Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts

Sander Huyghebaert
Steven Keuchel
Coen De Roover
Dominique Devriese

Outline

Introduction

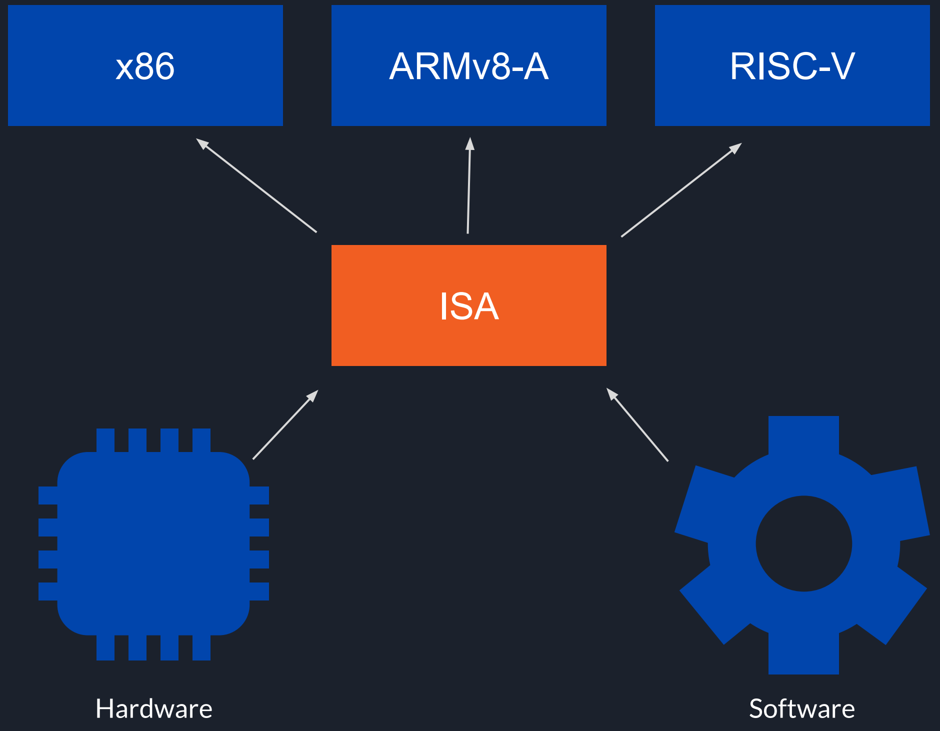
Universal Contracts

MinimalCaps

RISC-V PMP

Conclusion

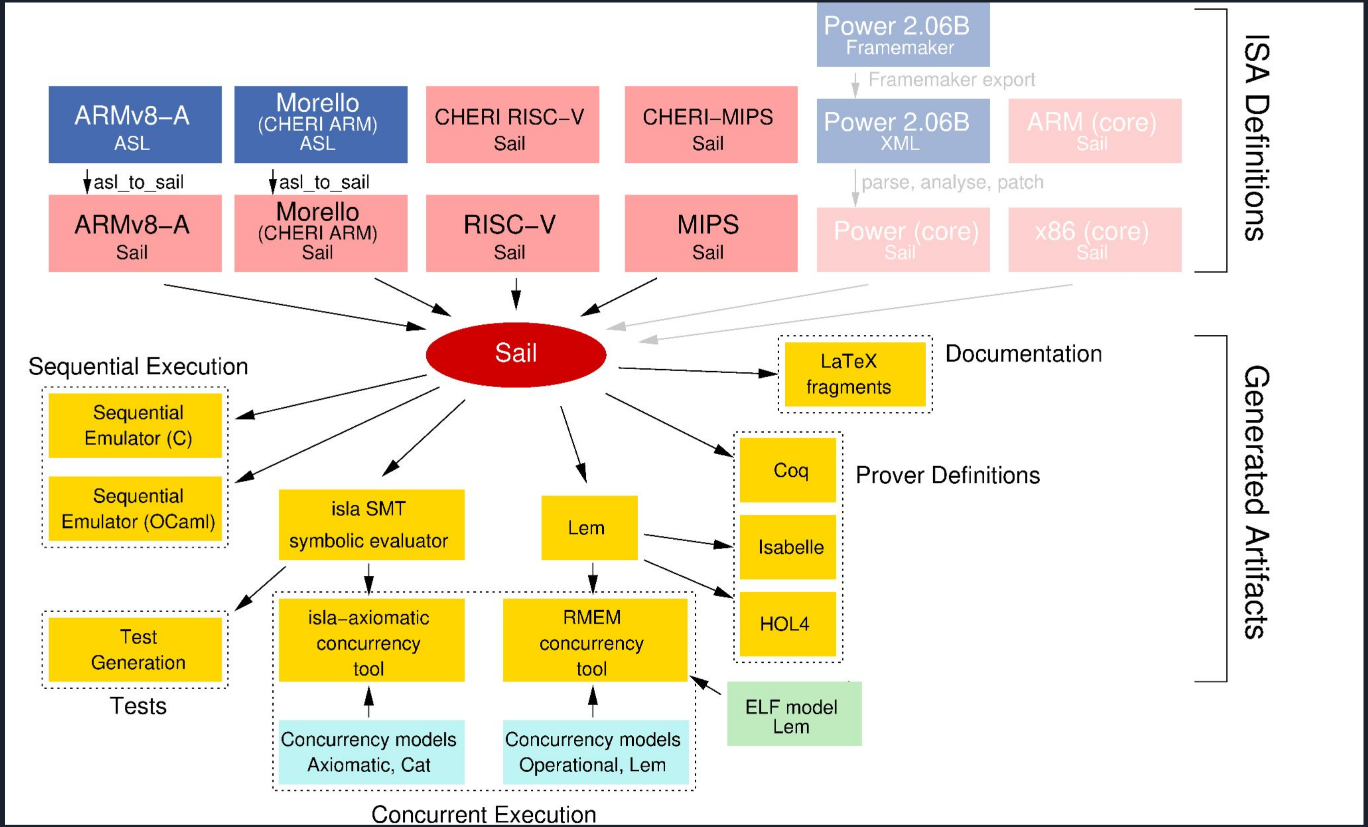
Introduction



Traditionally:
- Long manuals
- Prose/Pseudocode

Recently:
- Formal & executable spec

Sail



Power 2.06B
Framemaker
Framemaker export
ARMv8–A
ASL
Morello
(CHERI ARM)
ASL
CHERI RISC–V
Sail
CHERI–MIPS
Sail
Power 2.06B
XML
ARM (core)
Sail
asl_to_sail
asl_to_sail
parse, analyse, patch
ARMv8–A
Sail
Morello
(CHERI ARM)
Sail
RISC–V
Sail
MIPS
Sail
Power (core)
Sail
x86 (core)
Sail
ISA Definitions
Sail
Sequential Execution
Sequential
Emulator (C)
Sequential
Emulator (OCaml)
LaTeX
fragments
Documentation
Coq
Prover Definitions
isla SMT
symbolic evaluator
Lem
Isabelle
HOL4
Generated Artifacts
Test
Generation
Tests
isla–axiomatic
concurrency
tool
RMEM
concurrency
tool
ELF model
Lem
Concurrency models
Axiomatic, Cat
Concurrency models
Operational, Lem
Concurrent Execution

