



Cerisier: A Program Logic for Attestation in a Capability Machine

JUNE ROUSSEAU, Aarhus University, Denmark

DENIS CARNIER, KU Leuven, Belgium

THOMAS VAN STRYDONCK, Fortanix, Netherlands

STEVEN KEUCHEL, KU Leuven, Belgium

DOMINIQUE DEVRIESE, KU Leuven, Belgium

LARS BIRKEDAL, Aarhus University, Denmark

A key feature in trusted computing is attestation, which allows encapsulated components (enclaves) to prove their identity to (local or remote) distrusting components. Reasoning about software that uses the technique requires tracking how trust evolves after successful attestation. This process is security-critical and non-trivial, but no existing formal verification technique supports modular reasoning about attestation of enclaves and their clients, or proving end-to-end properties for systems combining trusted, untrusted and attested code.

We contribute Cerisier, the first program logic for modular reasoning about trusted, untrusted and attested code, fully mechanized in the Iris separation logic and the Rocq Prover. We formalize a recent proposal, CHERI-TrEE, to extend capability machines with enclave primitives, as an extension to the Cerise capability machine and program logic. Our program logic comes with a universal contract for untrusted code, which captures both capability safety and local enclave attestation. Like Cerise, this universal contract is phrased in terms of a logical relation defining capabilities' authority. We demonstrate Cerisier by proving end-to-end properties for three representative applications of trusted computing: secure outsourced computation, mutual attestation and a modeled trusted sensor component.

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Formal security models**; • **Theory of computation** → **Logic and verification**; **Program verification**.

Additional Key Words and Phrases: capability machines, enclaves, attestation, logical relation, universal contracts, Iris, separation logic

ACM Reference Format:

June Rousseau, Denis Carnier, Thomas Van Strydonck, Steven Keuchel, Dominique Devriese, and Lars Birkedal. 2026. Cerisier: A Program Logic for Attestation in a Capability Machine. *Proc. ACM Program. Lang.* 10, PLDI, Article 209 (June 2026), 25 pages. <https://doi.org/10.1145/3808287>

1 Introduction

Trusted Computing refers to techniques for allowing software components to establish trust in one another [28]. The technology has applications in digital rights management, anti-cheat measures in gaming, private contact discovery, etc. A simple representative application is client code outsourcing computation to untrusted code, for example, from a kernel client to a local user process to reduce available privilege, or from a resource-constrained device to a large but untrusted cloud server.

Authors' Contact Information: [June Rousseau](mailto:june.rousseau@cs.au.dk), Aarhus University, Aarhus, Denmark, june.rousseau@cs.au.dk; [Denis Carnier](mailto:denis.carnier@kuleuven.be), KU Leuven, Leuven, Belgium, denis.carnier@kuleuven.be; [Thomas Van Strydonck](mailto:thomas.vanstrydonck@fortanix.com), Fortanix, Eindhoven, Netherlands, thomas.vanstrydonck@fortanix.com; [Steven Keuchel](mailto:steven.keuchel@kuleuven.be), KU Leuven, Leuven, Belgium, steven.keuchel@kuleuven.be; [Dominique Devriese](mailto:dominique.devriese@kuleuven.be), KU Leuven, Leuven, Belgium, dominique.devriese@kuleuven.be; [Lars Birkedal](mailto:lars.birkedal@cs.au.dk), Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART209

<https://doi.org/10.1145/3808287>

In this paper, we are interested in systems featuring *attestation* and a *dynamic root of trust*, where a component C (an *enclave*) can ask the system for an attestation certificate. This certificate attests to C 's identity, which might consist of a hash of C 's source code, information about its location in memory and the system it is running on. The certificate can be used to prove C 's identity to other components, assuming that they trust the system's Trusted Computing Base (TCB), which typically consists only of hardware and firmware, but not the operating system or application software. The certificate can be securely communicated to other components on the same system (local attestation) or cryptographically signed using a system-embedded cryptographic key for transmission to remote components (remote attestation).

Attestation involves a complex interplay and non-trivial trust relationship between trusted components, enclave components (initially untrusted, but become trusted after successful attestation), and surrounding untrusted code and trusted hardware/firmware. It is typically applied in security-critical scenarios and trusted components are kept small for security reasons. These characteristics (*small* but *complex* and *critical*) suggest the field as a prime target for rigorous software verification. Until now, verification has remained restricted to only one of the involved components (for example, the crypto used by the trusted hardware/firmware [37, 47]), which does not yield end-to-end properties of the combined system.

