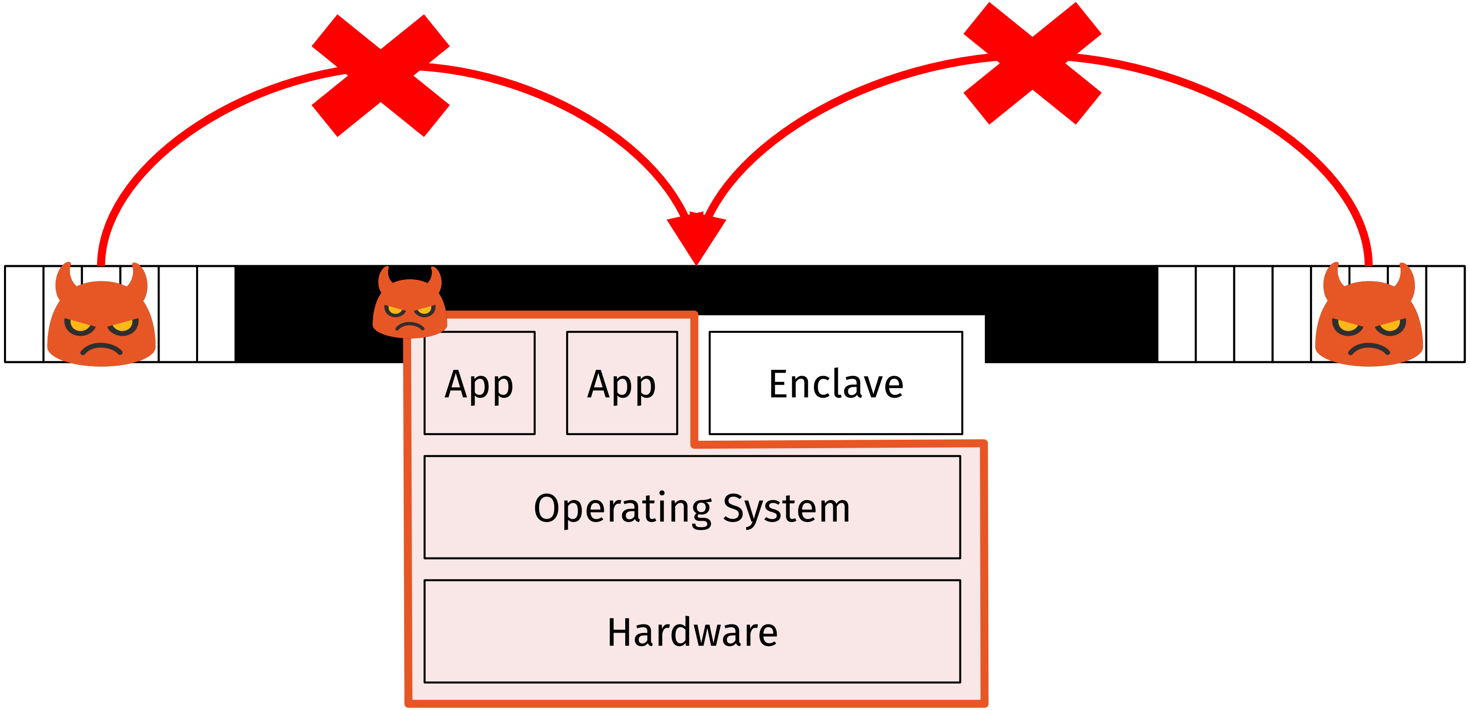
# Reasoning about enclaved execution and attestation in Cerise

Thomas Van Strydonck Dominique Devriese

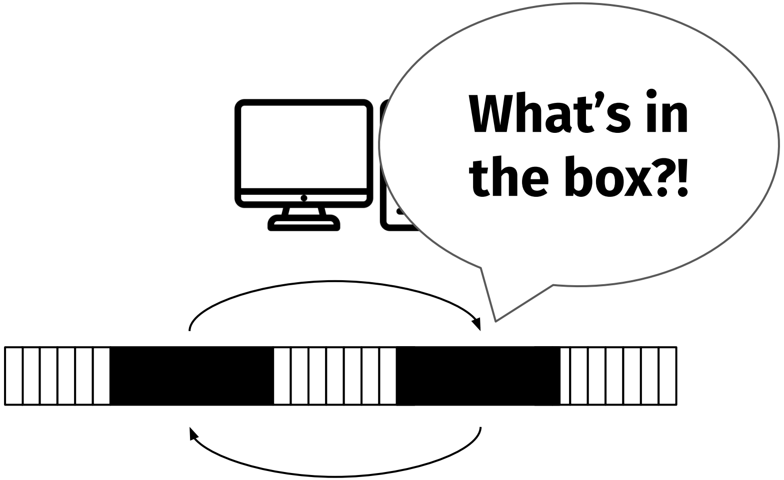
KU Leuven

# Enclaves are like black boxes in memory

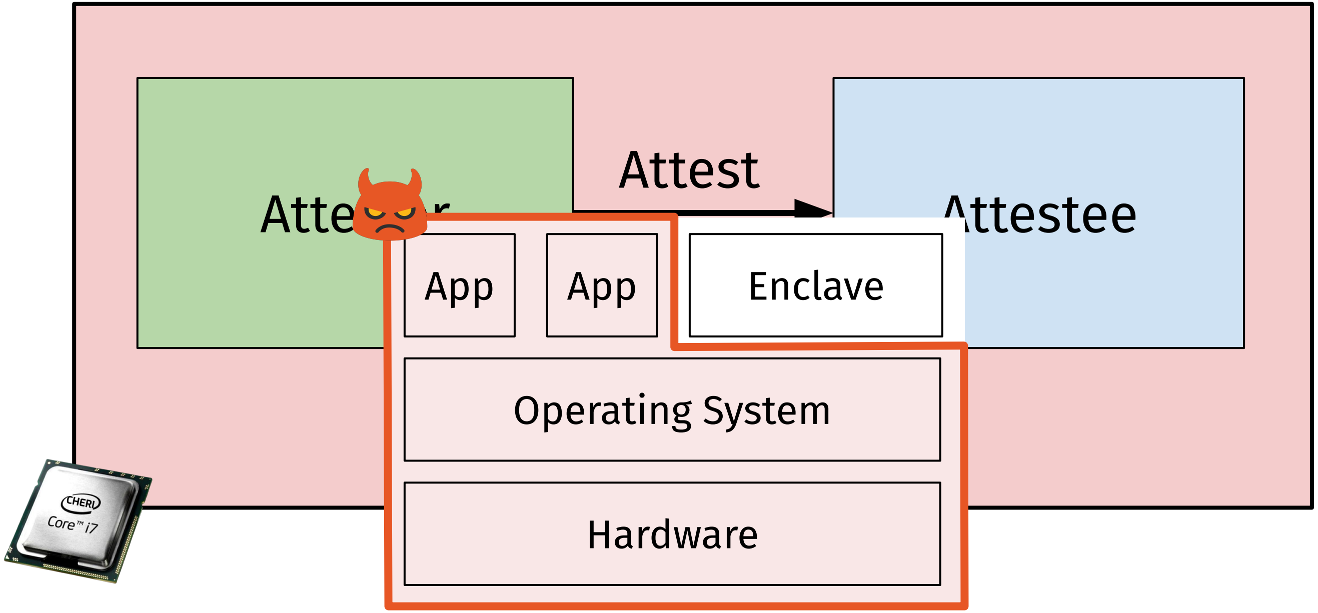


# Attestation: authentication makes enclaves useful

# Attestation: authentication makes enclaves useful



# How do we **formalize** the security guarantees obtained from attestation?



Atte r

Attest

Attestee

App

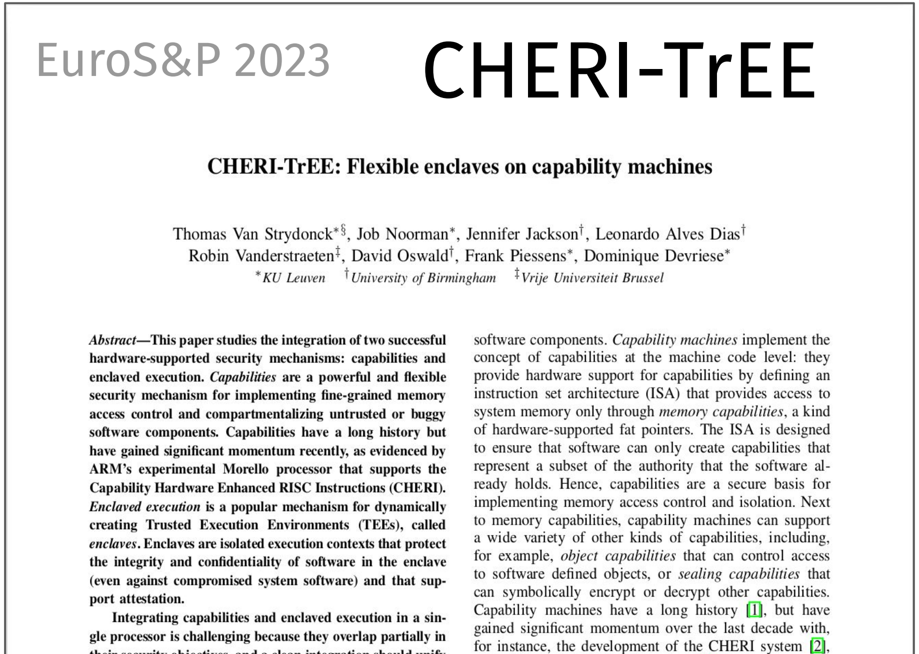
App

Enclave

Operating System

Hardware

# Concretely: **flexible** enclaved execution system



EuroS&P 2023

CHERI-TrEE

### CHERI-TrEE: Flexible enclaves on capability machines

Thomas Van Strydonck*§, Job Noorman*, Jennifer Jackson†, Leonardo Alves Dias†
Robin Vanderstraeten‡, David Oswald†, Frank Piessens*, Dominique Devriese*
*KU Leuven †University of Birmingham ‡Vrije Universiteit Brussel