Security Guarantees

Example: AMD64

“Only privileged software running at CPL=0 can manage the TLBs.”

> “Page translation is controlled by the PG bit in CR0 (bit 31). When CR0.PG is set to 1, page translation is enabled.”

“Most instructions used to access these resources are privileged and can only be executed while the processor is running at CPL=0, although some instructions can be executed at any privilege level.”

- AMD64 Architecture Programmer’s Manual Volume 2: System Programming

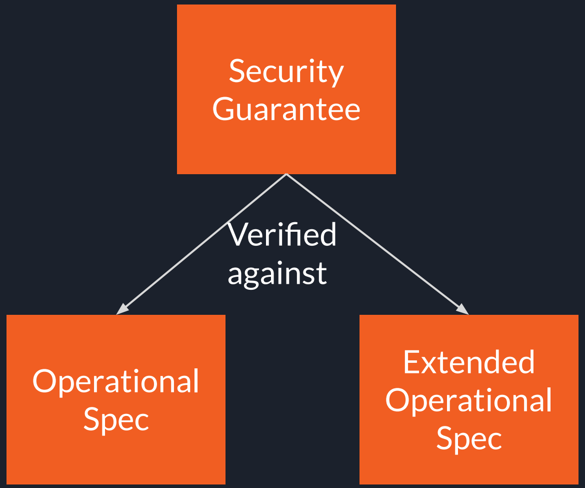
Security Guarantees

Current Approach

- Informal ISA specs offer *promise* of security guarantee
  - “Security feature *X* offers *Y* / prevents attack *Z*”
  - Holds for future updates to the ISA
- Formal ISA specs *lack* security specifications
  - Focus is on operational specification

Universal Contracts

Motivation



- Security guarantees should be
  - Part of ISA specification
  - Formal
  - Verifiable against operational spec
  - Specific enough for reasoning
  - Not overspecified
    - Optimizations and extensions should be possible
  - Mechanized
- Current approaches do *not* meet these requirements

Universal Contracts

Concept

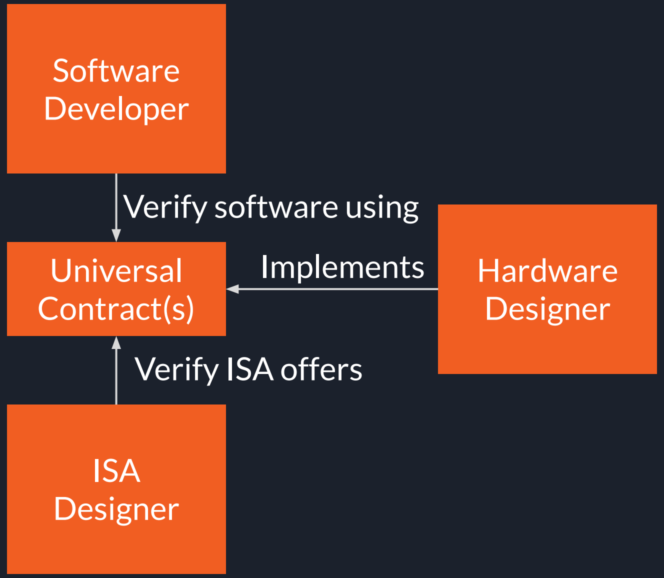
{ P } ASM code { Q }

- Formal security guarantee…
- … expressed as a contract
    - Upper bound of the authority
- Holds for *any* code
- Verifiable against operational specification of ISA
    - Sail
    - Fetch-Decode-Execute Cycle

Universal Contracts
Concept

{ P } ASM code { Q }

- Formal security guarantee…
- … expressed as a contract
  - Upper bound of the authority
- Holds for *any* code
- Verifiable against operational specification of ISA
  - Sail
  - Fetch-Decode-Execute Cycle

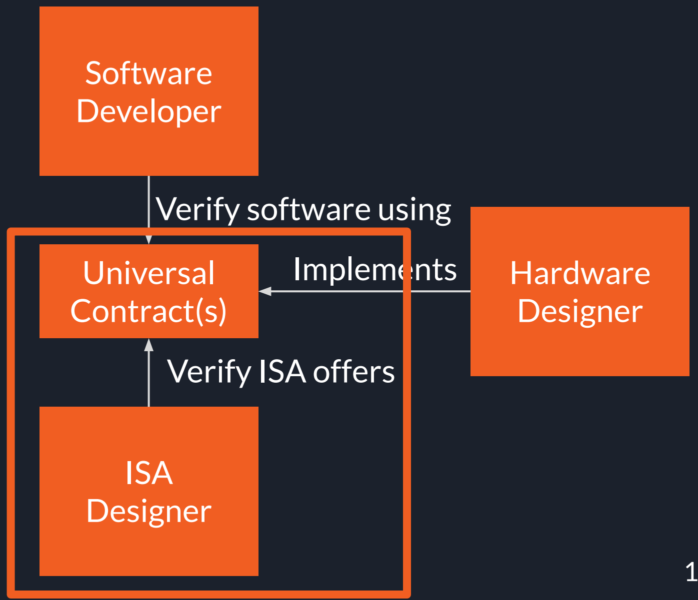


Universal Contracts

Concept

{ P } *ASM code* { Q }

- Formal security guarantee…
- … expressed as a contract
  - Upper bound of the authority
- Holds for *any* code
- Verifiable against operational specification of ISA
  - Sail
  - Fetch-Decode-Execute Cycle



