tau_{\mathsf{ret}}}_{(\tau l,inst,hfs)}(w, lh)$ applies the expression relation to a break instruction within the sub-context *lh* of appropriate size, and the remaining layers as the new surrounding evaluation context.

$\mathscr{R}et$ similarly applies the expression relation to the appropriate return target, as described by the evaluation context. We refer to the Coq formalization for their formal definitions.

Next, we give the relational interpretation of module typing, primarily inhabited by *instances*. As we will see, the interpretation of module types via the instance relation,

$\boxed{\mathscr{I}[\![C]\!]_{hfs} : Instance \to iProp}$

$$\mathscr{I}\left[\!\!\left[\begin{Bmatrix} \mathsf{types} = ts,\ \mathsf{func} = fts, \\ \mathsf{global} = gts,\ \mathsf{table} = [tt,\cdots], \\ \mathsf{memory} = [mt,\cdots], \\ \mathsf{locals},\mathsf{labels},\mathsf{return} = \cdots \end{Bmatrix}\right]\!\!\right]_{hfs} \left(\begin{Bmatrix} \mathsf{types} = ts',\ \mathsf{funcs} = fs \\ \mathsf{globs} = gs, \\ \mathsf{tabs} = [t,\cdots] \\ \mathsf{mems} = [m,\cdots] \end{Bmatrix}\right) \triangleq$$
$$ts = ts' \ast \mathop{\ast}_{f\in fs; ft\in tfs} \mathscr{F}unc[\![ft]\!]_{hfs}(f) \ast \mathop{\ast}_{g\in gs; gt\in gts} \mathscr{G}lob[\![gt]\!](g)$$
$$\ast\, \mathscr{T}able[\![tt]\!](t) \ast \mathscr{M}em[\![mt]\!](m)$$

$\boxed{\mathscr{C}tx[\![C]\!]_{(inst,hfs)} : Lholed \to iProp}$

$$\mathscr{C}tx[\![\{\cdots;\tau l;\tau_{\mathsf{lbs}};\tau_{\mathsf{ret}}\}]\!]_{(inst,hfs)}(lh) \triangleq \mathsf{StructuralCond}(\tau_{\mathsf{lbs}},lh)\ast \mathop{\ast}_{j\mapsto ts\in\tau_{\mathsf{lbs}}} \mathscr{K}[\![ts]\!]^{\tau_{\mathsf{lbs}},\tau_{\mathsf{ret}}}_{(\tau l,inst,hfs)}(lh,j)$$

Figure 5.9: The instance relation. The interpretation $\mathscr{K}$ of continuations is given in the Coq formalization.

denoted $\mathscr{I}[\![C]\!]_{hfs}$, is the keystone to derive specifications for unknown functions. Notice that we are defining a relational interpretation over C, rather than m itself. This is so we can interpret the evaluation context. However, to distinguish between the static and dynamic parts of C, we define two relations over C: the aforementioned instance relation, and a context relation $\mathscr{C}tx[\![C]\!]_{(inst,hfs)} : Lholed \to iProp$. Both are parameterized by a host identifier list *hfs*, and the latter is additionally parameterized by an instance *inst*. The instance relation is populated by instances, while the context relation is populated by evaluation contexts. Again, we focus here on their most salient features, and leave their complete formal definitions to the Coq formalization.

Figure 5.9 defines the instance and context relations. The instance relation takes each field in the typing context, and applies a relation to the corresponding field in the instance. The table and memory fields respectively contain a list (types of, for the context) of tables and of memories. However, in Wasm 1.0, any table and memory operation implicitly refer only to the first element of those lists, as module instances are limited to at most one table and one memory. Likewise, the instance relation extracts the head of the table and memory lists (which can be empty).

We focus on the key definition, that of the function table relation and function relation, and refer to the Coq formalization for the other fields. In general, it suffices to know that they each are defined in terms of (non-atomic) invariants containing the appropriate resources, and some conditions on the state of those resources, according to their type.

The table relation $\mathscr{T}able[\![\tau t]\!](t)$ asserts ownership of a non-atomic invariant for each entry in the table at table address t, which either points to None, or to some function entry which is valid, that is, in the function relation. In Wasm, a table can

$$\boxed{\mathscr{C}los\llbracket ts \to ts' \rrbracket_{hfs} : Closure \to iProp}$$

$$\begin{aligned}
&\mathscr{C}los\llbracket ts \to ts' \rrbracket_{hfs}\big(\{(inst, tlocs); e\}^{\mathsf{NativeCl}}_{ts \to ts'}\big) \triangleq \\
&\quad \Box\, \forall vs, f, \mathscr{V}\llbracket ts \rrbracket(\mathbf{immV}\ vs) * [\mathrm{NaInv} : \top] * \xhookrightarrow{\textsc{Fr}} F \mathrel{-\!\!*} \\
&\qquad \mathscr{E}\llbracket ts' \rrbracket_{(F,hfs)}(\mathbf{local}_{\mathsf{len}(ts')}\{inst; vs \mathbin{+\!\!+} \mathbf{zeros}(tlocs)\}\ \mathbf{label}_{\mathsf{len}(ts')}\{\varepsilon\}\ e\ \mathbf{end}\ \mathbf{end})
\end{aligned}$$

$$\mathscr{C}los\llbracket ts \to ts' \rrbracket_{hfs}\big(\{h\}^{\mathsf{HostCl}}_{ts \to ts'}\big) \triangleq (h, ts \to ts') \in hfs$$

$$\boxed{\mathscr{F}unc\llbracket ts \to ts' \rrbracket_{hfs} : \mathbb{N} \to iProp}$$

$$\mathscr{F}unc\llbracket ts \to ts' \rrbracket_{hfs}(n) \quad \triangleq \quad \exists cl, \mathsf{NaInv}^{\mathcal{N}_{wf}.n}(n \xmapsto{\mathsf{wf}} cl) * \triangleright \mathscr{C}los\llbracket ts \to ts' \rrbracket_{hfs}(cl)$$

Figure 5.10: Function Relation

contain functions of different types, and the host can even replace a function in a table by another of a different type. Therefore, the onus is on indirect calls, which dynamically verify that the invoked function has the correct type (in `call-indirect`). Hence, a table type τt merely gives the size limits of a table, but does not dictate any function types for the table content itself. Accordingly, our table relation existentially quantifies over the function type for each table entry.

The function relation (Figure 5.10) asserts ownership of a non-atomic invariant containing the resource for function address n, invariantly pointing to a closure cl. This closure is either a native WebAssembly closure, or a host closure. A native closure is valid when for any parameter vs of the valid input type, and for any outer frame F, the expression relation holds for the closed local expression surrounding the body of cl. The new inner frame contains the environment of the closure; the instance $inst$, and a list of locals consisting of the parameters vs, followed by a list of type $tlocs$ initialised to 0. A host closure is valid when the host identifier and function type pair is an element of hfs.

Native function closures introduce circularity: an instance is valid insofar as each closure it contains is safe to execute, but when a module is instantiated, each function of that module is stored within the instance, as a closure around it. To break this circularity, we guard the closure relation with a later modality, $\triangleright$.

The context relation $\mathscr{C}tx$ asserts various structural conditions over the evaluation context, according to τ_{lbs}. In particular, it applies a continuation relation on each break target in τ_{lbs}. If τ_{lbs} is nil, the context is necessarily empty.