In this paper, we present Ceriser: the first rigorous program logic for modular reasoning about trusted, untrusted and attested code, and producing end-to-end guarantees about the combined system. We base ourselves on the CHERI-TrEE [43] trusted computing primitives for the CHERI-RISC-V capability machine [44, §4], and on Cerise [18], a program logic implemented in Iris [26] for a capability machine modeled loosely after CHERI-RISC-V.

Capability machine instruction set architectures (ISAs) offer a primitive notion of capabilities that represent the authority to perform certain actions, e.g., read-write access to a certain memory range, or the permission to invoke a piece of code with access to private data and capabilities, without exposing that private data to the caller.

CHERI-TrEE aims to maximally reuse existing CHERI-RISC-V primitives and only adds a minimal set of new ones to support trusted computing. Specifically, it reuses so-called sealed capability pairs for encapsulating components, and sealed capabilities for secure communication. It adds an *exclusive memory ownership primitive* for components to verify their exclusive access to a memory region. The CPU guarantees this exclusive access based on exhaustively scanning the memory for overlapping capabilities. This happens at enclave initialization time, or at the request of a component to dynamically extend its memory footprint. This essentially enables what is known as a dynamic root-of-trust: a component can start out as untrusted code, but at some point ask the CPU to initialize itself as an enclave. CHERI-TrEE also adds a *primitive for local attestation*, through which a component can ask the CPU to vouch for its identity. Concretely, the CPU will provide the component with certain private capabilities that can be verifiably connected to the component's identity by distrusting local software.

Ceriser extends the Cerise program logic with support for CHERI-TrEE: adaptations of existing techniques for supporting sealed capabilities (based on program logics for symbolic cryptography [41]) and the exclusive memory ownership primitive (based on a program logic for garbage collection [23]).¹ Additionally, Ceriser adds a novel technique for reasoning about CHERI-TrEE's attestation feature by extending Cerise's universal contract. This universal contract provides guarantees about untrusted code [19, 24], formalized in terms of a logical relation. Specifically, before using the universal contract, one can register predicates for a number of enclaves of interest, hereafter *enclave sealing predicates*. After verifying that the registered enclaves do indeed respect their

¹We do not support sealed pairs for encapsulation, since Cerise already offers an alternative: sentry capabilities.

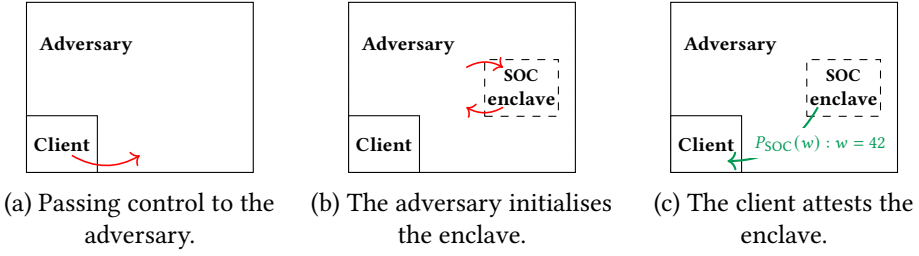


Fig. 1. Interaction scenario for the secure outsourced computation (SOC) example.

registered enclave sealing predicates, the universal contract then ensures that capabilities produced by untrusted code will respect attested enclaves' enclave sealing predicates. The resulting reasoning pattern is unusual: for completely arbitrary code, we get quite specific guarantees about the capabilities they can produce, namely that those capabilities can only be correctly attested if they satisfy the registered enclave sealing predicate (we present a concrete simple example in the next section). The contributed technique quite generally captures intuitive reasoning on trusted execution systems and we believe it applies to general trusted computing systems even though our current development is specific to capability machines.

To summarize, in this paper we contribute:

- An operational semantics for a capability machine with attestation primitives (Section 4.1).
- The first program logic with support for modular reasoning about attested enclaves, their clients and untrusted code, including support for sealing, memory revocation and local attestation (Section 4.3).
- A universal contract capturing both capability safety and local enclave attestation, formalized using a novel parameterized logical relation (Sections 4.4 and 4.5).
- Three case studies inspired by the literature, in which we prove non-trivial end-to-end properties of representative applications of trusted computing (Section 5).