***Abstract*—This paper studies the integration of two successful hardware-supported security mechanisms: capabilities and enclaved execution. *Capabilities* are a powerful and flexible security mechanism for implementing fine-grained memory access control and compartmentalizing untrusted or buggy software components. Capabilities have a long history but have gained significant momentum recently, as evidenced by ARM's experimental Morello processor that supports the Capability Hardware Enhanced RISC Instructions (CHERI). *Enclaved execution* is a popular mechanism for dynamically creating Trusted Execution Environments (TEEs), called *enclaves*. Enclaves are isolated execution contexts that protect the integrity and confidentiality of software in the enclave (even against compromised system software) and that support attestation.**

**Integrating capabilities and enclaved execution in a single processor is challenging because they overlap partially in**

software components. *Capability machines* implement the concept of capabilities at the machine code level: they provide hardware support for capabilities by defining an instruction set architecture (ISA) that provides access to system memory only through *memory capabilities*, a kind of hardware-supported fat pointers. The ISA is designed to ensure that software can only create capabilities that represent a subset of the authority that the software already holds. Hence, capabilities are a secure basis for implementing memory access control and isolation. Next to memory capabilities, capability machines can support a wide variety of other kinds of capabilities, including, for example, *object capabilities* that can control access to software defined objects, or *sealing capabilities* that can symbolically encrypt or decrypt other capabilities. Capability machines have a long history [1], but have gained significant momentum over the last decade with, for instance, the development of the CHERI system [2],

**Hardware capabilities**

# **How** do we formalize the security guarantees obtained from ***local*** attestation?

Cerise

## Reasoning About a Machine with Local Capabilities

### Provably Safe Stack and Return Pointer Management

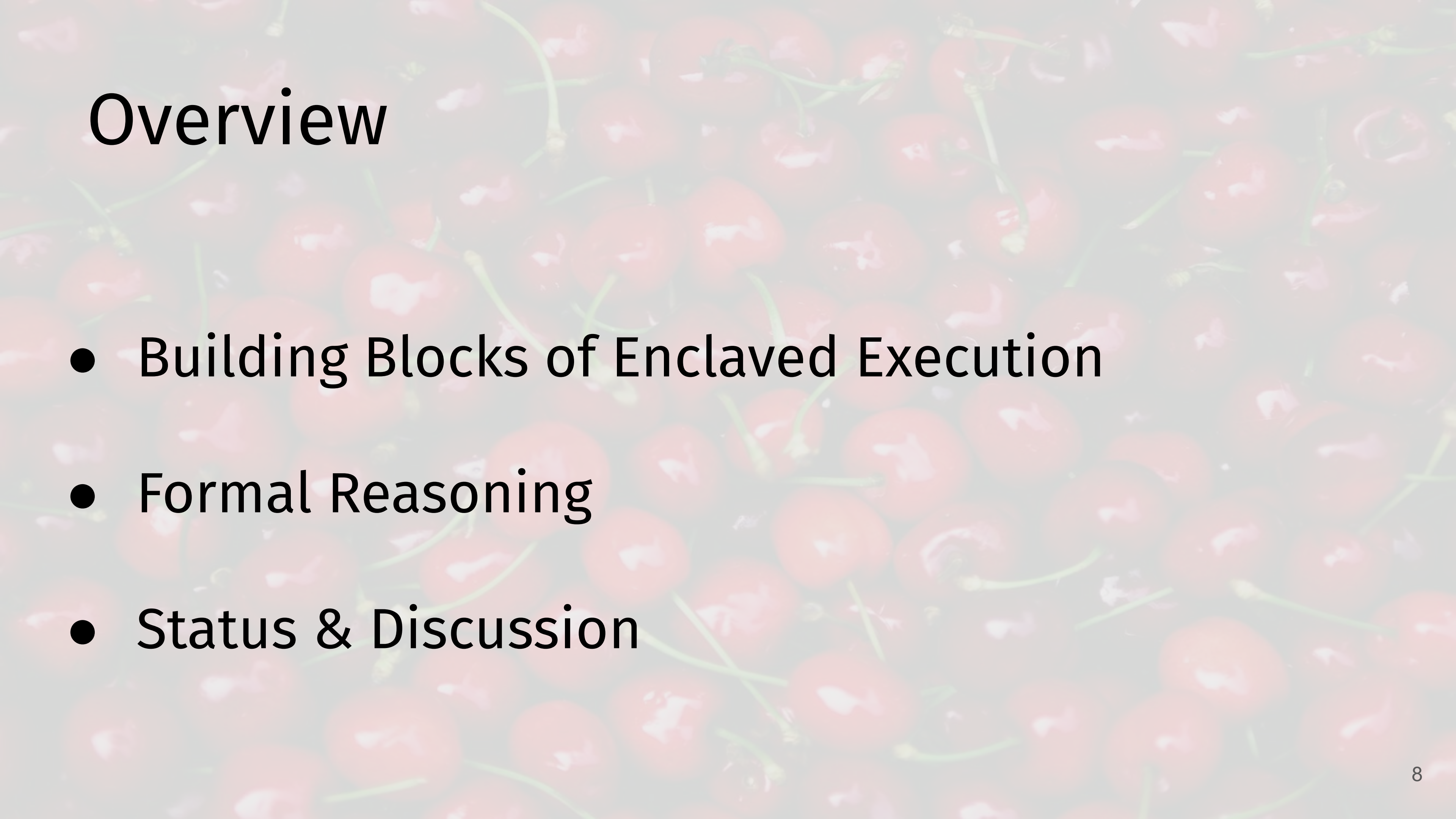
## Efficient and Provable Local Capability Revocation using Uninitialized Capabilities

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
SANDER HUYGHEBAERT, Vrije Universiteit Brussel, Belgium
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

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.

nts: • **Security and privacy → Logic and verification**; **Formal security models**; • **Theory of** → **Program verification**; **Program specifications**.