5.5.1.3 Fundamental Theorem of Logical Relations

Thus far, we have defined relational interpretations of all WebAssembly types. What remains, is to assemble the pieces and define a relational interpretation of the typing

$$
\begin{aligned}
C \vDash es : ts \to ts' \quad \triangleq \quad & \forall inst, lh, hfs, \mathscr{I}[\![C]\!]_{hfs}(inst) * \mathscr{C}tx[\![C]\!]_{(inst,hfs)}(lh) \twoheadrightarrow* \\
& \forall F, vs, \mathscr{V}[\![ts]\!](vs) * \mathscr{F}rame[\![\tau l]\!]_i(F) \twoheadrightarrow* \mathscr{E}[\![ts']\!]^{\tau_{\mathsf{lbs}},\tau_{\mathsf{ret}}}_{(\tau l, inst, hfs)}(lh, vs +\!\!+ es)
\end{aligned}
$$

$$
\text{where} \quad \tau l = C.\mathsf{locals} \quad \tau_{\mathsf{lbs}} = C.\mathsf{labels} \quad \tau_{\mathsf{ret}} = C.\mathsf{return} \quad n = \mathsf{len}(ts')
$$

Figure 5.11: Semantic typing

judgment, in other words *semantic typing*, and prove that this definition is sound via the *fundamental theorem of logical relations*.

In essence, the fundamental theorem states that syntactic typing implies semantic typing, the latter being defined in terms of the expression relation. Intuitively, it means that the resources accessible via the enclosed instance are sufficient for a well-typed function to execute without getting stuck. Figure 5.11 defines semantic typing of WebAssembly programs. $C \vDash es : ts \to \eta s$ states that, for any instance *inst* that satisfies the instance relation at type C, and for any evaluation context lh that is similarly valid, the expression relation holds for the expression $vs +\!\!+ es$, where vs is any value of type ts. The fundamental theorem is stated as follows:

Theorem 18 (FTLR). *If $C \vdash es : ts \to \eta s$, then $C \vDash es : ts \to \eta s$.*

Proof. The proof proceeds by induction over the typing judgment. We here go through four illustrative cases, and refer to the Coq mechanization for the remaining ones.

- Case: $C \vdash t.binop : [t;t] \to [t]$:

  Upon unfolding the definition of semantic typing, we introduce the following into our context (note that the first three hypotheses are persistent):

$$
\begin{aligned}
& \mathscr{I}[\![C]\!]_{hfs}(inst) && (5.1)\\
* \; & \mathscr{C}tx[\![C]\!]_{(inst,hfs)}(lh) && (5.2)\\
* \; & \mathscr{V}[\![t;t]\!](vs) && (5.3)\\
* \; & \mathscr{F}rame[\![\tau l]\!]_i(F) && (5.4)
\end{aligned}
$$

  Our goal is to show the following weakest precondition, where $\cdots$ stands for the remaining disjunctions in the postcondition for $\mathscr{E}$, as defined in Figure 5.8.

$$
\mathsf{wp}\, vs +\!\!+ [t.binop]\, \{w, \mathscr{V}[\![[t]]\!](w) \vee \cdots\}
$$

  From hypothesis 5.3, we know that vs is either a **trapV**, or a list of two values of type t: **immV** $[t.\mathbf{const}\; c_1; t.\mathbf{const}\; c_2]$.

  In the first case, we apply the Iris-Wasm proof rule for traps, which consumes the surrounding evaluation context until the final value is **trap**. In this case, we must show that the postcondition holds for **trapV**. This is easily established, since

$\mathscr{V}[\![\cdot]\!](\textbf{trapV})$ holds at all types. Henceforth, we will disregard this sub-case, as it proceeds identically in the remaining cases.

In the second case, we apply the Iris-Wasm proof rule for binary operations, namely `wp_binop` (Section 5.2). Let c be the result of the binary operation on c_1 and c_2. We must now show that the postcondition holds for the logical value **immV** $[t.\textbf{const}\ c]$, in other words:

$$\mathscr{V}[\![[t]]\!](\textbf{immV}\ [t.\textbf{const}\ c])$$

Which holds by definition of $\mathscr{V}$, since the result is indeed a constant of type t.

- Case: $C \vdash t.\textbf{load}\ \mathit{flags} : [\textbf{i32}] \to [t]$:

  As in the previous case, we introduce hypotheses into the context, and unfold the goal to reveal the following weakest precondition:

  $$\mathsf{wp}\ vs +\!\!+ [t.\textbf{load}\ \mathit{flags}]\ \{w, \mathscr{V}[\![[t]]\!](w) \vee \cdots\}$$

  Given the assumption $\mathscr{V}[\![[\textbf{i32}]]\!](vs)$, we know that vs is either a trap, or a single 32-bit integer v. The trap case proceeds as usual, and we are left with the case where we must reason about a memory load from address **i32**.**const** v. However, in order to apply the proof rule for **load**, we must first acquire the memory points-to predicates starting at v up to the number of bytes we are loading (as indicated by *flags*).

  From the typing of **load**, we know that $C.\mathsf{memory} \neq []$. Given the assumption $[\![C]\!]_{\mathit{hfs}}(\mathit{inst})$, we can thus infer an instance of $\mathscr{M}em[\![mt]\!](m)$ for some m that happens to be at the head of $C.\mathsf{memory}$. The memory relation unfolds to a non atomic iris invariant containing points to predicates for each byte of memory. Using the non atomic token $[\mathrm{NaInv} : \top]$ from the $\mathscr{F}rame[\![\tau l]\!]_i(F)$ hypothesis, we can then open the invariant, and extract the relevant points to predicates, so we can apply the proof rule for **load**. Once applied, we close the invariant, and show that the postcondition holds for the loaded value, which as in the previous case, follows from a successful load.

  What we have here omitted, are the cases in which the load dynamically *fails*, for instance if v is out of bounds. In those cases, we apply versions of the proof rule that assume a failing condition, and reason as usual about the resulting trap.

- Case: $C \vdash \textbf{br}\ i : ts_1 +\!\!+ ts \to ts_2$:
  Additional assumption: $C.\mathsf{labels}[i] = ts$

  We proceed as usual, and derive the following goal, in which we have asserted that the value stack equals $vs +\!\!+ vs_1$, for some vs of type ts, and for some vs_1 of type ts_1.

  $$\mathsf{wp}\ vs +\!\!+ vs_1 +\!\!+ [\textbf{br}\ i]\ \{w, \cdots \vee \mathscr{B}r[\![C.\mathsf{labels}]\!]^*_*(w, lh) \vee \cdots\}$$

  Note that we have preemptively highlighted the disjunct in the postcondition that will be of interest in this case, with the super- and sub- scripts left out for readability.

The expression in our goal is a stuck break expressions. In other words, we cannot yet reason about taking a step, and must first go to the postcondition. The relevant disjunct is the break relation, and we must now show the following:

$$\mathscr{B}r[\![C.\mathsf{labels}]\!]^*_*(vs +\!\!+ vs_1 +\!\!+ [\mathbf{br}\ i], lh)$$

Unfolding the break relation reveals a new weakest precondition, in which the expression is the labeled evaluation context consisting of the innermost $i+1$ layers of lh, where the hole has been filled by $vs +\!\!+ vs_1 +\!\!+ [\mathbf{br}\ i]$, with a postcondition that resembles that of the expression relation, expect it now allows a final value of any type, and the break relation is now instantiated to a new context lh', representing the remaining layers of lh.

Note that we rely on the condition, that lh is constructed of at least $i+1$ layers. This follows from the definition of $\mathscr{B}r[\![C.\mathsf{labels}]\!](\cdot, lh)$, which asserts that lh has at least depth $\mathsf{len}(C.\mathsf{labels})$, and from the assumption that $C.\mathsf{labels}$ has length at least $i+1$.

Next, we reason as usual by applying the proof rule for **br** i. The result is a value, namely the value stack vs, and we reason as usual about the new postcondition.

- Case: $C \vdash \mathbf{block}\ (ts_1 \to ts_2)\ es : ts_1 \to ts_2$: Induction Hypothesis: $C[\mathsf{labels} = ts_2 :: C.\mathsf{labels}] \vDash es : ts_1 \to as_2$

  For the inductive cases, the proof proceeds by taking a step, and applying the induction hypothesis to reason about the remaining execution.