All definitions and proofs are mechanized in Rocq [35, 36], and available in the accompanying material [34]. In the rest of the paper, the definitions marked in blue correspond to novel contributions of this work, when it extends preexisting definitions.

2 Trusted Computing Motivating Example: Secure Outsourced Computation

To motivate our technical development, let us take a closer look at a representative application of trusted execution: a simple form of *secure outsourced computation* (SOC) [4, 27]. We assume that a client wants a computation-intensive task to be executed by untrusted code. For ease of explanation, this section considers local attestation of a stateless computation task with public inputs and we hide some technical details behind pseudocode until subsequent sections.

In our example, a trusted client interacts with an untrusted adversary and an enclave (initially untrusted), in a scenario depicted in Figure 1 and implemented by the pseudocode in Figure 2. After the trusted client invokes the adversary (Figure 1a), the adversary initializes the enclave using a trusted execution primitive (Figure 1b). This initialization triggers the encapsulation of the enclave: it gets exclusive access to its own memory and can only be invoked at its entry point. Additionally, the enclave obtains a public/private key pair (k_p, k_s) for attesting messages (for now, imagine a symbolic model of cryptographic signatures). The untrusted adversary then passes control to the enclave (Figure 1b) which will perform the heavy computation and emit a message containing the attested result r^* and the public attestation key k_p .

```

Enclave  $\triangleq$   $(k_p, k_s) \leftarrow \text{GetAttestationKeys}()$       Client  $\triangleq$   $(r^*, k_p) \leftarrow \text{Adversary}()$ 
   $r \leftarrow 42$  // computation...
  return  $(\text{Attest}(r, k_s), k_p)$ 
  if  $\neg \text{VerifyKey}(k_p, \text{ID}_{\text{soc}})$  then abort
  if  $\neg \text{VerifyMessage}(r^*, k_p)$  then abort
Adversary  $\triangleq$   $\text{enclave}^* \leftarrow \text{Initialize}(\text{Enclave})$ 
  assert  $\text{Value}(r^*) = 42$ 
  return  $\text{enclave}()$ 

```

Fig. 2. Simplified pseudocode of Secure Outsourced Computation (SOC) using CHERI-TrEE primitives. The Enclave and Client components contain known code. Under normal circumstances the Adversary executes arbitrary code, here we fix it to capture its intended execution.

After receiving this message from the untrusted adversary acting as an intermediary, the client validates the received message (Figure 1c). Successful validation allows the client to trust that the message originated from a correctly initialized enclave with a certain identity ID_{soc} . This identity is the combination of the enclave’s executable code with details about the system it runs on, summarized using a cryptographic hash. The client can then verify that the corresponding enclave will only attest messages containing correct results, and conclude to trust the received result.

Let us now give a very rough sketch to explain how our approach enables us to obtain an end-to-end guarantee for this example.

First, we must prove that the SOC enclave code respects its enclave sealing predicate. In this case, we’ve picked the predicate that states the enclave will compute 42. Since the enclave’s identity—and hence its code—is known after validation by the client, we can employ standard Hoare-style reasoning using the rules of the logic to complete the proof. Second, we must prove the specification (Hoare triple) for the client program. Again, since the client program code is known, we can proceed using the proof rules of the logic. However, note that the client invokes (and later awaits) the untrusted adversary. In general, we will not know which code the adversary executes², thus, when we arrive at the invocation of the adversary, we make use of a *universal contract*, defined using the program logic, to obtain a Hoare triple for the unknown adversary’s code. To clarify, the universal contract can be viewed as a specification for arbitrary code. In our approach, we parameterize this contract by a list of identities of enclaves and their associated predicates, which the adversary is allowed to initialize. In this particular example, this list only contains the identity of the SOC enclave. We then instantiate the contract with the proof of the SOC enclave to obtain a property for the adversary’s code. Finally, we can prove that the assert of the client holds because the value’s attest must respect the enclave sealing predicate of the SOC enclave.

This section ignores many details and assumptions for ease of explanation. In particular, Figure 2 uses high-level primitives, but in the rest of the paper, we use the instructions of the Cerisier machine. In the next sections, we explain some background knowledge before providing more detailed explanations of the Cerisier machine.