Outline

Introduction

Universal Contracts

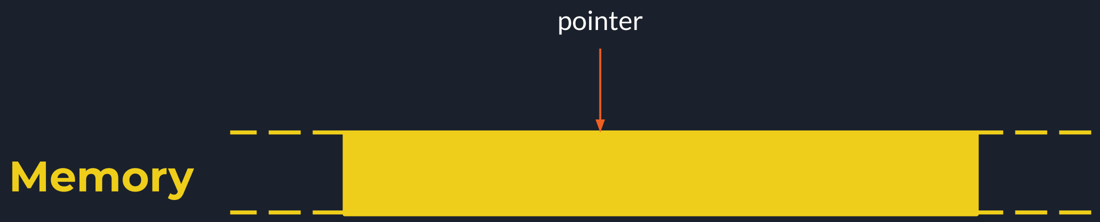
MinimalCaps

RISC-V PMP

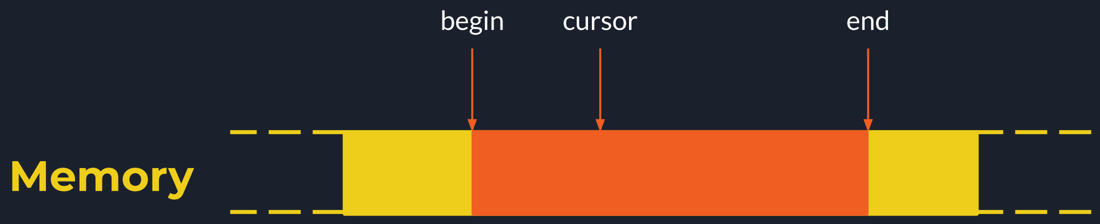
Conclusion

MinimalCaps

Traditional Machine



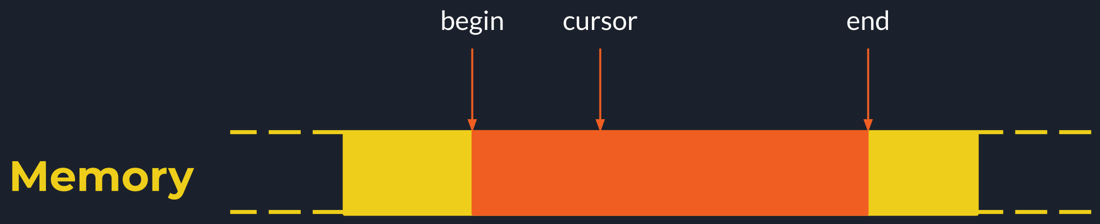
The MinimalCaps Capability Machine



Capability

- perm ∈ {O, E, R, RW}
- cursor : address
- begin : address
- end : address

The MinimalCaps Capability Machine



Capability

- perm ∈ {O, E, R, RW}
- cursor : address
- begin : address
- end : address

Hardware Guarantees

- Capabilities are unforgeable
- Permissions are checked
- Capability manipulation is safe

$\text{CorrectPC}(p, b, e, a) = a \in [b, e) \ast \mathrm{R} \sqsubseteq p$

Capability Safety

Universal Contract

$\{ (\exists\ c, pc \mapsto c \ast \mathcal{V}(c) \ast \text{CorrectPC}(c)) \ast (\ast_{r \in GPR} \exists\ w.\ r \mapsto w \ast \mathcal{V}(w)) \}$

fdeCycle

$\{ \top \}$

Capability Safety

Logical Relations

Value Relation $\mathcal{V}$: (Integer + Capability) -> iProp

$$\mathcal{V}(w) \begin{cases} \mathcal{V}(z), \mathcal{V}(O, -, -, -) = \text{True } (z \text{ is an integer}) \\ \mathcal{V}(E, b, e, a) \quad = \triangleright \square\, \boldsymbol{\varepsilon}(R, b, e, a) \\ \mathcal{V}(R, b, e, -) \quad = \mathop{\ast}_{a \in [b, e]} \boxed{\exists\, w, a \mapsto w \ast \mathcal{V}(w)} \\ \mathcal{V}(RW, b, e, -) \quad = \mathop{\ast}_{a \in [b, e]} \boxed{\exists\, w, a \mapsto w \ast \mathcal{V}(w)} \end{cases}$$

Expression Relation $\boldsymbol{\varepsilon}$: (Integer + Capability) -> iProp

$$\boldsymbol{\varepsilon}(w) = (pc \mapsto w \ast (\mathop{\ast}_{r \in GPR} \exists\, w.\ r \mapsto w \ast \mathcal{V}(w))) -\!\!\ast \text{ wp fdeCycle T}$$

Capability Safety

Logical Relations

Value Relation $\mathcal{V}$: (Integer + Capability) -> iProp

$$
\mathcal{V}(w) \begin{cases} \mathcal{V}(z), \mathcal{V}(O,-,-,-) = \text{True (z is an integer)} \\ \mathcal{V}(E,b,e,a) = \triangleright \Box\, \mathcal{E}(R,b,e,a) \\ \mathcal{V}(R,b,e,-) = \mathop{\ast}_{a \in [b,e]} \exists\, w, a \mapsto w \ast \mathcal{V}(w) \\ \mathcal{V}(RW,b,e,-) = \mathop{\ast}_{a \in [b,e]} \exists\, w, a \mapsto w \ast \mathcal{V}(w) \end{cases}
$$

Expression Relation $\mathcal{E}$: (Integer + Capability) -> iProp

$$
\mathcal{E}(w) = (pc \mapsto w \ast (\mathop{\ast}_{r \in GPR} \exists\, w.\, r \mapsto w \ast \mathcal{V}(w))) \mathrel{-\!\!\ast} \text{wp fdeCycle T}
$$

Capability Safety

Logical Relations

Value Relation $\mathcal{V}$: (Integer + Capability) -> iProp

$\mathcal{V}(w)$

- $\mathcal{V}(z), \mathcal{V}(O, -, -, -)$ = True (z is an integer)
- $\mathcal{V}(E, b, e, a) = \triangleright \square\, \mathcal{E}(R, b, e, a)$
- $\mathcal{V}(R, b, e, -) = \ast_{a \in [b, e]}\ \exists\, w, a \mapsto w \ast \mathcal{V}(w)$
- $\mathcal{V}(RW, b, e, -) = \ast_{a \in [b, e]}\ \exists\, w, a \mapsto w \ast \mathcal{V}(w)$

Invariants

Expression Relation $\mathcal{E}$: (Integer + Capability) -> iProp

$\mathcal{E}(w) = (pc \mapsto w \ast (\ast_{r \in GPR}\ \exists\, w.\ r \mapsto w \ast \mathcal{V}(w))) -\!\ast$ wp fdeCycle T