  The main difficulty lies in the two different typing contexts. Indeed, the **block** instruction generates a new label, which means all existing break targets shift, and the innermost label presents a new valid break target. Concretely, it means we must reestablish the context relation on the new list of labels.

  Upon introducing the expected assumptions into the context, we derive the following goal:

  $$\mathsf{wp}\, vs +\!\!+ \mathbf{block}\ (ts_1 \to ts_2)\ es\, \{w, \cdots\}$$

  We begin by applying the proof rule for **block**:

  $$\mathsf{wp}\, lh_1[es]\, \{w, \cdots\}$$

  where $lh_1 = vs +\!\!+ \mathbf{label}_{\mathsf{len}(ts_2)}\{[]\}\ [\_]\ \mathbf{end}$. The induction hypothesis provides a weakest precondition statement for es. We thus need to bind into the evaluation context lh_1, by applying `wp_ctx_bind`:

  $$\mathsf{wp}\, lh_1[es]\, \{w, \mathsf{wp}\, lh_1[w]\, \{w', \cdots\}\} \tag{5.5}$$

  Next, we must derive a weakest precondition statement from the induction hypothesis. This involves instantiating the semantic typing judgment with an instance, an evaluation context, a list of host functions, and an input stack, along with the relevant iris resources, most notably an assertion of the instance and context relations.

Only the labels field of the context C has changed. Thus, the main proof effort lies in establishing the context relation on the new typing context, instantiated to an evaluation context with a new innermost label. Let $\mathsf{push}(lh, lh')$ denote an evaluation context where we replace the base layer of lh with lh'. The main proof obligation is to show the following:

$$\mathscr{C}tx[\![C]\!]_{(inst,hfs)}(lh) \mathrel{-\!\!*} \mathscr{C}tx[\![C[\mathsf{labels} = ts_2 :: C.\mathsf{labels}]]\!]_{(inst,hfs)}(\mathsf{push}(lh, lh_1))$$

The structural conditions follow by construction of the new evaluation context. What remains is to derive the continuation relation $\mathscr{K}$ for each element of $C.\mathsf{labels}$, now shifted on index up, and to prove $\mathscr{K}$ for the new break target in lh_1.

Finally, once a weakest precondition statement for es has been established, the final step is to show that its postcondition implies the postcondition of 5.5, by considering each possible final value. In particular, it involves shifting the break target in case of a stuck break value.

□

The fundamental theorem allows us to derive specifications for each declared function in an arbitrary well typed module. However, these specifications can only be applied, and are thus only useful, once we establish that the produced instance is valid. The following key theorem states that the result of instantiating a well-typed module $\vdash m : timps \to texps$ produces a valid instance, given that all imports are valid according to $timps$.

Theorem 19 (Valid Instance Allocation)**.** *If $\vdash m : timps \to texps$, and inst is the result of instantiating module m with imports imps, then*

$$\mathsf{resources}(m, imps, timps, \cdots, inst) \mathrel{-\!\!*} \mathsf{valid}[\![timps]\!](imps) \mathrel{-\!\!*} \mathscr{I}[\![C]\!](inst)$$

where C is the module type, determined syntactically, $\mathsf{resources}(\cdots, inst)$ *corresponds to the ghost resources allocated by module instantiation as depicted by Lemma 27, and* $\mathsf{valid}[\![timps]\!](imps)$ *unfolds the list of imports, and applies the relevant relation on each import object.*

Proof. By unfolding the definition of module typing, inferring properties about the result of instantiating m, and component-wise proving the instance relation. Validity of imported types is established by the $\mathsf{valid}[\![timps]\!](imps)$ assumption, while the rest are established using typing information, and the definition of the logical relation. In particular, the fundamental theorem of logical relations (Theorem 18) is applied to prove the validity of each module function. □

5.5.2 Illustrative Example

To illustrate the application of the logical relation, we present a synthetic example that depends on the encapsulation of locally declared memory, against an unknown

```
(module ;; LSE of memory
  (import "adv" "f" (func $f))
  (memory 1)
  (global $ret (import "host" "ret")
      i32)
  (func $main
    const 0; const 42; store; const 0;
        call $f;
    load i32; global.set $ret))
```

```
inst_decl [] "adv" ["f"]
inst_decl ["f";"ret"] "lse" []
get_global $ret
```

Figure 5.12: Robust safety example: testing the local state encapsulation of memory. We refer to imports and modules via names rather than indices, for the sake of readability.

but well-typed function. Figure 5.12 depicts a WebAssembly module "lse" (for 'local state encapsulation') that imports a function "f" from an arbitrary module called "adv" (for 'adversary'), declares a memory, and imports a global variable "ret" from the host to store the return value of its $main function.

The $main function stores 42 to memory address 0, calls the imported function $f, after which it loads from memory address 0 and stores the result into $ret. Since the memory is encapsulated in the module, the integrity of the memory is preserved during the call to $f. We should therefore be able to prove that $ret contains 42 upon return. In short, our goal is to prove the following specification, where main refers to the body of $main, and we use $f and $ret to refer to the indices of their respective names:

$$
\begin{array}{l}
\mathsf{NaInv}^{\mathcal{N}_{wf}.\$\mathsf{f}}(\$\mathsf{f} \xmapsto{\mathsf{wf}} \{(\mathit{inst}, \mathit{tlocs}); \mathit{es}\}^{\mathsf{NativeCl}}_{[]\to[]}) \vdash \\
\left\{\begin{array}{l}
\$\mathsf{ret} \xmapsto{\mathsf{wg}} \{\mathbf{mut}; -\} * \\
\xhookrightarrow{\textsc{Fr}} F * \\
F.\mathsf{inst.mems} \xmapsto{\mathsf{wm}}_0 -
\end{array}\right\} \mathsf{main} \left\{ w, \begin{array}{l}
w = \mathbf{trap} \lor \\
\left(\begin{array}{l}
w = \mathbf{immV}\ [] * \\
\$\mathsf{ret} \xmapsto{\mathsf{wg}} \{\mathbf{mut}; 42\} * \\
\xhookrightarrow{\textsc{Fr}} F * \ F.\mathsf{inst.mems} \xmapsto{\mathsf{wm}}_0 42
\end{array}\right)
\end{array}\right\}
\end{array}
$$

We will refer to this specification as $\mathsf{mainSpec}(\mathit{es}, \mathit{tlocs}, \mathit{inst})$. Note that the resource for $f is in a non-atomic invariant, such that it matches the expected definition of function validity.

Figure 5.12 depicts the host code that instantiates and links the two modules. Recall that the host type checks any module during instantiation, which includes type checking imports and exports. In this scenario, we expect the unknown module $adv to require no imports, and export a single function of type [] → []. This function is then passed as the import object to the instantiation of the $lse module, together with the ambient global variable $ret.

Thus, in the above specification, *es* is the body of a native closure of type [] → []. In order to *prove* the above triple, we need a specification to reason about the call to $f inside of main. More precisely, we need a specification that does not break the integrity of *F*.inst.mems. Since *es* is arbitrary and possibly adversarial, we cannot assume the existence of such a specification. Instead, we will rely on the fact that the imported closure has been type checked. Because of that, it will (by the fundamental theorem) satisfy the semantic invariants of the module system, captured by the logical relation, and those suffice to carry out our proof.

However, in order to get a weakest precondition statement out of the fundamental theorem, we need to assert that the enclosed instance is valid. We therefore split the proof into two stages. First, we assume that the adversary instance is valid, and prove the specification for the WebAssembly code. Next, we show that the host code instantiating the adversarial and trusted module produces a valid adversary instance.

The following theorem captures, that, as long as the unknown function is syntactically well-typed (which is ensured by instantiation), it cannot break the local state encapsulation of our module.