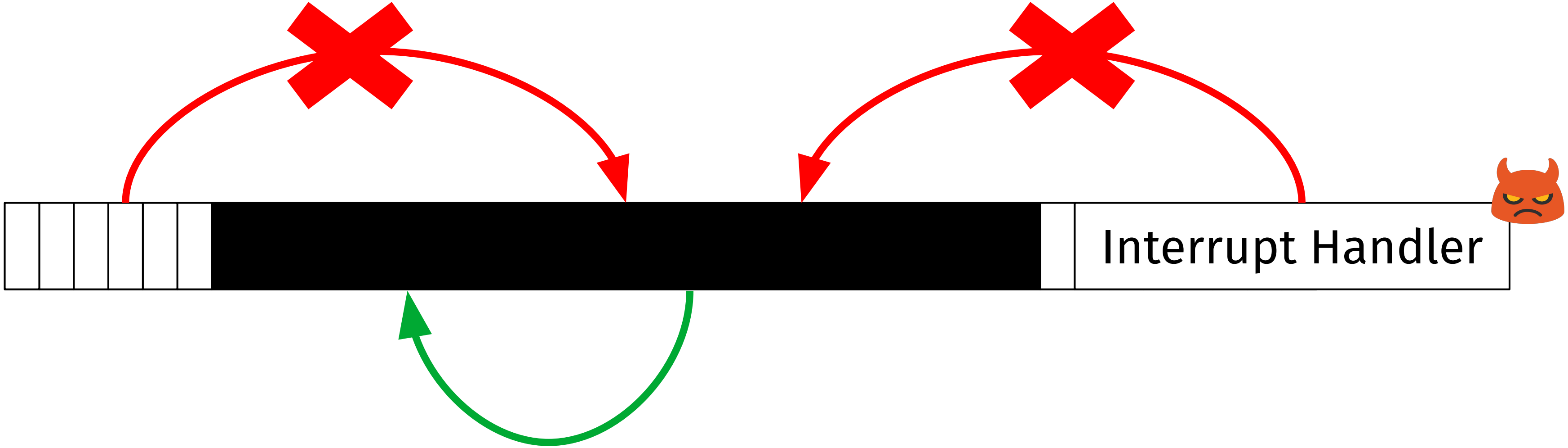
# Overview

- Building Blocks of Enclaved Execution
- Formal Reasoning
- Status & Discussion

# Overview

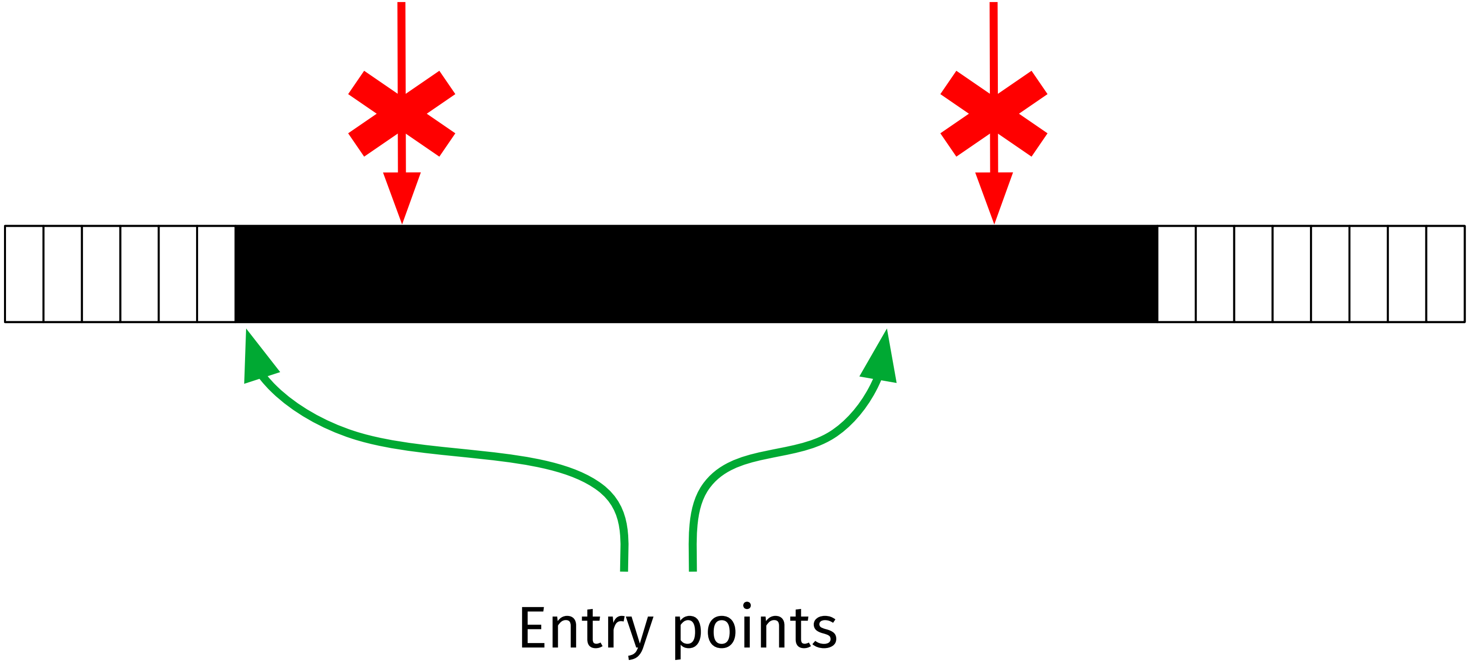
- **Building Blocks of Enclaved Execution**
- Formal Reasoning
- Status & Discussion

# 1. Exclusive access

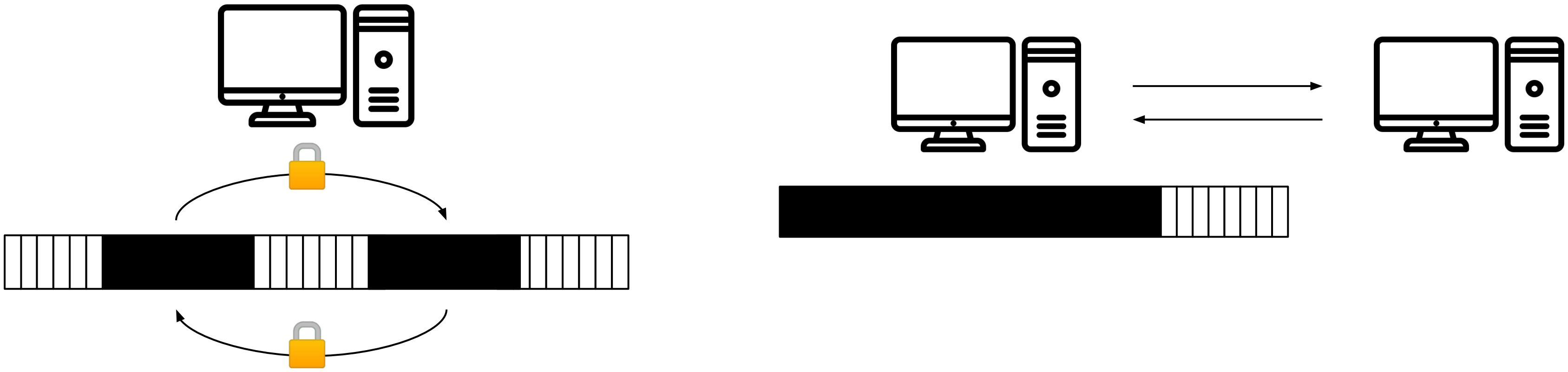


At least during initialization

# 2. Controlled invocation

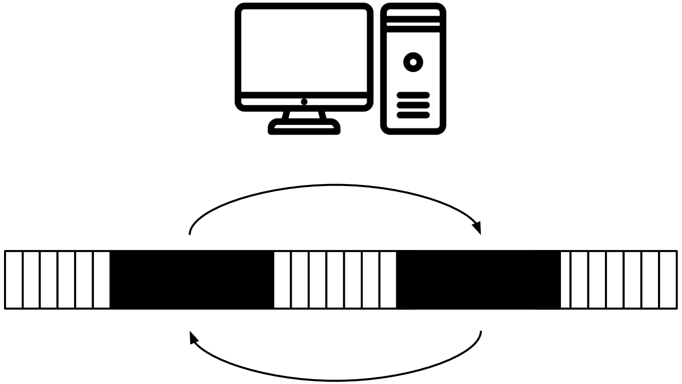


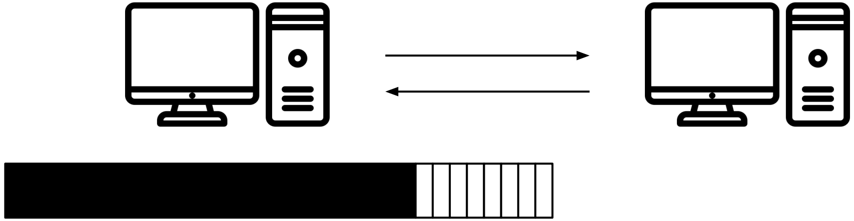
# 3. Secure communication



Efficient

# 4. Attestation



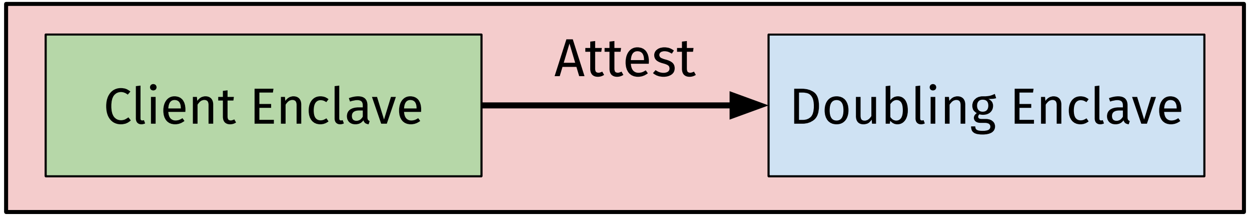


- Authentication: identity → Hash
- Secure look-up (remote: reuse)

# Overview

- Building Blocks of Enclaved Execution
- **Formal Reasoning [WIP]**
- Status & Discussion