Capability Safety

Logical Relations

Value Relation $\mathcal{V}$: (Integer + Capability) -> iProp

$$\mathcal{V}(w) \begin{cases} \mathcal{V}(z), \mathcal{V}(O,-,-,-) = \text{True } (z \text{ is an integer}) \\ \mathcal{V}(E,b,e,a) = \triangleright \square\, \mathcal{E}(R,b,e,a) \\ \mathcal{V}(R,b,e,-) = \ast_{a \in [b,e]} \exists\, w, a \mapsto w \ast \mathcal{V}(w) \\ \mathcal{V}(RW,b,e,-) = \ast_{a \in [b,e]} \exists\, w, a \mapsto w \ast \mathcal{V}(w) \end{cases}$$

Expression Relation $\mathcal{E}$: (Integer + Capability) -> iProp

$$\mathcal{E}(w) = (pc \mapsto w \ast (\ast_{r \in GPR} \exists\, w.\ r \mapsto w \ast \mathcal{V}(w))) \twoheadrightarrow \text{wp fdeCycle T}$$

$\text{CorrectPC}(p, b, e, a) = a \in [b, e) \ast R \sqsubseteq p$

Capability Safety

Step Contract

$\{ (\exists\ c, pc \mapsto c \ast \mathcal{V}(c) \ast \text{CorrectPC}(c)) \ast (\ast_{r \in GPR}\ \exists\ w.\ r \mapsto w \ast \mathcal{V}(w)) \}$

step

$\{ (\exists\ c, pc \mapsto c \ast (\mathcal{V}(c) \vee \boldsymbol{\varepsilon}(c))) \ast (\ast_{r \in GPR}\ \exists\ w.\ r \mapsto w \ast \mathcal{V}(w)) \}$

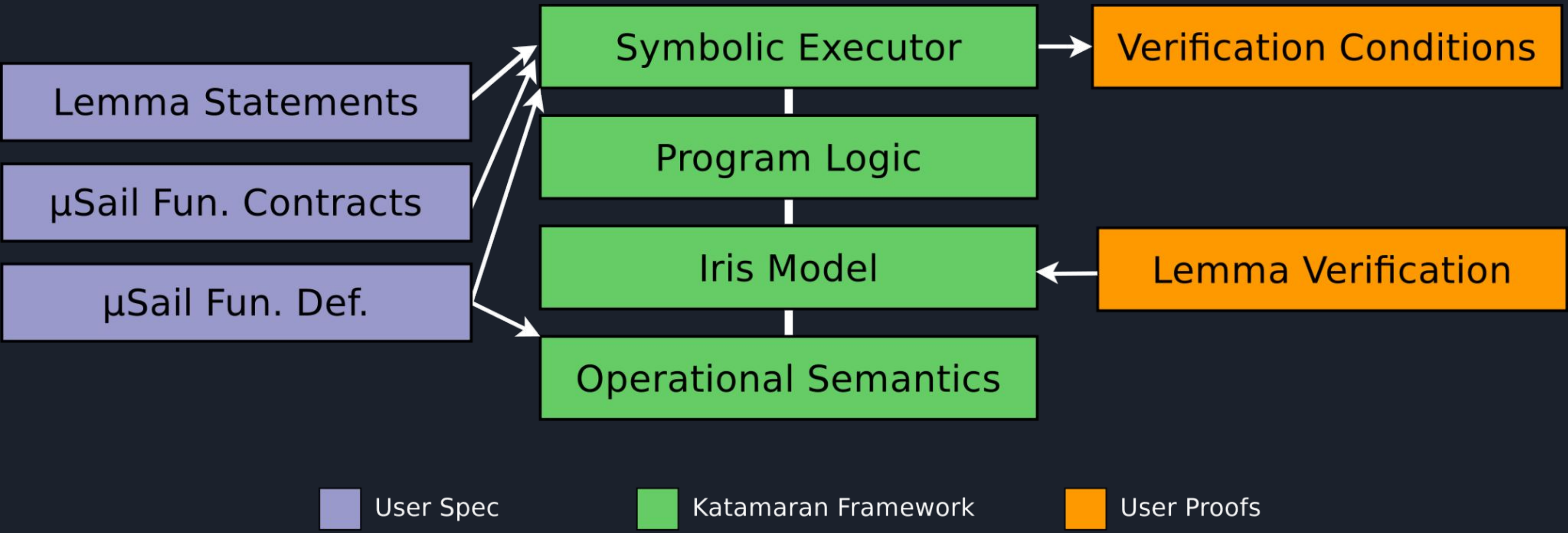
Katamaran

Semi-automatic separation logic verifier

97

Verified Symbolic Execution with Kripke Specification Monads (and No Meta-programming)

STEVEN KEUCHEL, Vrije Universiteit Brussel, Belgium
SANDER HUYGHEBAERT, Vrije Universiteit Brussel, Belgium
GEORGY LUKYANOV, Newcastle University, United Kingdom
DOMINIQUE DEVRIESE, KU Leuven, Belgium



Verifying MinimalCaps' Security Guarantees

$\{ (\exists\ c, pc \mapsto c * \mathcal{V}(c) * \text{CorrectPC}(c)) * (\ast_{r \in GPR}\ \exists\ w.\ r \mapsto w * \mathcal{V}(w)) \}$

```
function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let b := call write_allowed perm in
  assert b ;;
  let w := call read_reg rs in
  use lemma (subperm_not_E RW perm) ;;
  use lemma (move_cursor base_cap c) ;;
  call write_mem c w ;;
  call update_pc ;; true
```

$\{ (\exists\ c, pc \mapsto c * (\mathcal{V}(c) \lor \boldsymbol{\varepsilon}(c))) * (\ast_{r \in GPR}\ \exists\ w.\ r \mapsto w * \mathcal{V}(w)) \}$

{ (p = R ∨ p = RW) ✱ $p \sqsubseteq p'$ } subperm_not_E p p' { $p' \neq$ E }

Verifying MinimalCaps' Security Guarantees

```
{ (∃ c, pc ↦ c ✱ 𝒱(c) ✱ CorrectPC(c)) ✱ (✱_{r ∈ GPR} ∃ w. r ↦ w ✱ 𝒱(w)) }
function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let b := call write_allowed perm in
  assert b ;;
  let w := call read_reg rs in
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ...}
  use lemma (subperm_not_E RW perm) ;;
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ✱ perm ≠ E ...}
  use lemma (move_cursor base_cap c) ;;
  call write_mem c w ;;
  call update_pc ;; true
{ (∃ c, pc ↦ c ✱ (𝒱(c) ∨ ε(c))) ✱ (✱_{r ∈ GPR} ∃ w. r ↦ w ✱ 𝒱(w)) }
```