Theorem 20 (Robust Safety of the motivating example)**.** *If* $C.\mathsf{labels} = [] \land C.\mathsf{return} = \mathsf{None}$ *and* $C[\mathsf{locals} \leftarrow tlocs][\mathsf{labels} \leftarrow [[]]][\mathsf{return} \leftarrow \mathsf{Some}([])] \vdash es : [] \to []$*, then*

$$\mathscr{I}[\![C]\!]_{[]}(inst) \mathrel{-\!\!*} [NaInv : \top] \mathrel{-\!\!*} \mathsf{mainSpec}(es, tlocs, inst)$$

Proof. The proof largely consists of applying weakest precondition rules for each known instruction. For adversary invocation, we open the invariant to apply Theorem 18 to get a specification for the adversary code. Since the specification depends only the resources in $\mathscr{I}[\![C]\!]_{[]}(inst)$ (the context relation $\mathscr{C}tx$ is manually established for the known evaluation context of one layer), we keep the remaining resources separate, and we can easily step through the remainder of the program. □

Next, we need to show that module instantiation indeed lets us fulfill the assumptions and preconditions of Theorem 20. We assume relatively little about the adversary module, only that it is well-typed against an empty import list and a single function export, that it does not have a start function, and that instantiating it will succeed (see restrictions on instantiation described in Section 5.3). The host owns a preallocated global variable for the return value of the main function, which it will read from post instantiation. It first instantiates the adversary module, given the assumptions above. It then uses the exported adversary function as the import for the instantiation of the trusted module, jumping to its main function. The specification uses sugared syntax, and corresponds to a triple in the host program logic.

Theorem 21 (Full Host and WebAssembly Specification)**.** *If* $\vdash m_{\mathsf{adv}} : []\ [\mathbf{func_e}\ ([] \to [])]$ *and the syntactic restrictions on* m_{adv} *hold, then*

$$\left\{\begin{array}{l} \text{"adv"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{adv}} * \\ \text{"lse"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{lse}} * \\ \text{"ret"} \xmapsto{\mathsf{vis}} \$\mathsf{ret} * \\ \text{"f"} \xmapsto{\mathsf{vis}} - * \\ \$\mathsf{ret} \xmapsto{\mathsf{wg}} - * \\ [NaInv : \top] \end{array}\right\} \begin{array}{l} \mathbf{inst\_decl}\ []\ \text{"adv"}\ [\text{"f"}] \\ \mathbf{inst\_decl}\ [\text{"f"}; \text{"ret"}]\ \text{"lse"}\ [] \\ \mathbf{get\_global}\ \$\mathsf{ret} \end{array} \left\{ hw, \begin{array}{l} hw = (\mathbf{trap}, []) \lor \\ hw = ([\mathbf{i32.const}\ 42], []) \end{array} \right\}$$

Proof. Given the typing assumption on the adversary module m_{adv}, its instantiation successfully allocates an instance *inst*, with all the resources that go with it. In particular, given the export type of the module, we know that instantiation allocates a function pointer for some well-typed function of type $[] \to []$. This resource is then used to instantiate the trusted module m_{lse}. Once each module has been instantiated, we establish the validity of *inst* by Theorem 19. The remaining assumptions are derived from the typing of m_{adv}, and we conclude by applying Theorem 20. □

5.5.3 Applications of the Logical Relation on the Stack Module

Next we describe two scenarios, each involving our stack module interacting with some unknown function. In each case, the two modules interact via imported closures. We will therefore employ the closure relation $\mathscr{C}los$ as the principal logical relation in our reasoning.

The two applications highlight a conceptual distinction between two kinds of scenarios in which known code interacts with unknown code. In the first example, known code imports functions from an unknown module, and has a certain amount of control over how these are applied. The second example exports known code to an unknown module, and in that case, exported closures must carefully guard against misuse.

Figure 5.7 depicts a client of the stack module, which imports a closure "f" of type $[\mathbf{i32}] \to [\mathbf{i32}]$ from an unknown module. The client creates a new stack, pushes two values, then applies map using the imported unknown function, and finally computes the length of the stack by calling a function from the stack module. The stack module hides its internal representation from the context. Likewise, the host makes sure to hide the stack module operations from the unknown module. WebAssembly's coarse grained encapsulation thus guarantees that the integrity of the allocated stack is maintained, no matter what the unknown imported function does: as long as it does not trap, the final length operation succeeds and returns the original size of the stack, namely 2. We refer to imports and modules via names rather than indices, for the sake of readability. The following theorem expresses robust safety formally:

Theorem 22 (Top-level Host Specification). *If* $\vdash m_{\mathsf{adv}} : [] \; [\mathbf{func_e} \; ([\mathbf{i32}] \to [\mathbf{i32}])]$ *and the syntactic restrictions on* m_{adv} *hold, then*

$$\left\{ \begin{array}{l} \text{"stack"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{stack}} * \\ \text{"adv"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{adv}} * \\ \text{"client"} \xhookrightarrow{\mathsf{mod}} m_{\mathsf{client}} * \\ \text{"g"} \xmapsto{\mathsf{vis}} \$\mathrm{g} * \\ \$\mathrm{g} \xmapsto{\mathsf{wg}} - * \; [\mathit{NaInv} : \top] * \\ \quad \text{"f"}, \text{"tab1"}, \text{"map"}, \\ \qquad \ldots, \text{"pop"} \xmapsto{\mathsf{vis}} - \end{array} \right\} \mathsf{stack\_client} \left\{ hw, \begin{array}{l} (hw = ([]; []) \wedge \$\mathrm{g} \xmapsto{\mathsf{wg}} 2) \vee \\ hw = (\mathbf{trap}; []) \end{array} \right\}$$

Proof. Once the host has allocated the unknown module, we apply Theorem 19 to conclude that its instance is valid, which guarantees that each of its components, including the exported closure of type $[\mathbf{i32}] \rightarrow [\mathbf{i32}]$, is valid. As a result, we know that the unknown import of our client is in the closure relation $\mathscr{C}los$, which by definition of the relational interpretation includes a specification for the unknown function. Crucially, this specification does not depend on the stack internals, and thus we are able to prove that the stack size is maintained. □

Next we consider a scenario in which an unknown module imports operations from the stack module, namely new_stack, push and pop. The encapsulation of the stack module's internal state, alongside careful checks at the boundaries of each operation, which we will elaborate on below, should guarantee that the stack module memory indeed stores and maintains stacks, as defined by the **isStack** predicate, irrespectively of what the unknown module does. Henceforth we will refer to this as the representation invariant, denoted by $\mathsf{stackInvariant}(m)$, where m is the index of the encapsulated memory. Roughly, the representation invariant is an Iris (non-atomic) invariant containing a big separation of **isStack** predicates, one for each allocated stack.

The basic type system of WebAssembly guarantees that the adversary code does not get stuck. However, our goal is to reason about integrity of the data representation enforced by the module system. While the type system defines the typing of an individual module, it does not consider interweaving of module instantiations, since instantiation is handled by a host, typically written in untyped JavaScript. Therefore, the type system is too weak to capture the data abstraction enforced by the module system, which we are relying on here. As such, our interpretation of the type system does not capture the refined interpretation (with the representation invariant) of the stack module.

We use the standard type interpretation of the adversary module to reason about its execution. However, we want this interpretation to depend on the *refined representation invariant* of the stack module internals, rather than the *default interpretation granted by the logical relation*. Since each import must be valid when applying Theorem 19, we *manually* prove that, given the representation invariant, each exported function (new_stack, push, and pop) is in the closure relation.