3 Background and Technical Foundations

In this section, we recall the Cerise capability machine model and its program logic. Moreover, we discuss the addition of sealed capabilities, the memory revocation primitives, and hashing primitives, which are all used as a foundation for attestation.

²In Figure 2 we have shown concrete example code for the adversary to make the example easier to understand.

$a \in \text{Addr} \triangleq [0, \text{AddrMax}]$ $reg \in \text{Regs} \triangleq \text{RegName} \rightarrow \text{Word}$
 $p \in \text{Perm} ::= o \mid e \mid RO \mid RX \mid RW \mid RWX$ $m \in \text{Mem} \triangleq \text{Addr} \rightarrow \text{Word}$
 $c \in \text{Cap} \triangleq \{(p, b, e, a) \mid b, e, a \in \text{Addr}\}$ $s \in \text{ExecState} ::= \text{Running} \mid \text{Halted} \mid \text{Failed}$
 $w \in \text{Word} \triangleq \mathbb{Z} + \text{Cap}$ $\sigma \in \text{ExecConf} \triangleq \text{Regs} \times \text{Mem}$
 $r \in \text{RegName} ::= pc \mid r_0 \mid r_1 \mid \dots \mid r_{31}$ $\Sigma \in \text{MState} \triangleq \text{ExecState} \times \text{ExecConf}$

$\rho \in \text{InstrArg} \triangleq \mathbb{Z} + \text{RegName}$
 $i \in \text{Instr} ::= \text{jmp } r_d \mid \text{jnz } r_d \ r_{\text{cond}} \mid \text{fail} \mid \text{halt} \mid \text{mov } r_d \ \rho_s \mid \text{add } r_d \ \rho_1 \ \rho_2 \mid$
 $\text{sub } r_d \ \rho_1 \ \rho_2 \mid \text{lt } r_d \ \rho_1 \ \rho_2 \mid \text{lea } r_d \ \rho_s \mid \text{load } r_d \ r_s \mid \text{store } r_d \ \rho_s \mid \text{restrict } r_d \ \rho_s \mid$
 $\text{subseg } r_d \ \rho_1 \ \rho_2 \mid \text{getp } r_d \ r_s \mid \text{getb } r_d \ r_s \mid \text{gete } r_d \ r_s \mid \text{geta } r_d \ r_s$

Fig. 3. The base Cerise capability machine's words, state, and instructions.

EXEC SINGLE

$$(\text{Running}, \sigma) \rightarrow \begin{cases} \llbracket \text{decode}(z) \rrbracket(\sigma) & \text{if } \sigma.\text{reg}(\text{pc}) = (p, b, e, a) \wedge b \leq a < e \wedge p \in \{RX, RWX\} \wedge \sigma.\text{mem}(a) = z \\ (\text{Failed}, \sigma) & \text{otherwise} \end{cases}$$

i	$\llbracket i \rrbracket(\sigma)$	Conditions
...	...	...
load $r_d \ r_s$	updPC($\sigma[\text{reg}.r_d \mapsto w]$)	$\sigma.\text{reg}(r_s) = (p, b, e, a)$ and $w = \sigma.\text{mem}(a)$ and $b \leq a < e$ and $p \in \{RO, RX, RW, RWX\}$
jmp r	(Running, $\sigma[\text{reg}.pc \mapsto \text{new}]$)	if $\sigma.\text{reg}(r) = (e, b, e, a)$ then $\text{new} = (RX, b, e, a)$ else $\text{new} = \sigma.\text{reg}(r)$
getb $r_d \ r_s$	updPC($\sigma[\text{reg}.r_d \mapsto b]$)	$\sigma.\text{reg}(r_s) = (_, b, _, _)$

$$\text{updPC}(\sigma) = \begin{cases} (\text{Running}, \sigma[\text{reg}.pc \mapsto (p, b, e, a + 1)]) & \text{if } \sigma.\text{reg}(\text{pc}) = (p, b, e, a) \\ (\text{Failed}, \sigma) & \text{otherwise} \end{cases} \quad \text{decode} : \mathbb{Z} \rightarrow \text{Instr}$$

Fig. 5. Excerpt of the operational semantics for of a single instruction. The complete operational semantics can be found in the Supplemental Materials, under Appendix A.