{ $\mathcal{V}(p, b, e, a) * p \neq E$ } move_cursor (p, b, e, a) (p, b, e, a') { $\mathcal{V}(p, b, e, a) * \mathcal{V}(p, b, e, a')$ }

Verifying MinimalCaps' Security Guarantees

{ $(\exists\ c, pc \mapsto c * \mathcal{V}(c) * \text{CorrectPC}(c)) * (\ast_{r \in GPR} \exists\ w.\ r \mapsto w * \mathcal{V}(w))$ }

```
function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let b := call write_allowed perm in
  assert b ;;
  let w := call read_reg rs in
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ...}
  use lemma (subperm_not_E RW perm) ;;
  use lemma (move_cursor base_cap c) ;;
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ✱ 𝒱(c) ...}
  call write_mem c w ;;
  call update_pc ;; true
```

{ $(\exists\ c, pc \mapsto c * (\mathcal{V}(c) \vee \varepsilon(c))) * (\ast_{r \in GPR} \exists\ w.\ r \mapsto w * \mathcal{V}(w))$ }

{ $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ } write_mem *c w* { $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ }

Verifying MinimalCaps' Security Guarantees

{ (∃ c, pc ↦ c ✱ $\mathcal{V}$(c) ✱ CorrectPC(c)) ✱ ($✱_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

```
function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let b := call write_allowed perm in
  assert b ;;
  let w := call read_reg rs in
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ...}
  use lemma (subperm_not_E RW perm) ;;
  use lemma (move_cursor base_cap c) ;;
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ✱ 𝒱(c) ...}
  call write_mem c w ;;
  call update_pc ;; true
```

{ (∃ c, pc ↦ c ✱ ($\mathcal{V}$(c) ∨ **ε**(c))) ✱ ($✱_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

{ $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ } write_mem *c w* { $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ }

Verifying MinimalCaps' Security Guarantees

{ (∃ c, pc ↦ c ✱ $\mathcal{V}$(c) ✱ CorrectPC(c)) ✱ (✱$_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

```
function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=
  let base_cap := call read_reg_cap rb in
  let (perm, beg, end, cursor) := base_cap in
  let c := (perm, beg, end, cursor + immediate) in
  let b := call write_allowed perm in
  assert b ;;
  let w := call read_reg rs in
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ...}
  use lemma (subperm_not_E RW perm) ;;
  use lemma (move_cursor base_cap c) ;;
  { rb ↦ base_cap ✱ 𝒱(base_cap) ✱ rs ↦ w ✱ 𝒱(w) ✱ 𝒱(c) ...}
  call write_mem c w ;;
  call update_pc ;; true
```

{ (∃ c, pc ↦ c ✱ ($\mathcal{V}$(c) ∨ ε(c))) ✱ (✱$_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

RISC-V PMP

RISC-V

- Free, open, extensible ISA
- 32-bit instructions
- We focus on RV32I
    - Official RISC-V spec
    - But with PMP support
    - U and M modes
- Simplifications
    - Limited PMP entries

Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
C	16-bit Compressed Instructions
Xext	Non-Standard User-Level Extension

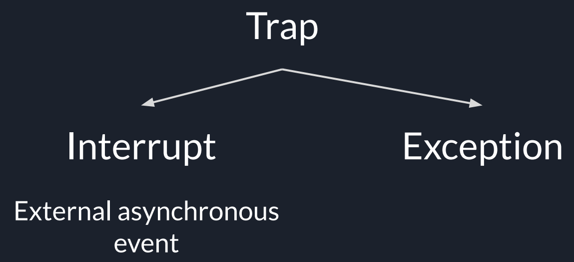
G (I, M, A, F, D)

RISC-V PMP

Physical Memory Protection

- Optional
- *Grant* permissions to S and U modes
  - By default *none*
- *Revoke* permissions from M mode
  - By default *full*
- PMP violations => trap
  - Load access fault, store access fault, …
  - Exception
- Up to 64 PMP regions
  - Statically prioritized
  - Lowest number has highest priority

RISC-V PMP
Physical Memory Protection

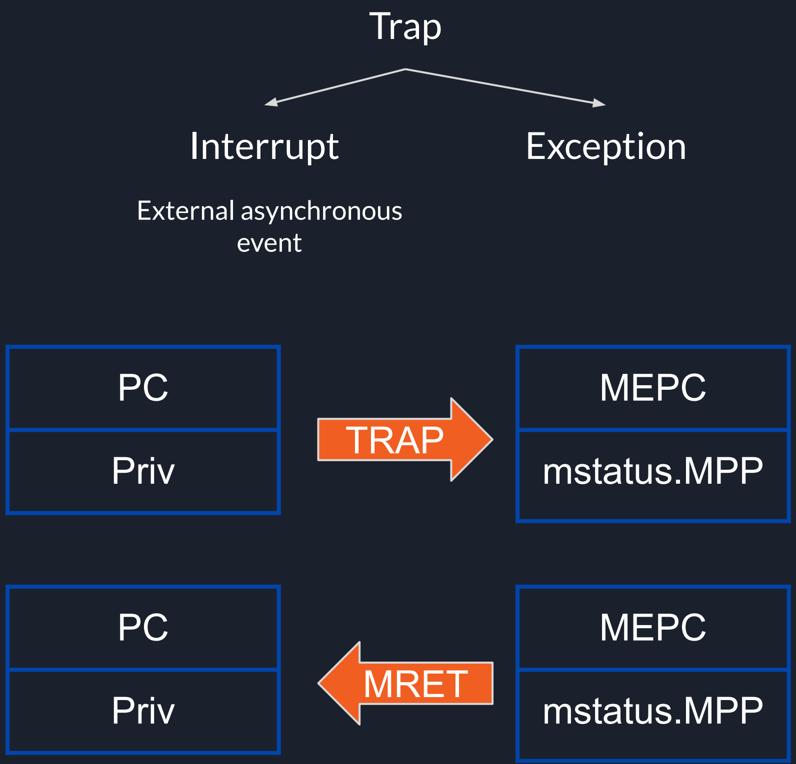


- Optional
- *Grant* permissions to S and U modes
  - By default *none*
- *Revoke* permissions from M mode
  - By default *full*
- PMP violations => trap
  - Load access fault, store access fault, …
  - Exception
- Up to 64 PMP regions
  - Statically prioritized
  - Lowest number has highest priority