As a result, we must now consider the case where a stack operation is applied on an arbitrary input value. Consider, for instance, push – it takes two arguments, one of which is a stack value, which is interpreted as a memory address. A malicious adversary could apply push to a masked stack value (a bogus memory address), thus breaking the expected internal behavior of the stack module. push must thus guard against such a situation by dynamically checking the validity of all safety-critical parameters. These dynamic checks ensure that no stack gets corrupted. Relying on those dynamic checks, we can then prove specifications that maintain the representation invariant:

Theorem 23 (Validity of Select Stack Module Operations)**.** *If $inst.\mathsf{mems} = [m]$ then,*

$$\begin{aligned}\mathsf{stackInvariant}(m) \twoheadrightarrow\ & \mathscr{C}los\llbracket[\mathbf{i32};\mathbf{i32}] \to []\rrbracket(\{(inst,[\mathbf{i32}]);\mathsf{push}\}^{\mathsf{NativeCl}})\\ &\ast \mathscr{C}los\llbracket[\mathbf{i32}] \to [\mathbf{i32}]\rrbracket(\{(inst,[\mathbf{i32}]);\mathsf{pop}\}^{\mathsf{NativeCl}})\\ &\ast \mathscr{C}los\llbracket[] \to [\mathbf{i32}]\rrbracket(\{(inst,[\mathbf{i32}]);\mathsf{new\_stack}\}^{\mathsf{NativeCl}})\end{aligned}$$

The representation invariant is allocated upon instantiation of the stack module, at which point there are no allocated stacks. Theorem 23 is then applied on each of the relevant stack module exports, such that we can apply Theorem 19, and conclude with the standard type interpretation of the adversary module, while maintaining the now allocated representation invariant.

5.6 Related work

Watt et al. [159] develop a mechanised first-order separation logic for what they call "encapsulated" WebAssembly, that is, code limited to a single module, with no exports or imports, and no uses of the **call_indirect** instruction or the host, and they do not handle instantiation. For their subset of the language, our proof rules are similar up to presentational details, except for the handling of breaks, where, as mentioned in §5.2.2, we use a novel approach with a bind rule which scales to higher-order programs, unlike the approach taken by Watt et al. [159].

WebAssembly provides coarse-grained memory safety, at the boundary of memory objects, and coarse-grained isolation, at the boundary of modules. Lehmann et al. [88] show that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against Wasm programs that not specifically written to take advantage of module isolation. We show in our examples that, when Wasm programs are written with module isolation in mind, the language specification does indeed enforce expected isolation guarantees.

MSWasm [38, 95] (Memory-Safe Wasm) is a proposed extension of WebAssembly that adds first-class support for CHERI-like [154] fine-grained runtime-checked memory capabilities. The logical relation of Cerise [55, 58–60], mechanised in Iris, captures encapsulation for hardware capabilities in an idealised assembly model and may be used as a starting point to formalise the guarantees of MSWasm on top of Iris-Wasm.

CapableWasm [53] is a (work-in-progress) extension of the type system of WebAssembly to support compositional compilation from different languages. They rely on their type system to enforce finer-grained encapsulation than at the module boundary.

Kolosick et al. [81] use a logical relation to show that WebAssembly programs naturally compile to unsafe platform assembly in such a way that the compiled code obeys a safe calling convention and certain isolation properties with respect to the rest of the system. Narayan et al. [105] rely on this result to implement a sandboxing technique whereby C code is first compiled to WebAssembly which is then ultimately compiled to native assembly for linking. They use this technique to sandbox a number of Firefox libraries.

The λ_{Rust} calculus of the RustBelt project [76] explores features of a modern industrial language in Iris. Their aim is to investigate Rust's novel ownership-based type system, using a core calculus that captures the essence of Rust, and define a logical relation which characterises a discipline for composing safe code which obeys the Rust type system with untyped unsafe code.

In Iris-Wasm, we exploit the higher-order features of Iris. In particular, we use that weakest preconditions can be nested inside other weakest preconditions to specify higher-order functions and WebAssembly control structures, and quantification over predicates to get abstract modular specifications of modules. These features of Iris stem from earlier work on higher-order separation logic, starting with Biering et al. [19]. To show robust safety, we further rely on Iris' invariants, the idea of which can be traced back at least to recursively-defined Kripke models of type systems [7] and separation logics with hidden state [125].

Many related works deal with the mechanized formalization of low-level languages. RockSalt [101] is a verified checker that validates code binaries against a sandbox policy, similar to that of Google's Native Client (NaCl). RockSalt is mechanically verified using a formalization of a subset of x86 in Coq. Kennedy et al. [79] use Coq to build a macro assembler for x86, while relating machine code to separation-logic formulas suitable for program verification.

The Certified Assembly Programming (CAP) family of frameworks [51, 107, 164, 165] support the definition of second-order Hoare logics for verifying modular specifications of low-level assembly programs, using expressive features such as embedded code pointers, concurrency, and dynamic thread creation. As such, CAP focuses on features that are abstracted away by Wasm. Built on top of these frameworks, Gu et al. [63] presents CertiKOS, an extensible architecture for certifying concurrent OS kernels. Using CertiKOS, Gu et al. [63, 64] develop and verify a concurrent OS kernel consisting of both C and x86 assembly code. By leveraging CompCertX [62], CertiKOS is able to reason about interactions between C and x86 assembly. As is the case with Iris-Wasm, the setup assumes that the two languages share the same memory model. The recent DimSum [123] framework supports reasoning about multilingual programs between languages with different memory models. However, while Iris-Wasm focuses on mechanizing the full language of a real industrial standard, the DimSum approach has so far only been applied to a simple high-level imperative language and an idealized assembly language.

The W3C have announced a Public Working Draft for WebAssembly 2.0. It includes several features orthogonal to our focus on security, such as extra numeric operations. The two relevant features are: the lifting of the artificial restriction to one table per module (we have done this too), which corresponds to a simple update to the relation on instances; and the addition of opaque reference types to objects of the host language, which adds new WebAssembly values, but no actual complexity because of their opacity (this is trivial to do).

5.7 Conclusion

We have presented Iris-Wasm, a practical higher-order, mechanised program logic for the W3C WebAssembly 1.0 official language standard [119], building on the mechanized WasmCert-Coq specification [160]. We show how the reasoning of Iris-Wasm can handle the intricacies of WebAssembly, including interaction with its host language and the higher-order programs and reentrancy that it enables, going far beyond the 'encapsulated' fragment of WebAssembly in previous work [159]. We then leverage our program logic to build a logical relation which enforces robust safety, demonstrating that we can prove properties of encapsulation at module boundaries. This example illustrates the potential of what can be done with formal methods. We hope other researchers will use our formalisation to further investigate the WebAssembly ecosystem, and that industrial language communities will thereby be further enticed to embrace the formalisation of language specifications.

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999. URL https://doi.org/10.1007/3-540-48749-2_2. 196

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009. doi: 10.1145/1609956.1609960. URL https://doi.org/10.1145/1609956.1609960. 2

[3] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1351–1368, 2018. doi: 10.1145/3243734.3243745. URL https://doi.org/10.1145/3243734.3243745. 22, 141, 197

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 256–271. IEEE, 2019. doi: 10.1109/CSF.2019.00025. URL https://doi.org/10.1109/CSF.2019.00025. 22

[5] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 334–344. IEEE Computer Society, 1998. doi: 10.1109/LICS.1998.705669. URL https://doi.org/10.1109/LICS.1998.705669. 97

[6] Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. 120, 197

[7] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009. doi: 10.1145/1480881.1480925. URL https://doi.org/10.1145/1480881.1480925. 241

[8] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 340–353. ACM, 2009. doi: 10.1145/1480881.1480925. URL https://doi.org/10.1145/1480881.1480925. 123, 161

[9] Akram El-Korashy. *A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI*. Master thesis, Saarland University, September 2016. 150

[10] Sean Noble Anderson, Leonidas Lampropoulos, Roberto Blanco, Benjamin C. Pierce, and Andrew Tolmach. Security properties for stack safety. *CoRR*, abs/2105.00417, 2021. URL https://arxiv.org/abs/2105.00417. 154, 155, 156, 158, 160, 161, 196, 197

[11] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7. 70

[12] Arm. Morello project, 2021. URL https://www.morello-project.org/. 7, 153

[13] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Jon French, Kathryn E. Gray, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. The Sail instruction-set architecture (isa) specification language, 2013–2019. 17, 103

[14] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019. doi: 10.1145/3290384. URL https://doi.org/10.1145/3290384. 11, 17, 96