3.1 Background: Capability Machines and Cerise Operational Semantics

Hardware capabilities are one technique to isolate multiple software components executing on a single system. The ideas date back to seminal work by Dennis and Van Horn [12] and Carter et al. [9], yet more recently, they have garnered renewed interest through the Capability Hardware Enhanced RISC Instructions (CHERI) project [45]. CHERI ISAs offer fine-grained memory protection and fast protection-domain crossing through *capabilities*, unforgeable tokens that represent authority over memory or other resources. Since capabilities are unforgeable, they support the secure interaction between trusted code and untrusted, potentially adversarial code, by restricting the set of capabilities the untrusted code has access to, as depicted in Figure 4.

Figure 3 defines the machine state and instruction syntax of the Cerise capability machine, loosely modeled after CHERI. Chiefly, Cerise carefully distinguishes two kinds of machine words: regular integers $\mathbb{Z}$ for arithmetic and capabilities Cap for memory accesses and function closures. A Cerise capability is a tuple (p, b, e, a) , representing a memory region in the interval $[b, e)$, a permission p (such as read R , write W , execute X), and a memory address a which the capability points to. The machine's state is represented as a pair $(\text{Regs} \times \text{Mem})$. It has three execution states: Running, Halted or Failed. The register file $\text{reg} : \text{Regs}$ and the memory $\text{mem} : \text{Mem}$ map register names and memory addresses to machine words, respectively.

Figure 5 shows an excerpt of the operational semantics. At each step of execution, the rule EXEC SINGLE is applied if the machine is in the Running state. The machine first checks if the PC

$P, Q \in iProp ::=$	$\text{True} \mid \text{False} \mid \forall x. P \mid \exists x. P \mid \dots$	higher-order logic
	$\mid P * Q \mid P \multimap Q \mid [\varphi] \mid \Box P \mid \triangleright P \mid \boxed{P}$	separation logic and invariants
	$\mid a \mapsto lw \mid r \Rightarrow lw$	machine resources
	$\mid \langle P \rangle \rightarrow \langle s. Q \rangle \mid \{P\} \rightsquigarrow \bullet$	program logic

Fig. 6. Syntax of the Cerisier program logic.

contains a valid, executable capability. It then decodes the instruction i stored at current address a , and updates machine state according to the instruction's semantics $\llbracket i \rrbracket(\sigma)$, detailed in the table. Like other capability machines, Cerise instructions ensure that capabilities are unforgeable, i.e. they can only be derived from other capabilities with more authority. Memory accesses can only be performed with memory capabilities, and the machine dynamically checks the access against their range and permissions. For instance, executing the load $r_d \ r_s$ instruction requires a capability in source register r_s with at least r permission, and an in-bounds address $a \in [b, e)$. If so, the register file is updated to map destination register r_d to the word in memory at address a .

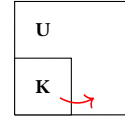
Controlled invocation is achieved with so-called *sealed entry* (*sentry*) capabilities, with permission ϵ . They can be thought of as opaque and immutable, only to be jumped to, roughly a low-level encapsulated closure. For instance, the `jmp` instruction will “upgrade” a capability's ϵ permission to rx when invoked, thus giving the invokee access to the capability's memory range (where private state may be stored).

3.2 Cerise Program Logic

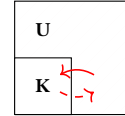
For reasoning, we start from the Cerise program logic [18], formalized and mechanized in Iris [26] and Rocq. It supports modular reasoning about systems combining both trusted and adversarial code.

The syntax of the program logic is presented in Figure 6 and extends Iris, an impredicative higher-order separation logic. It contains the usual higher-order logic quantifiers and connectives: universal $\forall$ and existential $\exists$ quantification, conjunction $\wedge$ and disjunction $\vee$, *etc.*, as well as the separating conjunction $*$ and magic wand $\multimap$. The pure proposition $[\varphi]$ holds if the meta-proposition φ holds in the meta-logic (in our case, Rocq). Iris comes with two kinds of propositions: ephemeral and persistent propositions. Ephemeral propositions might be invalidated at some point, and are therefore not duplicable. On the other hand, persistent propositions are, once valid, always valid. The persistently modality $\Box$ describes the persistent part of the proposition P . The later modality $\triangleright$ is mostly technical. It roughly means that the proposition P holds *after one logical step of execution*. The invariant $\boxed{P}$ states that P holds for now and forever. P can be accessed for one step of execution, but must be restored by the end of the step. We refer the reader to Jung et al. [26] for details.