RISC-V PMP

Physical Memory Protection

- Optional
- *Grant* permissions to S and U modes
    - By default *none*
- *Revoke* permissions from M mode
    - By default *full*
- PMP violations => trap
    - Load access fault, store access fault, …
    - Exception
- Up to 64 PMP regions
    - Statically prioritized
    - Lowest number has highest priority



RISC-V PMP

PMP Entry

permissions

$$(i, lock, rwx, address)$$

lock ∈ {0, 1}

upper bound

RISC-V PMP

PMP Entry

permissions

$(i, lock, rwx, address)$

lock ∈ {0, 1}

upper bound

Bounds $\text{PMP}_i = [address_{i-1}, address_i)$

RISC-V PMP

State Transition: Trap

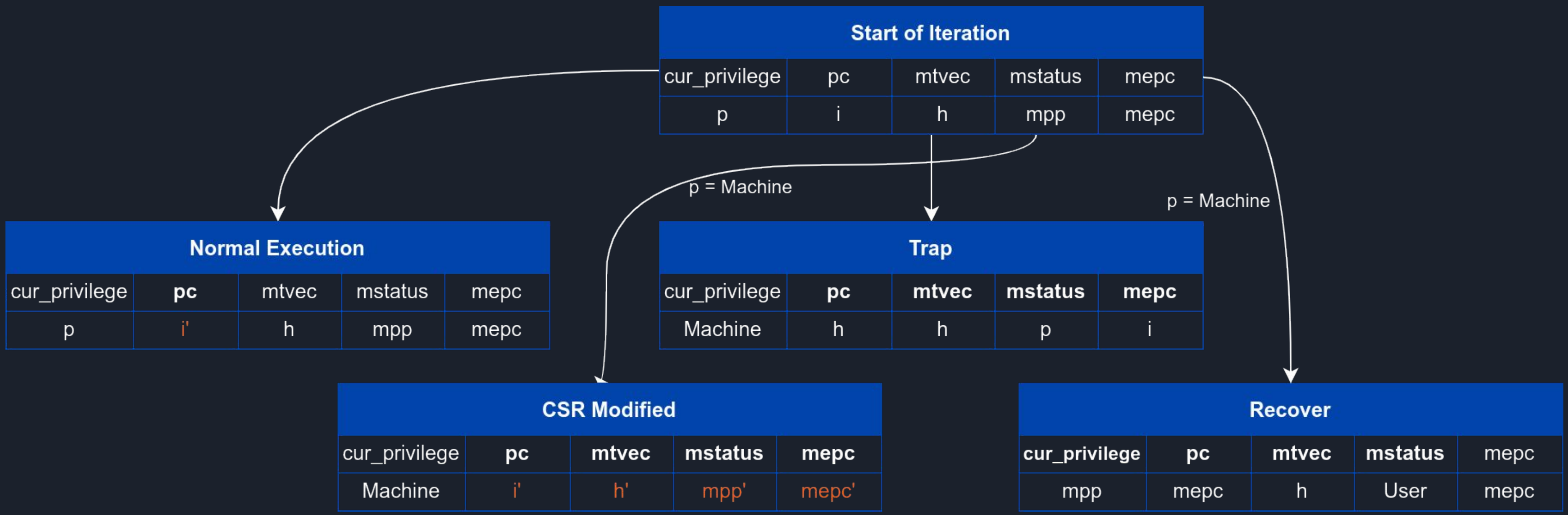
Start of Iteration				
cur_privilege	pc	mtvec	mstatus	mepc
p	i	h	mpp	mepc

↓

Trap				
cur_privilege	**pc**	**mtvec**	**mstatus**	**mepc**
Machine	h	h	p	i

RISC-V PMP

State Transition



RISC-V PMP

Universal Contract

{ **Start**

∗ ▷ (**CSRMod** -∗ wp fdeCycle T)

∗ ▷ (**Trap** -∗ wp fdeCycle T)

∗ ▷ (**Recover** -∗ wp fdeCycle T) }

fdeCycle

{ T }

RISC-V PMP
Universal Contract

{ **Start**

$*$ ▷ (**CSRMod** -$*$ wp fdeCycle T)

$*$ ▷ (**Trap** -$*$ wp fdeCycle T)

$*$ ▷ (**Recover** -$*$ wp fdeCycle T) }

fdeCycle

{ T }

RISC-V PMP

Full-System Proof: FemtoKernel

```
init:
    la    ra, adv
    csrrw zero, pmpaddr0, ra
    lui   ra, max
    csrrw zero, pmpaddr1, ra
    lui   ra, 0x0
    csrrw zero, pmp0cfg, ra
    lui   ra, 0xf
    csrrw zero, pmp1cfg, ra
    la    ra, adv
    csrrw zero, mepc, ra
    la    ra, trap_handler
    csrrw zero, mtvec, ra
    lui   ra, 0x0
    csrrw zero, mstatus, ra
    mret

trap_handler:
    auipc ra, 0
    lw    ra, 12(ra)
    mret

data: 42
adv:  …
```

Memory Integrity:
data will always contain the value *42*

RISC-V PMP

Full-System Proof: FemtoKernel

```
init:
    la    ra, adv
    csrrw zero, pmpaddr0, ra
    lui   ra, max
    csrrw zero, pmpaddr1, ra
    lui   ra, 0x0
    csrrw zero, pmp0cfg, ra
    lui   ra, 0xf
    csrrw zero, pmp1cfg, ra
    la    ra, adv
    csrrw zero, mepc, ra
    la    ra, trap_handler
    csrrw zero, mtvec, ra
    lui   ra, 0x0
    csrrw zero, mstatus, ra
    mret

trap_handler:
    auipc ra, 0
    lw    ra, 12(ra)
    mret

data: 42
adv:  …
```