[15] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21,*

2014, pages 165–178, 2014. doi: 10.1145/2535838.2535839. URL https://doi.org/10.1145/2535838.2535839. 197

[16] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 813–830, 2015. doi: 10.1109/SP.2015.55. URL https://doi.org/10.1109/SP.2015.55. 197

[17] Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. The meaning of memory safety. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804 of *Lecture Notes in Computer Science*, pages 79–105. Springer, 2018. doi: 10.1007/978-3-319-89722-6_4. URL https://doi.org/10.1007/978-3-319-89722-6_4. 197

[18] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. In *Proceedings of the 31st European Symposium on Programming*, April 2022. 11, 196

[19] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007. doi: 10.1145/1275497.1275499. URL https://doi.org/10.1145/1275497.1275499. 241

[20] Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. Technical report, Aarhus University, 2017. 117, 217

[21] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 119–132. ACM, 2011. doi: 10.1145/1926385.1926401. URL https://doi.org/10.1145/1926385.1926401. 120, 197

[22] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and*

Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011, pages 30–40. ACM, 2011. doi: 10.1145/1966913.1966919. URL https://doi.org/10.1145/1966913.1966919. 3

[23] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. A Multipurpose Formal RISC-V Specification. *arXiv:2104.00762 [cs]*, April 2021. URL http://arxiv.org/abs/2104.00762. 96

[24] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 66–77, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250743. URL https://doi.org/10.1145/1250734.1250743. 151

[25] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 161–176. USENIX Association, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini. 3

[26] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 319–327. ACM Press, 1994. doi: 10.1145/195473.195579. URL https://doi.org/10.1145/195473.195579. 7

[27] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994. doi: 10.1145/195473.195579. URL http://doi.acm.org/10.1145/195473.195579. 3, 26, 30, 35, 104, 105, 149, 153, 161

[28] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-control-data attacks are realistic threats. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005. URL https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats. 3

[29] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. CHERI JNI: Sinking the Java Security Model into the C. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 569–583. ACM, 2017. doi: 10.1145/3037697.3037725. 26, 95, 149

[30] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 234–245. ACM, 2011. doi: 10.1145/1993498.1993526. URL https://doi.org/10.1145/1993498.1993526. 15

[31] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 234–245. ACM, 2011. doi: 10.1145/1993498.1993526. URL https://doi.org/10.1145/1993498.1993526. 151

[32] Chromium. Memory safety, 2020. URL https://www.chromium.org/Home/chromium-security/memory-safety. 2, 153

[33] DC Cosserat. A capability oriented multi-processor system for real-time applications. In *ICC Conf., Washington, DC*, 1972. 7

[34] JM Cotton. The operational requirements for future communications control processors. In *Internat. Switching Symp., Cambridge, Mass*, 1972. 7

[35] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu. A complete formal semantics of x86-64 user-level instruction set architecture. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1133–1148. ACM, 2019. doi: 10.1145/3314221.3314601. URL https://doi.org/10.1145/3314221.3314601. 17

[36] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, March 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. 3, 4, 5, 6, 26, 101, 153

[37] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 147–162. IEEE, 2016. doi: 10.1109/

EuroSP.2016.22. URL https://doi.org/10.1109/EuroSP.2016.22. 11, 14, 97, 150, 196

[38] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372268. doi: 10.1145/3337167.3337171. URL https://doi.org/10.1145/3337167.3337171. 240

[39] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 143–156. ACM, 2010. doi: 10.1145/1863543.1863566. URL https://doi.org/10.1145/1863543.1863566. 14, 97, 103, 123, 144, 145, 150, 157, 161, 197

[40] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. A relational modal logic for higher-order stateful adts. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 185–198, 2010. doi: 10.1145/1706299.1706323. URL https://doi.org/10.1145/1706299.1706323. 197

[41] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *LMCS*, 7(2:16):1–37, June 2011. 197

[42] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. Holistic specifications for robust programs. In Heike Wehrheim and Jordi Cabot, editors, *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 420–440. Springer, 2020. doi: 10.1007/978-3-030-45234-6_21. URL https://doi.org/10.1007/978-3-030-45234-6_21. 150

[43] Daniel Ehrenberg. WebAssembly JavaScript interface W3C recommendation. Technical report, W3C, December 2019. URL https://www.w3.org/TR/wasm-js-api-1/. 200, 201

[44] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Capableptrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 421–436, Los

Alamitos, CA, USA, jun 2021. IEEE Computer Society. doi: 10.1109/CSF51468.2021.00036. URL https://doi.ieeecomputersociety.org/10.1109/CSF51468.2021.00036. 22, 26, 95, 196

[45] Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, and Catalin Hritcu. Secureptrs: Proving secure compilation with dataflow back-translation and turn-taking simulation. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*, pages 64–79. IEEE, 2022. doi: 10.1109/CSF54842.2022.9919680. URL https://doi.org/10.1109/CSF54842.2022.9919680. 22

[46] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In Michael Engel and Olaf Spinczyk, editors, *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, IIES '08, Glasgow, Scotland, April 1, 2008*, pages 35–40. ACM, 2008. doi: 10.1145/1435458.1435465. URL https://doi.org/10.1145/1435458.1435465. 7

[47] R Fabry. Preliminary description of a supervisor for a machine oriented around capabilities. *Inst. Comput. Res. Quart. Rep*, 18, 1968. 7

[48] Robert S Fabry. A user's view of capabilities. *ICR Quarterly Report*, 15:1–8, 1967.

[49] Robert S Fabry. List-structured addressing. Technical report, Univ. of Chicago, IL (United States), 1971. 7

[50] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (PSOS). In *1979 International Workshop on Managing Requirements Knowledge, MARK 1979, New York, NY, USA, June 4-7, 1979*, pages 329–334. IEEE, 1979. doi: 10.1109/MARK.1979.8817256. URL https://doi.org/10.1109/MARK.1979.8817256. 6

[51] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 254–267. ACM, 2005. doi: 10.1145/1086365.1086399. URL https://doi.org/10.1145/1086365.1086399. 15, 241

[52] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson.

Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020. 67, 149, 195

[53] Michael Fitzgibbons. CapableWasm: Bringing better interop down to WebAssembly, January 2022. URL https://www.youtube.com/watch?v=E44lTaa2qHk. POPL'22 student research competition presentation. 240

[54] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 442–451. ACM, 2018. doi: 10.1145/3209108.3209174. URL https://doi.org/10.1145/3209108.3209174. 86, 183

[55] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021. URL https://doi.org/10.1145/3434287. 13, 21, 25, 29, 71, 95, 96, 97, 154, 157, 160, 161, 165, 167, 169, 170, 171, 174, 183, 223, 225, 240

[56] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Artifact for le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities, 2022. URL https://doi.org/10.5281/zenodo.5821862. 155

[57] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–30, 2022. doi: 10.1145/3527318. URL https://doi.org/10.1145/3527318. 21

[58] Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu, Alix Devriese, and Lars Birkedal. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021*, April 2021. URL https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf. 240

[59] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. Technical report, Aarhus University, 2022. URL https://cs.au.dk/~birke/papers/cerise.pdf. 21

[60] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities. Technical report, Aarhus University, 2022. URL https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf. 240

[61] Paolo Giarrusso, Leo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step — soudness for DOT with step-indexed logical relations in iris. *Proc. ACM Program. Lang.*, (ICFP), 2020. 150

[62] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608. ACM, 2015. doi: 10.1145/2676726.2676975. URL https://doi.org/10.1145/2676726.2676975. 16, 241

[63] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu. 16, 241

[64] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 646–661. ACM, 2018. doi: 10.1145/3192366.3192381. URL https://doi.org/10.1145/3192366.3192381. 241

[65] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363. 3

[66] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988. doi: 10.1145/54289.871709. URL https://doi.org/10.1145/54289.871709. 6