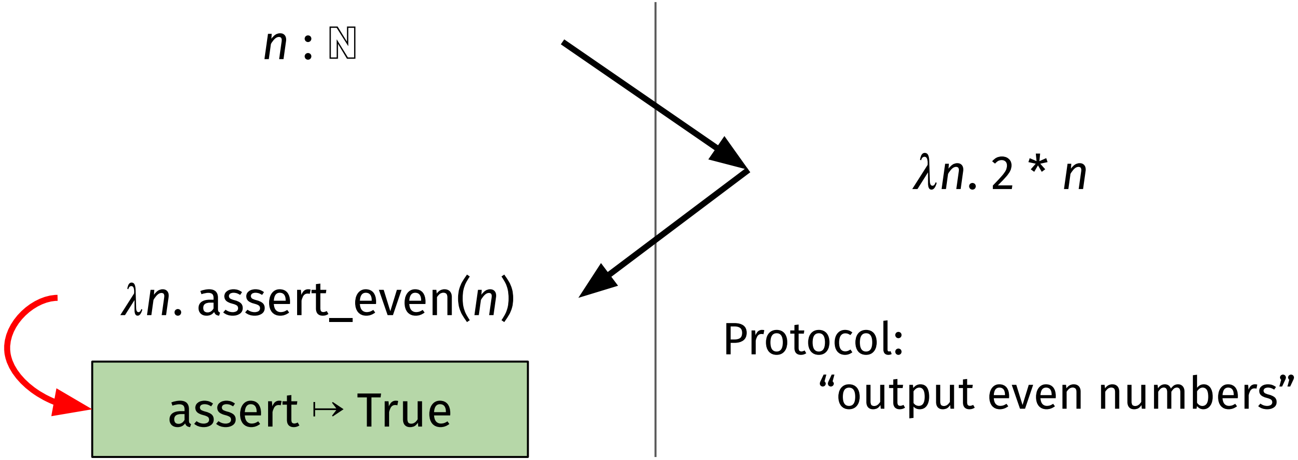
# Running example



Client Enclave

Attest

Doubling Enclave



$n : \mathbb{N}$

$\lambda n.\ 2 * n$

$\lambda n.\ \text{assert\_even}(n)$

assert ↦ True

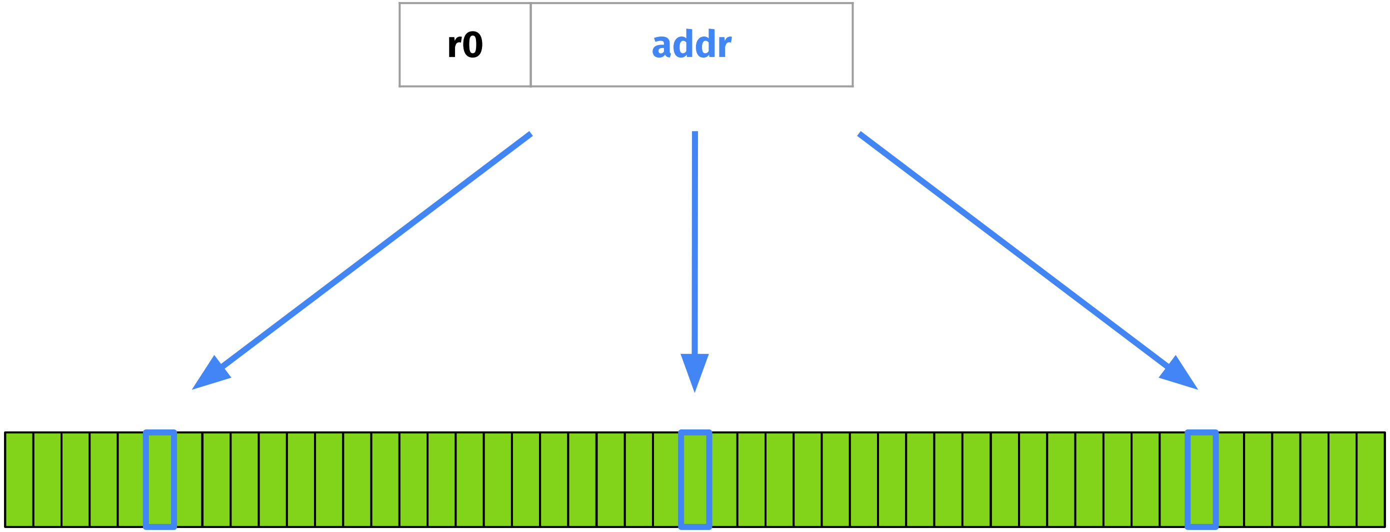
Protocol:
“output even numbers”

# Building blocks of enclaved execution

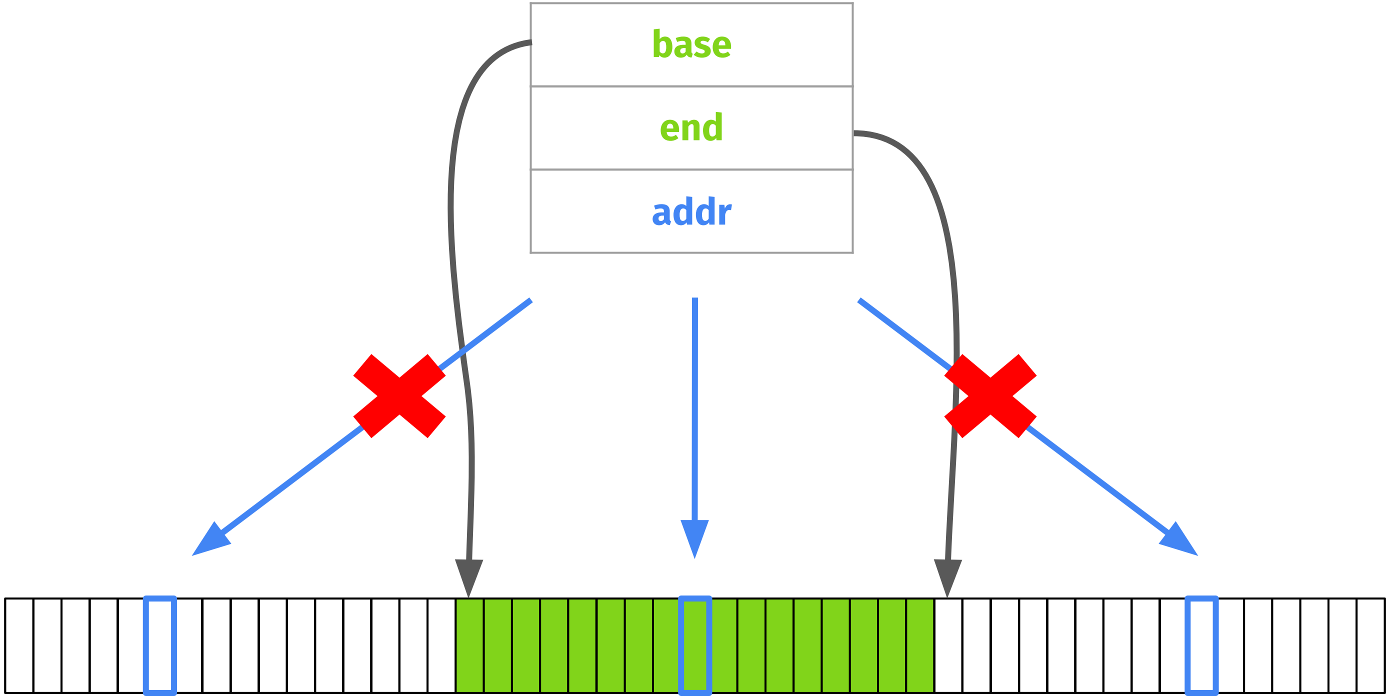
1. Exclusive access
2. Controlled invocation
3. Local secure communication
4. Local attestation

Capabilities → Cerise

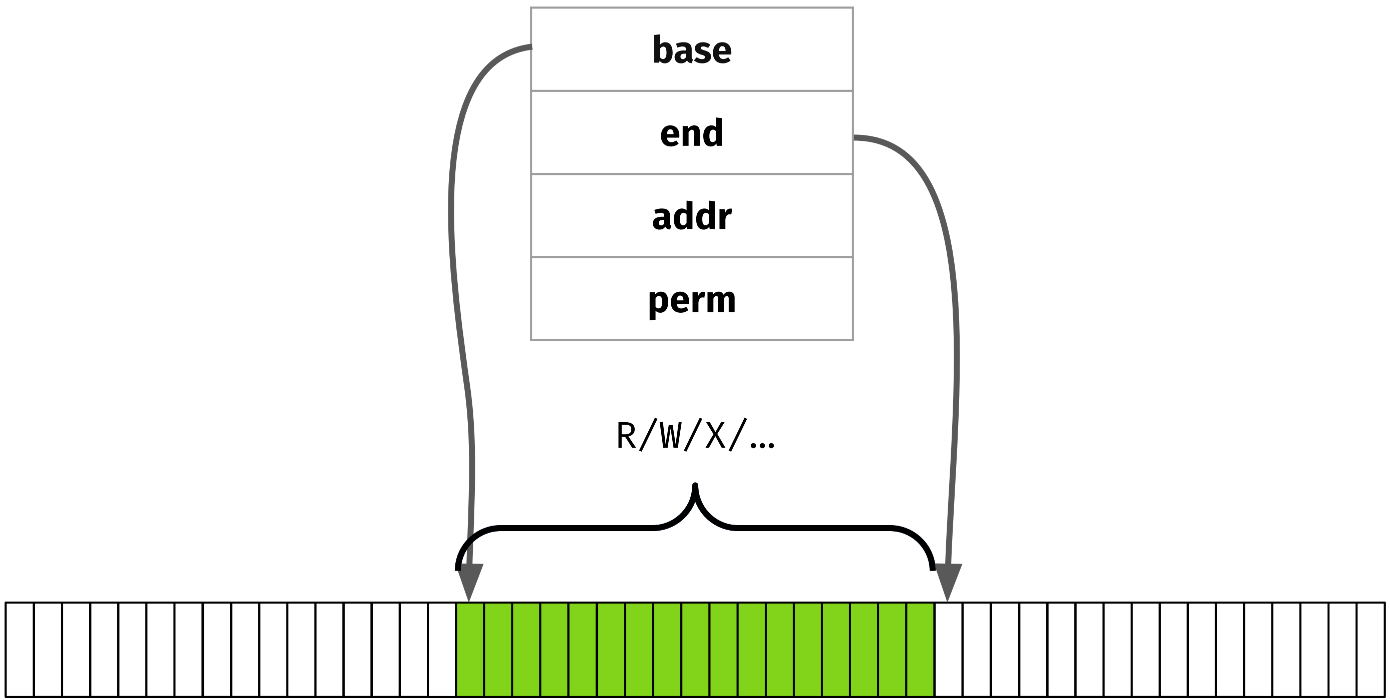
# Issue: pointers grant unrestricted access to memory