Configure PMP

Memory Integrity:
data will always contain the value *42*

RISC-V PMP

Full-System Proof: FemtoKernel

```
init:
    la    ra, adv
    csrrw zero, pmpaddr0, ra
    lui   ra, max
    csrrw zero, pmpaddr1, ra
    lui   ra, 0x0
    csrrw zero, pmp0cfg, ra
    lui   ra, 0xf
    csrrw zero, pmp1cfg, ra
    la    ra, adv
    csrrw zero, mepc, ra
    la    ra, trap_handler
    csrrw zero, mtvec, ra
    lui   ra, 0x0
    csrrw zero, mstatus, ra
    mret

trap_handler:
    auipc ra, 0
    lw    ra, 12(ra)
    mret

data: 42
adv: …
```

Configure PMP

Setup Trap Handler, Jump to *adv*

Memory Integrity:
data will always contain the value *42*

RISC-V PMP

Full-System Proof: FemtoKernel

```
init:
    la     ra, adv
    csrrw  zero, pmpaddr0, ra
    lui    ra, max
    csrrw  zero, pmpaddr1, ra
    lui    ra, 0x0
    csrrw  zero, pmp0cfg, ra
    lui    ra, 0xf
    csrrw  zero, pmp1cfg, ra
    la     ra, adv
    csrrw  zero, mepc, ra
    la     ra, trap_handler
    csrrw  zero, mtvec, ra
    lui    ra, 0x0
    csrrw  zero, mstatus, ra
    mret

trap_handler:
    auipc  ra, 0
    lw     ra, 12(ra)
    mret

data: 42
adv:  …
```

Configure PMP

Setup Trap Handler, Jump to *adv*

Store value of Memory[*data*] into *ra*

Memory Integrity:
data will always contain the value *42*

Conclusion

Evaluation

- Comparison with Cerise
  - Proof effort reduction
  - Not entirely fair
- Added Instruction
  - Uninteresting case
  - RISC-V PMP: +2 LoC
  - MinimalCaps: +23 LoC
- FemtoKernel

	MinimalCaps	RISC-V PMP	Cerise
LoC	2867	3880	7919

Summary

- Security Guarantees
  - Formalized as Universal Contracts
  - Part of *security* specification
  - Verified against *operational* specification
- Case Study: MinimalCaps
  - Capability safety
- Case Study: RISC-V PMP
  - Memory Integrity
- Katamaran
  - Semi-automatic separation logic verifier

This presentation is part of the UniversalContracts project that has received funding from the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (Grant agreement No. 101040088).
The work was also supported in part by the Research Foundation - Flanders (FWO), by the Flemish Research Programme Cybersecurity.
Views and opinions expressed are those of the author(s) only.

Side-channels
Future Work

- Current focus on *integrity* guarantees
- Software observable side-channels
    - Timing-based side-channels (Instruction Timing)
- Should be part of security specification
    - ISA should not specify all details…
    - … but enough to reason about it

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

AÏNA LINN GEORGES, Aarhus University, Denmark
ARMAËL GUÉNEAU, Aarhus University, Denmark
THOMAS VAN STRYDONCK, KU Leuven, Belgium
AMIN TIMANY, Aarhus University, Denmark
ALIX TRIEU, Aarhus University, Denmark
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

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 [Georges et al. 2021; Skorstengaard et al. 2019a; Van Strydonck et al. 2021], but was left somewhat implicit in those publications. In this paper we present a pedagogical introduction 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.

ACM Reference Format:
Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2021. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 1, 1 (October 2021), 55 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [Dennis and Van Horn 1966; Levy 1984], and in particular more recently as part of the CHERI project [Watson et al. 2020]. Capability machines offer fine-grained and scalable

Authors' addresses: Aïna Linn Georges, ageorges@cs.au.dk, Aarhus University, Denmark; Armaël Guéneau, armael@cs.au.dk, Aarhus University, Denmark; Thomas Van Strydonck, thomas.vanstrydonck@cs.kuleuven.be, KU Leuven, Belgium; Amin Timany, timany@cs.au.dk, Aarhus University, Denmark; Alix Trieu, alix.trieu@cs.au.dk, Aarhus University, Denmark; Dominique Devriese, dominique.devriese@vub.be, Vrije Universiteit Brussel, Belgium; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Denmark.

0004-5411/2021/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Verified security for the Morello capability-enhanced prototype Arm architecture

THOMAS BAUEREISS, University of Cambridge, UK
BRIAN CAMPBELL, University of Edinburgh, UK
THOMAS SEWELL, University of Cambridge, UK
ALASDAIR ARMSTRONG, University of Cambridge, UK
LAWRENCE ESSWOOD, University of Cambridge, UK
IAN STARK, University of Edinburgh, UK
GRAEME BARNES, Arm Ltd., UK
ROBERT N. M. WATSON, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

Memory safety bugs continue to be a major source of security vulnerabilities in our critical infrastructure. The CHERI project has proposed extending conventional architectures with hardware-supported *capabilities* to enable fine-grained memory protection and scalable compartmentalisation, allowing historically memory-unsafe C and C++ to be adapted to deterministically mitigate large classes of vulnerabilities, while requiring only minor changes to existing system software sources. Arm is currently designing and building Morello, a CHERI-enabled prototype architecture, processor, SoC, and board, extending the high-performance Neoverse N1, to enable industrial evaluation of CHERI and pave the way for potential mass-market adoption. However, for such a major new security-oriented architecture feature, it is important to establish high confidence that it does provide the protections it intends to, and that cannot be done with conventional engineering techniques.

In this paper we put the Morello architecture on a solid mathematical footing from the outset. We define the fundamental security property that Morello aims to provide, reachable capability monotonicity, and prove that the architecture definition satisfies it. This proof is mechanised in Isabelle/HOL, and applies to a translation of the official Arm Morello specification into Isabelle. The main challenge is handling the complexity and scale of a production architecture: 62,000 lines of specification, translated to 210,000 lines of Isabelle. We do so by factoring the proof via a narrow abstraction capturing the essential properties of instruction execution in an arbitrary CHERI ISA, expressed above a monadic intra-instruction semantics. We also develop a model-based test generator, which generates instruction-sequence tests that give good specification coverage, used in early testing of the Morello implementation and in Morello QEMU development. We also use Arm's internal test suite to validate our internal model.

This gives us machine-checked mathematical proofs of whole-ISA security properties of a full-scale industry architecture, at design-time. To the best of our knowledge, this is the first demonstration that that is feasible, and it significantly increases confidence in Morello.