To connect the logic with the machine's physical state, the program logic contains points-to separation logic predicates for registers ($r \Rightarrow w$), memory addresses ($a \mapsto w$) and contiguous memory ranges ($[b, e) \mapsto l$ for a list of words l). Furthermore, two different kinds of triples (both built on top of Iris' weakest-precondition) describe program specifications:



(a) Scenario 1: passing control to untrusted code



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

Fig. 4. Two scenarios: a trusted component (K) interacts with an untrusted context (U).

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \quad p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}\} \quad b \leq a < e \quad \text{decode}(n) = \text{Load } r_{dst} \ r_{src}}{\left\langle \begin{array}{l} pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, a_{pc}) \\ * a_{pc} \mapsto n * a \mapsto w \\ * r_{src} \Rightarrow (p, b, e, a) * r_{dst} \Rightarrow - \end{array} \right\rangle \rightarrow \left\langle \begin{array}{l} pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, \textcolor{orange}{a_{pc} + 1}) \\ * a_{pc} \mapsto n * a \mapsto w \\ * r_{src} \Rightarrow (p, b, e, a) * r_{dst} \Rightarrow \textcolor{orange}{w} \end{array} \right\rangle}$$

where $\text{ValidPC}(p, b, e, a, v) \triangleq p \in \{\text{RX}, \text{RWX}\} \wedge b \leq a < e$

Fig. 7. WP rule for a valid load. Changes between the pre- and postcondition are marked in orange.

- The single instruction execution $\langle P \rangle \rightarrow \langle s. Q \rangle$ states that, starting in a state satisfying P , the machine reaches a state satisfying Q after one step of execution, with execution state s . We omit s if the machine remains in the Running state.
- The complete, safe execution $\{P\} \rightsquigarrow \bullet$ states that, starting in a state satisfying P , the machine either diverges, or runs until it halts or fails without breaking registered invariants.

Each instruction comes with a program logic rule describing its behavior in the logic. For example, Figure 7 shows the load instruction rule. We refer to Georges et al. [18] for a more detailed exposition.

Cerise establishes *capability safety*, a formal property that captures interactions between known, trusted code K invoking (Figure 4a) and being invoked by (Figure 4b) unknown, untrusted code U . Safety of such interactions is proved by establishing a Hoare triple for the known code using program logic rules, and appealing to capability safety when unknown code is invoked. Capability safety is formalized as a universal contract: a contract for arbitrary code, expressing that its execution will not exceed the authority of the capabilities it has access to, or break registered invariants. The authority of capabilities is defined using a logical relation³ $\mathcal{V}$, which is defined using the program logic, and which we explain in the following.

THEOREM 3.1 (CERISE UNIVERSAL CONTRACT).

$$\vdash \left\{ \bigstar_{r \in \text{RegName}} (\exists w. r \Rightarrow w * \mathcal{V}(w)) \right\} \rightsquigarrow \bullet$$

Note that the Cerise universal contract is usually phrased in a different form, as a Fundamental Theorem of Logical Relations (FTLR) for $\mathcal{V}$. This FTLR is equivalent to the universal contract in Theorem 3.1 but harder to explain, so we will not discuss it in this paper.

Figure 8 shows the definition of $\mathcal{V}$ (“safe to share (with the adversary)”) and the supporting notion $\mathcal{E}$ (“safe to execute (by the adversary)”). Intuitively, a word w is *safe to share* if authority is available for anything the adversary can do with the word. We say that a word w is *safe to execute* if the machine executes safely when w is installed in the pc with safe words in all registers. Integers $z \in \mathbb{Z}$ and null-permission (o) capabilities are trivially safe.

A sentry capability ($\mathbb{E}$) is safe to share when the code it encapsulates is safe to execute. The later modality $\triangleright$ ensures that the apparently circular definition of $\mathcal{V}$ is well-defined. The persistent modality $\square$ expresses the fact that the sentry capability can be invoked several times. A RW/RWX ⁴ capability gives access to the memory region $[b, e)$, and the (authority of the) words it contains.