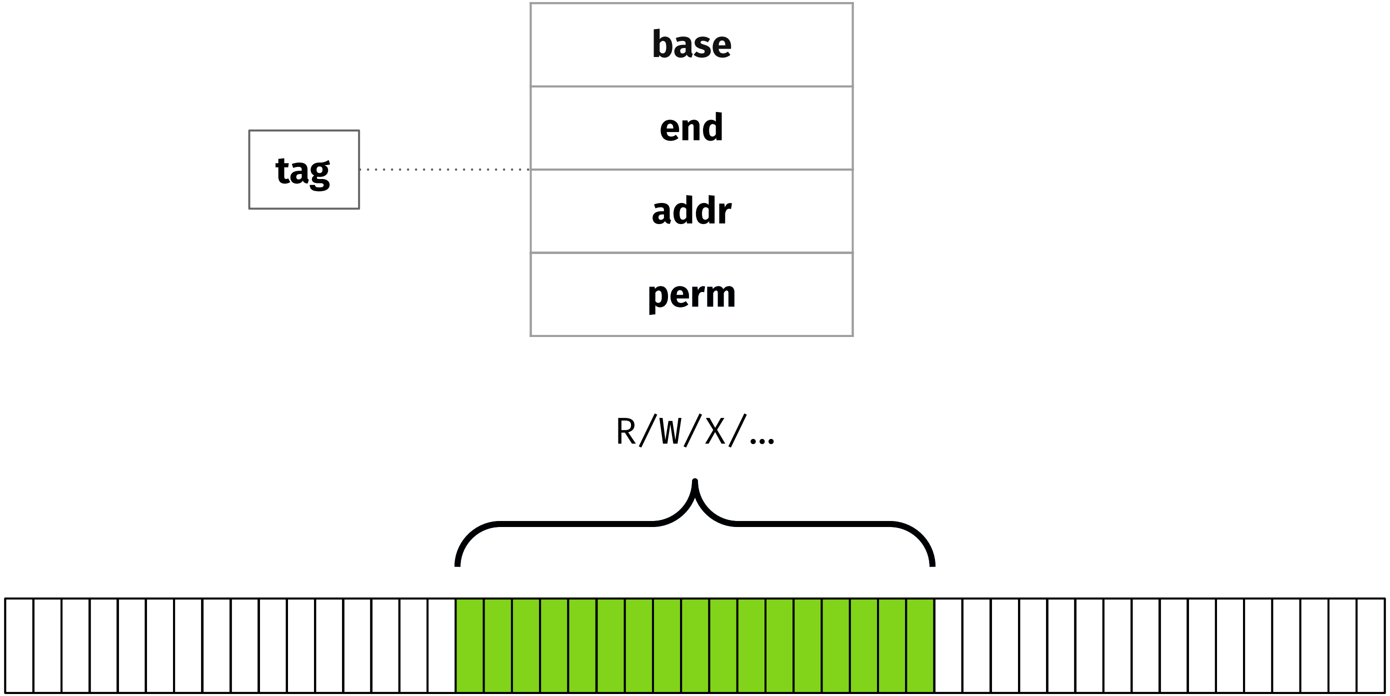
# Hardware capabilities restrict authority



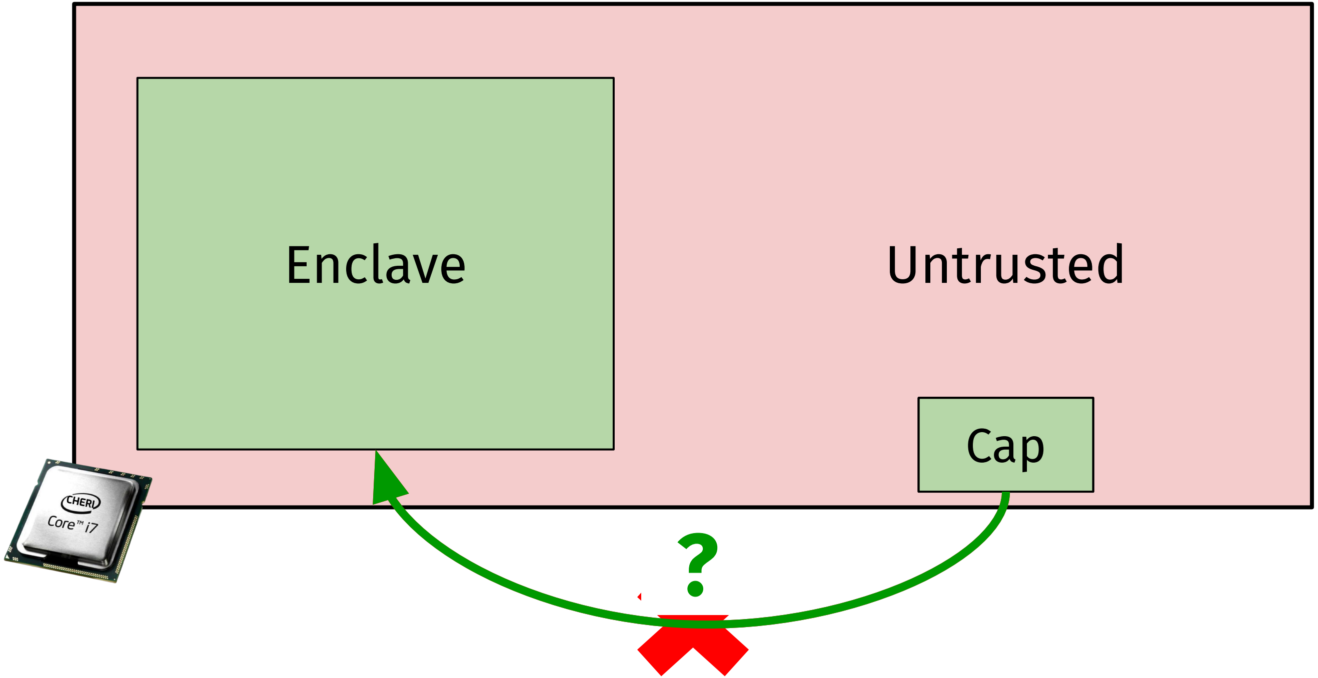
# Hardware capabilities restrict authority