[67] Pat Hickey. How Fastly and the developer community are investing in the WebAssembly ecosystem, May 2020. URL https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem. 202

[68] Sander Huyghebaert. A secure calling convention with uninitialized capabilities. Master's thesis, Vrije Universiteit Brussel, 2020. URL https://doi.org/10.5281/zenodo.4073111. 104

[69] Sander Huyghebaert, Steven Keuchel, and Dominique Devriese. Semi-automatic verification of isa security guarantees in the form of universal contracts. In *Workshop on the Security of Software/Hardware Interfaces (SILM)*, 2021. 23

[70] Koen Jacobs, Dominique Devriese, and Amin Timany. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022. doi: 10.1145/3527326. URL https://doi.org/10.1145/3527326. 201

[71] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 301–314, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318327. doi: 10.1145/2429069.2429105. URL https://doi.org/10.1145/2429069.2429105. 151

[72] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. Efficient Tagged Memory. In *IEEE International Conference on Computer Design (ICCD)*. IEEE, November 2017. doi: 10.1109/ICCD.2017.112. 14, 102

[73] Nicolas Joly, Saif ElSherei, and Saar Amar. Security analysis of cheri isa, 2020. URL https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/. 153

[74] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650, 2015. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980. 20, 103, 155, 200

[75] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269, 2016. doi: 10.1145/2951913.2951943. URL https://doi.org/10.1145/2951913.2951943. 20, 103, 123, 155

[76] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154. 3, 99, 225, 241

[77] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi: 10.1017/S0956796818000151. 11, 28, 43, 103, 117, 123, 155, 171, 200, 207, 222, 223

[78] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust. *Communications of the ACM*, 64(4):144–152, 2021. 3

[79] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24. ACM, 2013. doi: 10.1145/2505879.2505897. URL https://doi.org/10.1145/2505879.2505897. 17, 241

[80] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596. 7

[81] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael Lemay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2022. doi: 10.1145/3498688. URL https://doi.org/10.1145/3498688. 197, 240

[82] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 696–723, 2017. doi: 10.1007/978-3-662-54434-1_26. URL https://doi.org/10.1007/978-3-662-54434-1_26. 20, 103, 155

[83] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 205–217. ACM, 2017. doi: 10.1145/3009837.3009855. URL https://doi.org/10.1145/3009837.3009855. 20, 86, 103, 120, 155, 183, 197, 225

[84] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–77:30, 2018. doi: 10.1145/3236772. URL https://doi.org/10.1145/3236772. 103, 155

[85] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. A relational model of types-and-effects in higher-order concurrent separation logic. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 218–231. ACM, 2017. doi: 10.1145/3009837.3009877. URL https://doi.org/10.1145/3009837.3009877. 86, 183

[86] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. doi: 10.1145/2535838.2535841. URL https://doi.org/10.1145/2535838.2535841. 16

[87] Butler W. Lampson. Dynamic protection structures. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '69 Fall Joint Computer Conference, November 18-20, 1969, Las Vegas, Navada, USA*, volume 35 of *AFIPS Conference Proceedings*, pages 27–38. AFIPS / ACM, 1969. doi: 10.1145/1478559.1478563. URL https://doi.org/10.1145/1478559.1478563. 5, 6, 8, 12, 18

[88] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 217–234. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann. 240

[89] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4): 363–446, 2009. doi: 10.1007/s10817-009-9155-4. URL https://doi.org/10.1007/s10817-009-9155-4. 16, 194, 195

[90] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016. 16

[91] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. ISBN 978-1-4831-0106-4. URL `https://homes.cs.washington.edu/~levy/capabook/`. 3, 26, 101, 149, 153

[92] Barbara H. Liskov, Alan Snyder, Russell R. Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977. doi: 10.1145/359763.359789. URL `https://doi.org/10.1145/359763.359789`. 86

[93] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140. IEEE Computer Society, 2010. doi: 10.1109/SP.2010.16. URL `https://doi.org/10.1109/SP.2010.16`. 11, 149

[94] Adrian Mettler, David A. Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010. URL `https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java`. 6

[95] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, 7(POPL):425–454, 2023. doi: 10.1145/3571208. URL `https://doi.org/10.1145/3571208`. 20, 22, 201, 240

[96] Microsoft. Data execution prevention, 2006. Retrieved February 2023 from `https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`. 2

[97] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006. 6, 101

[98] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript. *Google, Inc., Tech. Rep*, 2008. 6

[99] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. 11

[100] James H. Morris Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973. doi: 10.1145/361932.361937. URL https://doi.org/10.1145/361932.361937. 6, 86

[101] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger SFI for the x86. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 395–404. ACM, 2012. doi: 10.1145/2254064.2254111. URL https://doi.org/10.1145/2254064.2254111. 16, 17, 241

[102] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 85–97. ACM, 1998. doi: 10.1145/268946.268954. URL https://doi.org/10.1145/268946.268954. 3

[103] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 809–824. ACM, 2022. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432. 224

[104] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, page 568–582, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540712084. 151

[105] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, USA, 2020. USENIX Association. ISBN 978-1-939133-17-5. 202, 240

[106] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997. doi: 10.1145/263699.263712. URL https://doi.org/10.1145/263699.263712. 3

[107] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. *SIGPLAN Not.*, 41(1):320–333, January 2006. ISSN 0362-

1340. doi: 10.1145/1111320.1111066. URL https://doi.org/10.1145/1111320.1111066. 15, 151, 241

[108] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2007. doi: 10.1007/978-3-540-74591-4_15. URL https://doi.org/10.1007/978-3-540-74591-4_15. 15

[109] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, May 2020. 11, 17, 27, 98, 150, 196

[110] Marco Patrignani and Deepak Garg. Robustly safe compilation. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 469–498. Springer, 2019. doi: 10.1007/978-3-030-17184-1_17. URL https://doi.org/10.1007/978-3-030-17184-1_17. 22

[111] Marco Patrignani and Deepak Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1):1:1–1:41, 2021. doi: 10.1145/3436809. URL https://doi.org/10.1145/3436809. 22

[112] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6), February 2019. ISSN 0360-0300. doi: 10.1145/3280984. URL https://doi.org/10.1145/3280984. 196

[113] Team PaX. Pax address space layout randomization (aslr), 2003. Retrieved February 2023 from http://pax.grsecurity.net/docs/aslr.txt. 2

[114] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–5. IEEE, 2018. doi: 10.1109/HPEC.2018.8547554. URL https://doi.org/10.1109/HPEC.2018.8547554. 2

[115] Xiaojia Rao, Aïna Linn Georges, Conrad Watt, Maxime Legoupil, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-wasm: Robust and modular verification of webassembly programs. Technical report, 2023. URL https://zenodo.org/record/7708441. 21

[116] Alexander Richardson. *Complete Spatial Safety for C and C++ Using CHERI Capabilities*. PhD thesis, University of Cambridge, Computer Laboratory, 2020. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.html. 95

[117] Nick Roessler and André DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 478–495. IEEE Computer Society, 2018. doi: 10.1109/SP.2018.00066. URL https://doi.org/10.1109/SP.2018.00066. 158, 170, 196

[118] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 241–252. ACM, 2003. doi: 10.1145/888251.888274. URL https://doi.org/10.1145/888251.888274. 86

[119] Andreas Rossberg. WebAssembly core specification W3C recommendation. Technical report, W3C, December 2019. URL https://www.w3.org/TR/wasm-core-1/. 200, 242

[120] Jerome H. Saltzer. Protection and the control of information sharing in MULTICS. In Herbert Schorr, Alan J. Perlis, Peter Weiner, and W. Donald Frazer, editors, *Proceedings of the Fourth Symposium on Operating System Principles, SOSP 1973, Thomas J. Watson, Research Center, Yorktown Heights, New York, USA, October 15-17, 1973*, page 119. ACM, 1973. doi: 10.1145/800009.808059. URL https://doi.org/10.1145/800009.808059. 3