1 INTRODUCTION

1.1 The CHERI and Morello Context

Memory safety bugs continue to be a major source of security vulnerabilities, responsible for around 70% of those addressed by Microsoft security updates, and around 70% of the high-severity bugs impacting Chromium [28, 42]. Their root causes are well-known legacy design choices and limitations of normal practice: pervasive uses of systems programming languages that do not enforce memory protection; hardware that enforces only coarse-grain protection, with virtual memory; and test-and-debug development methods that cannot provide high assurance. These are baked in to the critical systems codebase across the industry, and the result, in today's adversarial environment, is that programming errors can often lead to exploitable vulnerabilities.

There are many possible approaches to improving this situation, including development of safer programming languages, techniques for full functional-correctness verification, and better bug-finding tools. Each is the subject of much research in programming languages and semantics, and all are worthwhile, but the legacy investment, the need for systems code to work close to the machine, and the inability of bug-finding to provide high assurance, makes it very hard to radically improve mass-market systems.

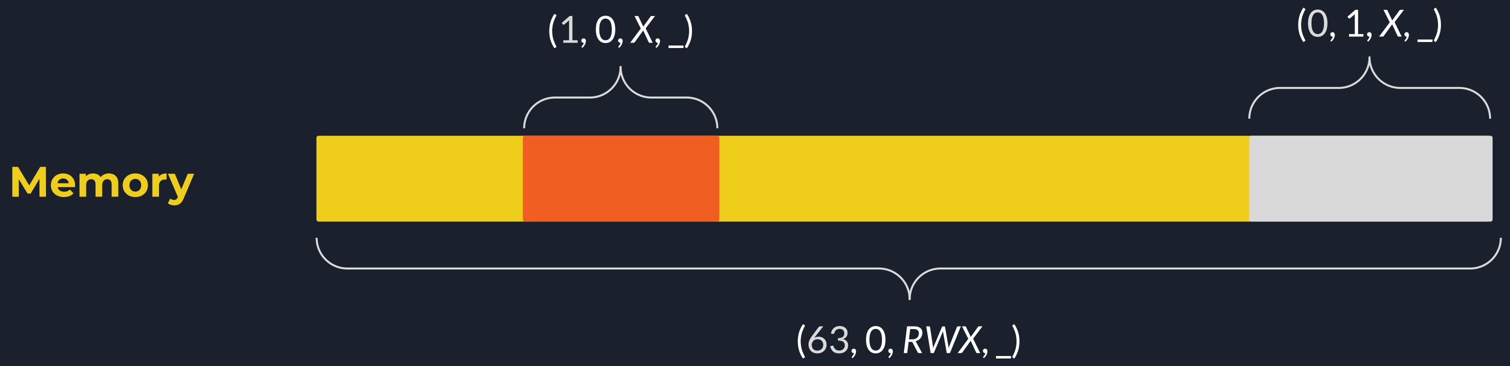
Another path, less well explored, is to change the architectural interface to provide hardware mechanisms that enable better enforcement of memory protection. Over the last ten years, the CHERI project [1] has been extending conventional hardware Instruction-Set Architectures (ISAs) with new architectural features to enable fine-grained memory protection and highly scalable software compartmentalisation. The CHERI memory-protection features allow historically memory-unsafe programming languages such as C and C++ to be adapted to have quite different

Authors' addresses: Thomas Bauereiss, Thomas.Bauereiss@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Brian Campbell, Brian.Campbell@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Thomas Sewell, Thomas.Sewell@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Alasdair Armstrong, Alasdair.Armstrong@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Lawrence Esswood, le277@cam.ac.uk, University of Cambridge, Cambridge, UK; Ian Stark, Ian.Stark@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Graeme Barnes, Graeme.Barnes@arm.com, Arm Ltd., Cambridge, UK; Robert N. M. Watson, Robert.Watson@cl.cam.ac.uk, University of Cambridge, Cambridge, UK; Peter Sewell, Peter.Sewell@cl.cam.ac.uk, University of Cambridge, Cambridge, UK.

Version of 2021-09-06 08:49.

RISC-V PMP

Example



{ $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ } write_mem *c w* { $\mathcal{V}(c)$ ✱ $\mathcal{V}(w)$ }

Verifying MinimalCaps' Security Guarantees

{ (∃ c, pc ↦ c ✱ $\mathcal{V}$(c) ✱ CorrectPC(c)) ✱ (✱$_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

function exec_sd(rs : GPR, rb : GPR, immediate : int) : bool :=

  let base_cap := **call** read_reg_cap rb **in**

  let (perm, beg, end, cursor) := base_cap **in**

  let c := (perm, beg, end, cursor + immediate) **in**

  let b := **call** write_allowed perm **in**

  assert b ;;

  let w := **call** read_reg rs **in**

  use lemma (subperm_not_E RW perm) ;;

  use lemma (move_cursor base_cap c) ;;

  { rb ↦ base_cap ✱ $\mathcal{V}$(base_cap) ✱ rs ↦ w ✱ $\mathcal{V}$(w) ✱ $\mathcal{V}$(c) ...}

  call write_mem c w ;;

  { rb ↦ base_cap ✱ $\mathcal{V}$(base_cap) ✱ rs ↦ w ✱ $\mathcal{V}$(w) ✱ $\mathcal{V}$(c) ...}

  call update_pc ;; true

{ (∃ c, pc ↦ c ✱ ($\mathcal{V}$(c) ∨ **ε**(c))) ✱ (✱$_{r \in GPR}$ ∃ w. r ↦ w ✱ $\mathcal{V}$(w)) }

Contract

Execute

{ (∃ c, pc ↦ c ∗ $\mathcal{V}$(c) ∗ CorrectPC(c)) ∗ (∗$_{r \in GPR}$ ∃ w. r ↦ w ∗ $\mathcal{V}$(w)) }

```
function execute() : bool :=
  let c := call read_reg_cap pc in
  let n := call read_mem c in
  match n with
  | inl n =>
    let i := call decode n in
    call exec_instr i
  | inr c => fail
  end
```

{ wp fdeCycle T }

Fetch

Decode

Execute

Future Work

CCS'22 Submission

Capability Safety
Capability safety of the MinimalCaps machine

RISC-V PMP
Memory integrity for RISC-V PMP with synchronous interrupts

Object Capabilities
Add support for object capabilities to the MinimalCaps case study

Virtual Memory
Verification of security properties of Virtual Memory

Proof Automation
Further improve proof automation of Katamaran

Larger ISAs
Scale up the number of instructions in ISAs we consider

Complex ISAs
Introduce features such as concurrency, interrupts, ...

Realistic ISAs
Verify security properties of real ISAs, i.e. RV32G..., CHERI-RISC-V, ...