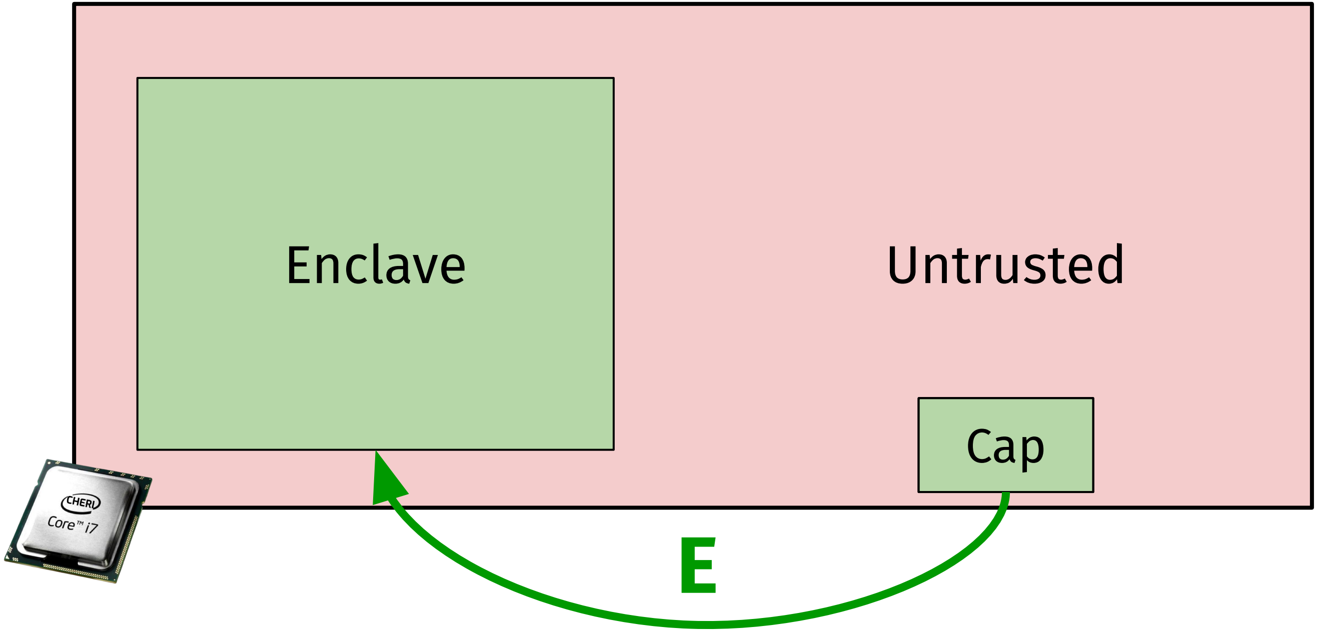
# Hardware capabilities are unforgeable



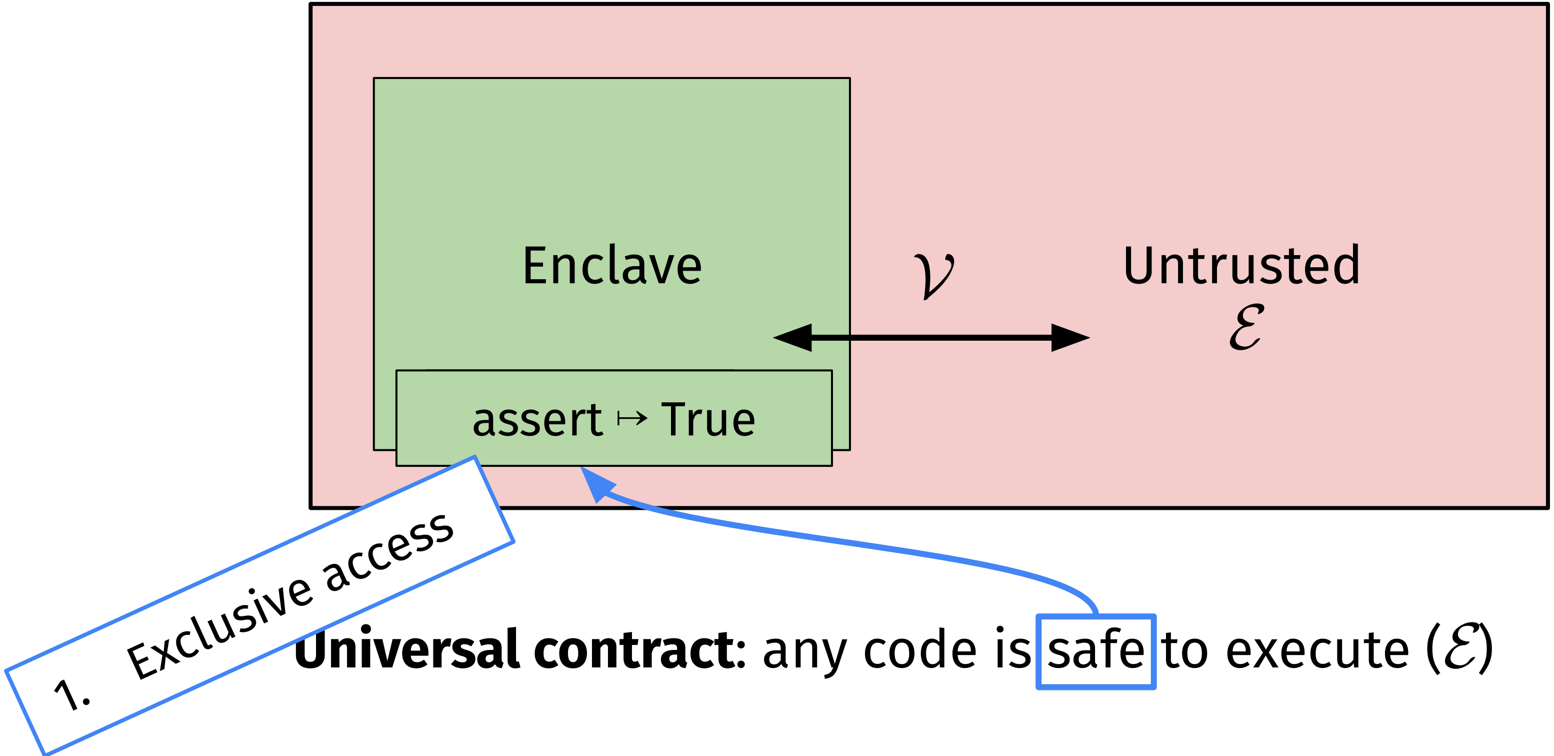
# Spatial memory safety: protection against adversary



# Enter capabilities implement compartmentalization



# Cerise: logical relations to reason about untrusted code



**Universal contract**: any code is safe to execute ($\mathcal{E}$)

# Cerise: Logical relations to reason about untrusted code

$$\mathcal{V}(w) \quad \begin{cases} \mathcal{V}(z) & \triangleq \\ \mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \\ \mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \\ \mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \end{cases}$$

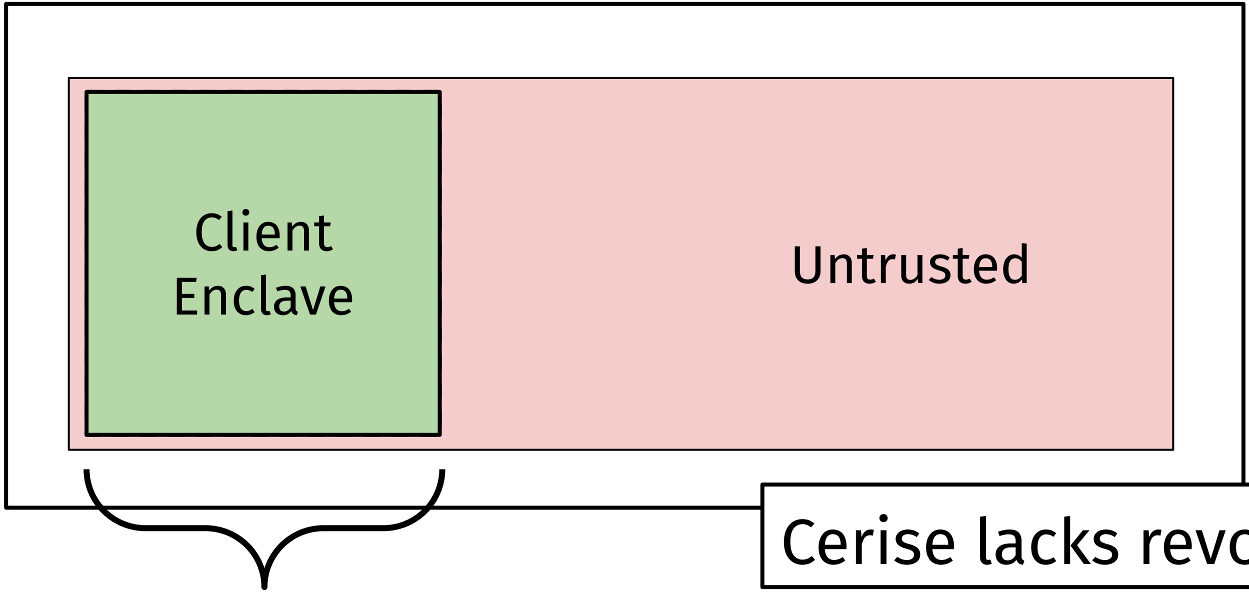
2. Controlled Invocation

# Building blocks of enclaved execution

1. Exclusive access
2. Controlled invocation
3. Local secure communication
4. Local attestation

Capabilities → Cerise

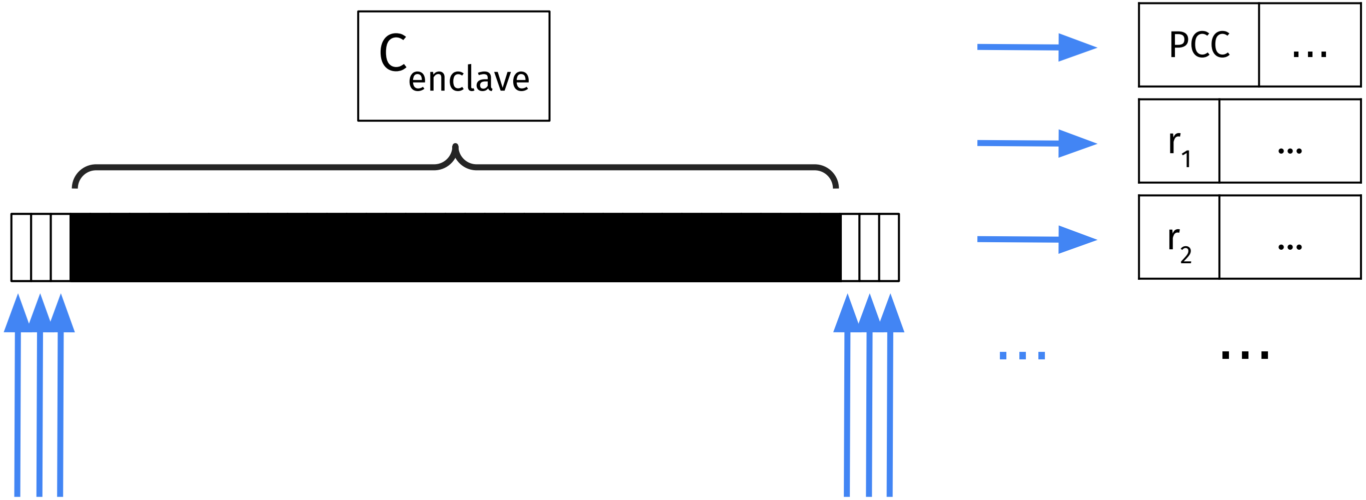
# 1. *Establishing* exclusive access



Cerise lacks revocation...

$$\mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) \triangleq \mathop{\ast}_{a\in[b,e)} \boxed{\exists w, \boxed{(a \mapsto w)} * \mathcal{V}(w)}$$

# *Establishing* exclusive access: memory sweep



# Logical layer to reason about memory sweep

Logical Memory: LAddr ⇀ LWord
Logical Registers: Reg → LWord

"Memory reachable from Regs and LRegs is isomorphic"

Physical Memory: Addr → Word
Physical Registers: Reg → Word

2011 26th Annual IEEE Symposium on Logic in Computer Science

**Separation Logic in the Presence of Garbage Collection**

Chung-Kil Hur Derek Dreyer Viktor Vafeiadis

*Max Planck Institute for Software Systems (MPI-SWS)*
*Kaiserslautern and Saarbrücken, Germany*
*E-mail: {gil, dreyer, viktor}@mpi-sws.org*

# Logical layer to reason about memory sweep

Logical Memory: LAddr ⇀ LWord
Logical Registers: Reg → LWord

“Memory reachable from Regs and LRegs is isomorphic”

$$\mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -, v) \triangleq \mathop{\ast}_{a\in[b,e)} \boxed{\exists w,\ \overline{\underline{(a, v \mapsto_{log}}}\ w) * \mathcal{V}(w)}$$