[121] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL): 32:1–32:32, 2020. URL https://doi.org/10.1145/3371100. 95, 99

[122] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 825–840. ACM, 2022. doi: 10.1145/3519939.3523434. URL https://doi.org/10.1145/3519939.3523434. 18, 23

[123] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7 (POPL):775–805, 2023. doi: 10.1145/3571220. URL https://doi.org/10.1145/3571220. 241

[124] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2001. doi: 10.1007/3-540-44577-3_6. URL https://doi.org/10.1007/3-540-44577-3_6. 3

[125] Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A semantic foundation for hidden state. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2010. doi: 10.1007/978-3-642-12032-9_2. URL https://doi.org/10.1007/978-3-642-12032-9_2. 241

[126] Thomas Sewell, Simon Winwood, Peter Gammie, Toby C. Murray, June Andronick, and Gerwin Klein. sel4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2011. doi: 10.1007/978-3-642-22863-6_24. URL https://doi.org/10.1007/978-3-642-22863-6_24. 7

[127] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561. ACM, 2007. doi: 10.1145/1315245.1315313. URL https://doi.org/10.1145/1315245.1315313. 3

[128] Lau Skorstengaard. *Formal Reasoning about Capability Machines*. PhD thesis, Aarhus University, 2019. 3, 12, 14, 15, 18, 149, 161

[129] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 475–501, 2018. doi: 10.1007/978-3-319-

89884-1_17. URL https://doi.org/10.1007/978-3-319-89884-1_17. 13, 14, 102, 103, 104, 109, 111, 112, 114, 115, 120, 123, 144, 150, 157, 160, 161, 165, 167, 183

[130] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning About a Machine with Local Capabilities. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 475–501. Springer International Publishing, 2018. 29, 95

[131] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42 (1):5:1–5:53, December 2019. ISSN 0164-0925. doi: 10.1145/3363519. 25, 29, 71, 95, 96, 98, 99, 112, 115, 123, 150, 153, 154, 165, 167

[132] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290332. URL https://doi.org/10.1145/3290332. 14, 26, 71, 96, 97, 99, 111, 123, 149, 150, 154, 155, 157, 160, 161, 188, 191, 196

[133] Ray Spencer, Stephen Smalley, Peter A. Loscocco, Mike Hibler, Dave G. Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In G. Winfield Treese, editor, *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*. USENIX Association, 1999. URL https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security. 6

[134] Marc Stiegler. Emily: A high performance language for enabling secure cooperation. In *Fifth International Conference on Creating, Connecting and Collaborating through Computing (C^5 2007), 24-26 January 2007, Kyoto, Japan*, pages 163–169. IEEE Computer Society, 2007. doi: 10.1109/C5.2007.13. URL https://doi.org/10.1109/C5.2007.13. 6

[135] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*, pages 80–95. IEEE, 2022. doi: 10.1109/CSF54842.2022.9919645. URL https://doi.org/10.1109/CSF54842.2022.9919645. 25, 29, 75, 95, 96, 97

[136] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*,

pages 161–172, 2004. doi: 10.1145/964001.964015. URL https://doi.org/10.1145/964001.964015. 79, 86

[137] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *J. ACM*, 54(5):26, 2007. doi: 10.1145/1284320.1284325. URL https://doi.org/10.1145/1284320.1284325. 161

[138] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA): 89:1–89:26, 2017. doi: 10.1145/3133913. URL https://doi.org/10.1145/3133913. 11, 27, 28, 75, 78, 79, 82, 95, 98, 99, 150, 196, 225

[139] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 48–62, 2013. doi: 10.1109/SP.2013.13. URL https://doi.org/10.1109/SP.2013.13. 2, 153

[140] Jérémy Thibault, Arthur Azevedo de Amorim, Roberto Blanco, Aïna Linn Georges, Catalin Hritcu, and Andrew Tolmach. Secomp2cheri: Securely compiling compartments from compcert c to a capability machine. 22

[141] Gavin Thomas. A proactive approach to more secure code, 2019. URL https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. 2, 153

[142] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341709. URL https://doi.org/10.1145/3341709. 150

[143] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runst. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158152. URL https://doi.org/10.1145/3158152. 150

[144] Stelios Tsampas, Dominique Devriese, and Frank Piessens. Temporal safety for stack allocated memory on capability machines. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 243–255, 2019. URL https://doi.org/10.1109/CSF.2019.00024. 141, 154, 157, 158, 159, 160, 161, 168, 196

[145] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 377–390, 2013. doi: 10.1145/2500365.2500600. URL https://doi.org/10.1145/2500365.2500600. 197

[146] Arjan van de Ven and Ingo Molnar. Exec shield, 2004. Retrieved February 2023 from https://people.redhat.com/mingo/exec-shield/docs/WHP0006US_Execshield.pdf. 2

[147] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, ICFP, 2019. 26, 95, 99, 149

[148] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.*, 31:e6, 2021. URL https://doi.org/10.1017/S0956796821000022. 196

[149] Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2012. doi: 10.1007/978-3-642-35308-6_13. URL https://doi.org/10.1007/978-3-642-35308-6_13. 16

[150] Simon Friis Vindum and Lars Birkedal. Contextual refinement of the michael-scott queue (proof pearl). In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 76–90. ACM, 2021. doi: 10.1145/3437992.3439930. URL https://doi.org/10.1145/3437992.3439930. 85

[151] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 203–216, 1993. doi: 10.1145/168619.168635. URL https://doi.org/10.1145/168619.168635. 197

[152] Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.*, 3(POPL):62:1–62:30, 2019. doi: 10.1145/3290375. URL https://doi.org/10.1145/3290375. 16

[153] Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. Compcertelf: verified separate compilation of C programs into ELF object files. *Proc. ACM Program. Lang.*, 4(OOPSLA):197:1–197:28, 2020. doi: 10.1145/3428265. URL https://doi.org/10.1145/3428265. 16

[154] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton,

M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9. 7, 8, 9, 10, 101, 102, 104, 149, 153, 161, 240

[155] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5):38–49, September 2016. ISSN 0272-1732. doi: 10.1109/MM.2016.84. 8, 101

[156] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 29–46. USENIX Association, 2010. URL http://www.usenix.org/events/sec10/tech/full_papers/Watson.pdf. 6

[157] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html. 101, 102, 105, 149

[158] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf. 7, 26, 30, 39, 153, 169

[159] Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated webassembly. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum

für Informatik, 2019. doi: 10.4230/LIPIcs.ECOOP.2019.9. URL https://doi.org/10.4230/LIPIcs.ECOOP.2019.9. 200, 207, 217, 240, 242

[160] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two mechanisations of webassembly 1.0. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 61–79. Springer, 2021. doi: 10.1007/978-3-030-90870-6_4. URL https://doi.org/10.1007/978-3-030-90870-6_4. 200, 214, 223, 224, 226, 242

[161] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 457–468. IEEE Computer Society, 2014. doi: 10.1109/ISCA.2014.6853201. URL https://doi.org/10.1109/ISCA.2014.6853201. 10

[162] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI concentrate: Practical compressed capabilities. *IEEE Trans. Computers*, 68(10):1455–1469, 2019. doi: 10.1109/TC.2019.2914037. URL https://doi.org/10.1109/TC.2019.2914037. 10, 26, 113, 168

[163] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM International Symposium on Microarchitecture*. ACM, October 2019. doi: 10.1145/3352460.3358288. 67, 149, 195

[164] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 175–188. ACM, 2004. doi: 10.1145/1016850.1016875. URL https://doi.org/10.1145/1016850.1016875. 15, 241

[165] Dachuan Yu, Nadeem Abdul Hamid, and Zhong Shao. Building certified libraries for PCC: dynamic storage allocation. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on*

Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, volume 2618 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2003. doi: 10.1007/3-540-36575-3_25. URL https://doi.org/10.1007/3-540-36575-3_25. 15, 241