Sweep succeeds? Mint $a, v' \mapsto_{log} w$ + rewrite in LRegs

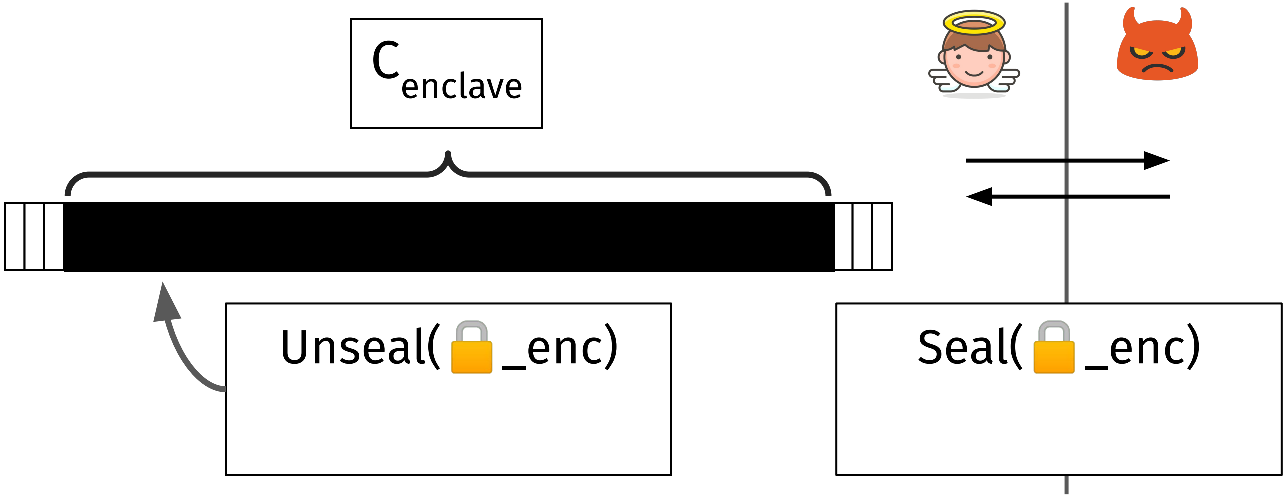
# Building blocks of enclaved execution

1. Exclusive access
2. Controlled invocation
3. Local secure communication
4. Local attestation

Sealed capabilities

# Sealed capabilities implement symbolic crypto

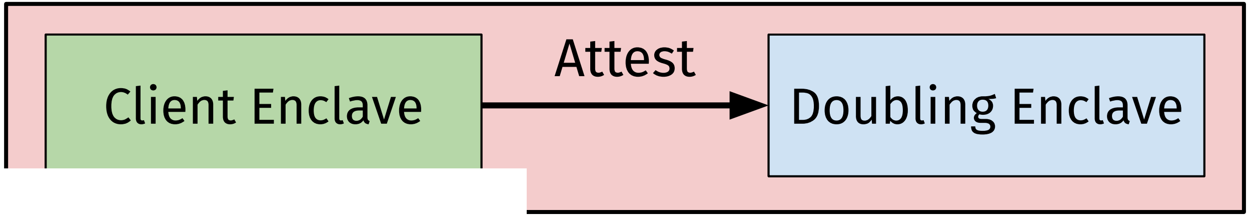
- Encrypt 🔒_enc
- Sign 🔒_sign

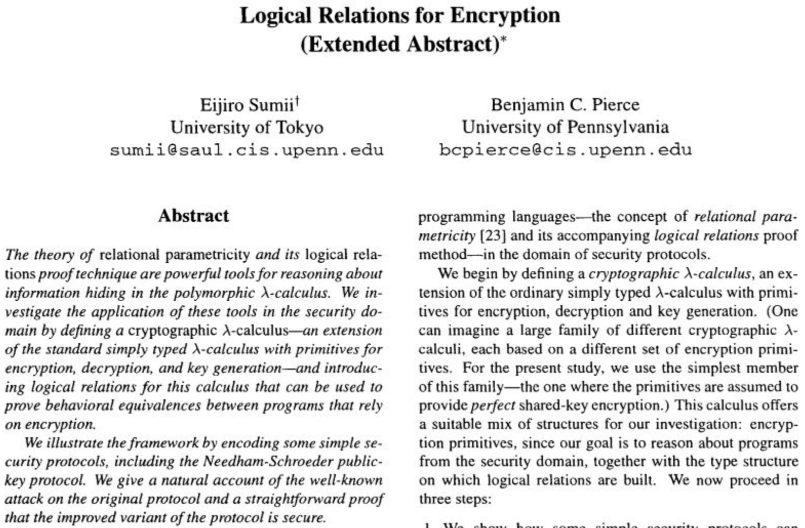


Unseal(🔒_enc)

Seal(🔒_enc)

# Attach a *protocol* to each sealing key





## Logical Relations for Encryption
## (Extended Abstract)*

Eijiro Sumii[†]
University of Tokyo
sumii@saul.cis.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

### Abstract

*The theory of* relational parametricity *and its* logical relations *proof technique are powerful tools for reasoning about information hiding in the polymorphic λ-calculus. We investigate the application of these tools in the security domain by defining a* cryptographic λ-calculus*—an extension of the standard simply typed λ-calculus with primitives for encryption, decryption, and key generation—and introducing logical relations for this calculus that can be used to prove behavioral equivalences between programs that rely on encryption.*

*We illustrate the framework by encoding some simple security protocols, including the Needham-Schroeder public-key protocol. We give a natural account of the well-known attack on the original protocol and a straightforward proof that the improved variant of the protocol is secure.*

programming languages—the concept of *relational parametricity* [23] and its accompanying *logical relations* proof method—in the domain of security protocols.

We begin by defining a *cryptographic λ-calculus*, an extension of the ordinary simply typed λ-calculus with primitives for encryption, decryption and key generation. (One can imagine a large family of different cryptographic λ-calculi, each based on a different set of encryption primitives. For the present study, we use the simplest member of this family—the one where the primitives are assumed to provide *perfect* shared-key encryption.) This calculus offers a suitable mix of structures for our investigation: encryption primitives, since our goal is to reason about programs from the security domain, together with the type structure on which logical relations are built. We now proceed in three steps:

$\lambda n.\ 2 * n$

Signing

Pr

$P_{sign}(c : \mathrm{Cap}) = \boxed{\exists\ n.\ c \mapsto 2 * n}$

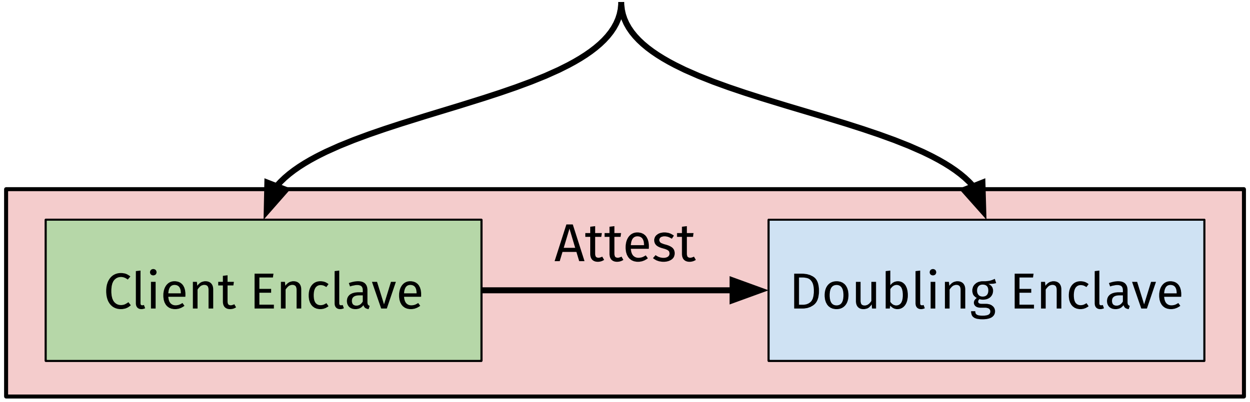
# Resource algebra for protocols

$$\mathrm{CanAlloc}\ k \qquad \Rrightarrow \qquad \mathrm{Pred}\ k\ P$$

$$\mathrm{Pred}\ k\ P \twoheadrightarrow \mathrm{Pred}\ k\ P' \twoheadrightarrow (\forall\ x.\ \triangleright (P\ x \equiv P'\ x))$$

# Reasoning about sealed capabilities

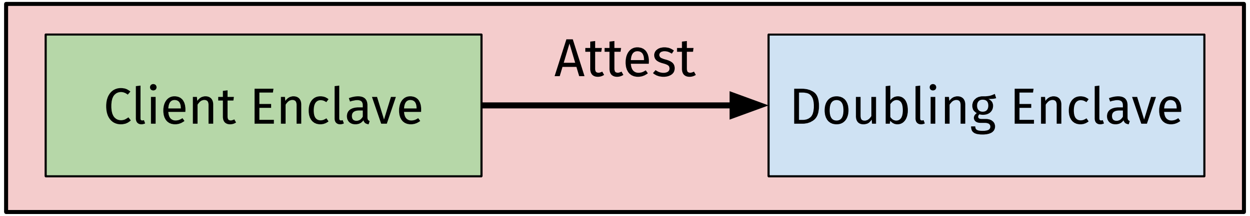
$$\mathcal{V}(\text{sealed}(w,k)) =$$
$$(\exists\ P.\ \ \text{Persistent}\ P * \text{Pred}\ k\ P * \triangleright P\ w)$$


Client Enclave

Attest

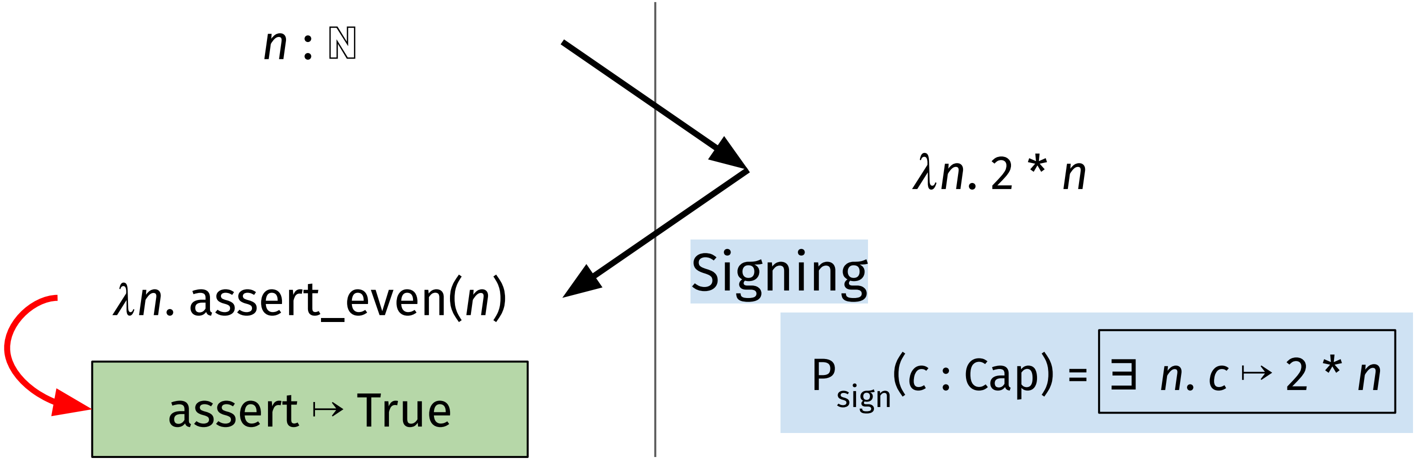
Doubling Enclave

# How does client *know* that $P_{sign}$ is used?


Client Enclave

Attest

Doubling Enclave


$n : \mathbb{N}$

$\lambda n.\ 2 * n$

Signing

$P_{sign}(c : \mathrm{Cap}) = \boxed{\exists\ n.\ c \mapsto 2 * n}$

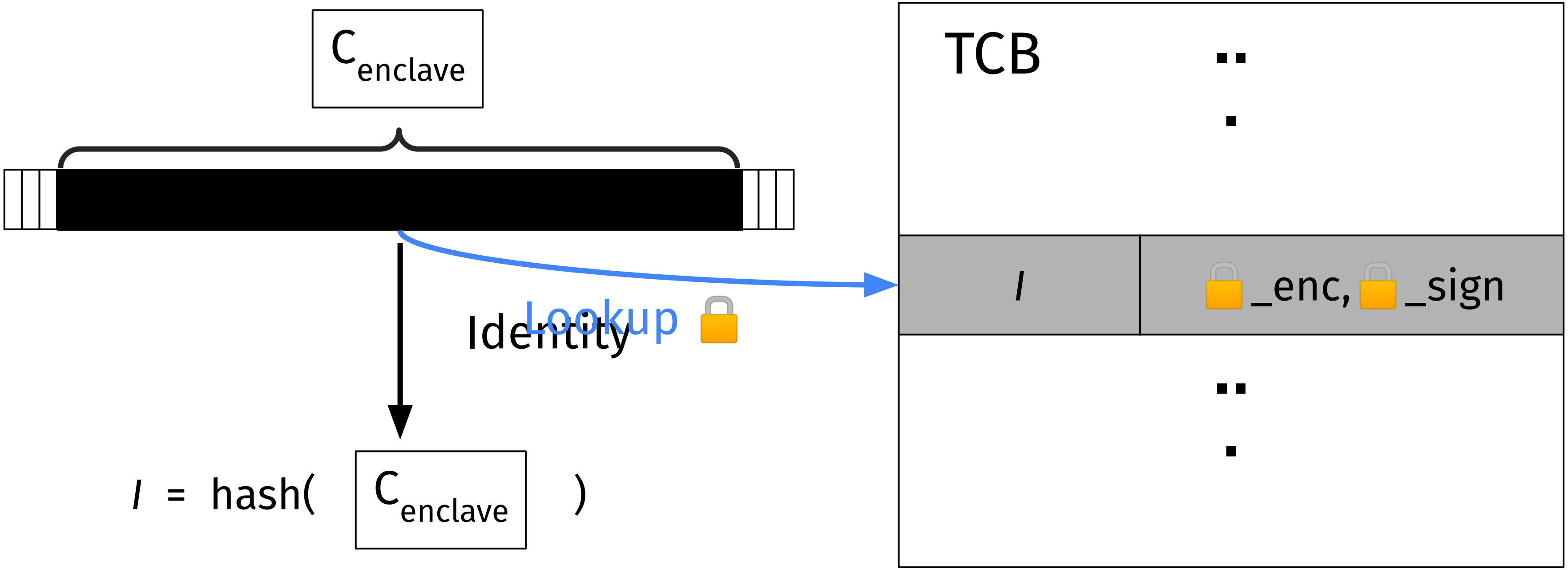
$\lambda n.\ \mathrm{assert\_even}(n)$

assert ↦ True

# Building blocks of enclaved execution

1. Exclusive access
2. Controlled invocation
3. Local secure communication
4. Local attestation

# Operational aspects of attestation



$C_{enclave}$

Identity

Lookup 🔒

$I$ = hash( $C_{enclave}$ )

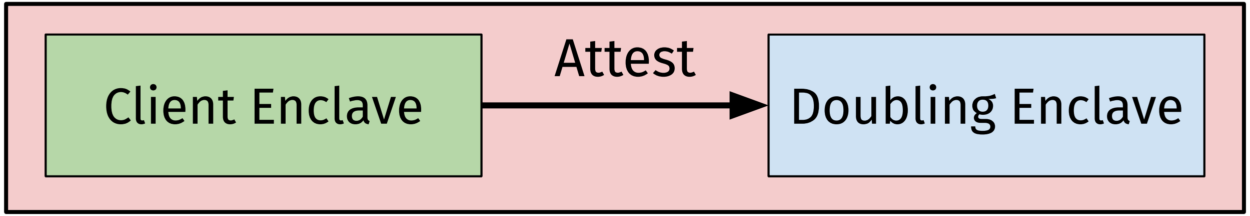
TCB

⋮

| $I$ | 🔒_enc, 🔒_sign |
|---|---|

⋮

# Reasoning about attestation: establish $P_{sign}$



Identity(DE) = I *Collision-free

Lookup(ksign) = I

Pred(ksign,$P_{sign}$)

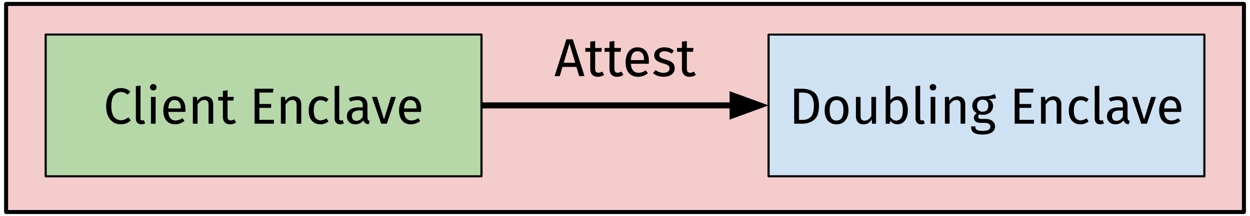
$\mathcal{V}$(sealed($w$,ksign)) =
(∃ $P$. Persistent $P * $ Pred ksign $P * \triangleright P\ w$)

custom_Psign :
Identity $\rightharpoonup$ (Word → iProp)

custom_Psign(I) = $P_{sign}$

(...) ∀ *enclaves*. (Lookup($k$) = $I$ ∗
custom_Psign($I$) = $P$) -∗ Pred($k$,$P$)

# Reasoning about attestation: initialize enclave

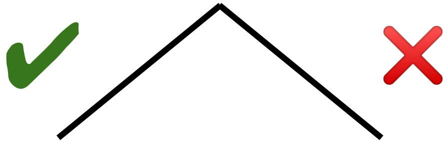


Client Enclave

Attest

Doubling Enclave

$I_{\text{new}} \in$ dom(custom_Psign)



$P_{\text{sign}}$ = custom_Psign($I$)

$P_{\text{sign}} = \mathcal{V}$

custom_Psign :
    Identity ⇀ (Word → iProp)

custom_Psign(I) = $P_{\text{sign}}$

(…) ∀ *enclaves*. (Lookup($k$) = $I$ ∗
custom_Psign($I$) = $P$) -∗ Pred($k$,$P$)

# Overview

- Building Blocks of Enclaved Execution
- Formal Reasoning
- **Status & Discussion**

# Status of reasoning about enclaved execution

| | |
|---|---|
| 1. Exclusive access | WIP (Memory Sweep) |
| 2. Controlled invocation | ✓ |
| 3. Local attestation | WIP |
| 4. Local secure communication | ✓ |

# Future work

- ←←Examples→→
  - Multiple enclaves/protocols, caller attestation
  - Binary Cerise: confidentiality
  - Verify interrupts
- Remote case (~~Asserts~~)
  - Probabilistic model?
  - Hash function
- Generalize?
  - Keystone
  - Sancus