³Traditionally, logical relations are used to interpret type systems, but they can also be used to interpret the universal type used by an untyped language [13, 33], which is essentially what is happening in Cerise.

⁴The universal contract does not distinguish between RW/RWX , because it models an interpretation of capability safety where the security model makes no restrictions on the code blocks that an adversary has access to for execution.

$$\begin{aligned}
\mathcal{E}(w) &\triangleq \left\{ \text{pc} \Rightarrow w * \left(\bigstar_{(r, w_r) \in \text{regs}} r \Rightarrow w_r * \mathcal{V}(w_r) \right) \right\} \rightsquigarrow \bullet \\
\mathcal{V}(w) &\triangleq \begin{cases} \mathcal{V}(z) & \triangleq \text{True for } z \in \mathbb{Z} \\ \mathcal{V}(o, -, -, -) & \triangleq \text{True} \\ \mathcal{V}(E, b, e, a) & \triangleq \triangleright \square \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(\text{RW/RWX}, b, e, -) & \triangleq \bigstar_{a \in [b, e)} \boxed{\exists w. a \mapsto w * \mathcal{V}(w)}^a \\ \mathcal{V}(\text{RO/RX}, b, e, -) & \triangleq \bigstar_{a \in [b, e)} \exists P. \boxed{\exists w. a \mapsto w * P(w)}^a \\ & \quad * \triangleright \square (\forall w. P(w) \multimap \mathcal{V}(w)) \\ \mathcal{V}([ps, o_b, o_e, o_i]) & \triangleq ([s \in ps] * \text{safe_to_seal}(o_b, o_e)) \\ & \quad * ([\cup \in ps] * \text{safe_to_unseal}(o_b, o_e)) \\ & \quad * ([ps = sv] * \text{safe_to_deinit}(o_b, o_e)) \\ \mathcal{V}(\{sc\}_o) & \triangleq \exists P. \text{seal_pred}(o, P) * P(sc) \end{cases}
\end{aligned}$$

where

$$\begin{aligned}
\text{regs} &\triangleq \{(r, w) \mid (r, w) \in \text{reg}, r \neq \text{pc}\} \\
\text{safe_to_seal}(o_b, o_e) &\triangleq \bigstar_{o_a \in [o_b, o_e)} \exists P. [\forall w. \text{Persistent}(P(w))] * \text{seal_pred}(o_a)(P) \\
&\quad * (\forall w. \mathcal{V}(w) \multimap P(w)) \\
\text{safe_to_unseal}(o_b, o_e) &\triangleq \bigstar_{o_a \in [o_b, o_e)} \exists P. \text{seal_pred}(o_a)(P) * (\forall w. P(w) \multimap \mathcal{V}(w)) \\
\text{safe_to_deinit}(o_b, o_e) &\triangleq \bigstar_{o_a \in [o_b, o_e)} \exists \text{tidx}. [\text{tidx_of_otype}(o_a) = \text{tidx}] \\
&\quad * \boxed{\exists I. \text{tidx} \mapsto_E I \vee \text{DeInitialized}(\text{tidx})}^{\text{tidx}}
\end{aligned}$$

Fig. 8. Logical relation defining “safe to share” ($\mathcal{V}$) and “safe to execute” ($\mathcal{E}$). The definitions marked in blue can be ignored for now, and will be explained in Section 3.3. `safe_to_deinit` is included for completeness and necessary for deinitialization of enclaves, but is not discussed further.

They are safe to share only if the memory locations are owned by an invariant that also enforces safety of the words they contain. Finally, similarly to RW/RWX, a RO/RX capability gives access to the memory region in between its bounds, but its contents cannot be modified. The logical relation expresses read-only access by requiring addressed memory to invariantly satisfy an arbitrary predicate P , that is only known to entail $\mathcal{V}$.

Finally, Theorem 3.1 expresses that if all registers contain safe to share values, the machine executes safely. It provides the foundation for reasoning about invoking untrusted code from trusted code, where verifying the safety of register contents becomes a proof obligation.

3.3 Sealed Capabilities

The idea of CHERI sealed capabilities dates back to Morris [29] and implements a form of symbolic cryptography [41], orthogonal to trusted execution. The act of sealing makes a capability opaque and immutable: it cannot be modified or dereferenced until it is unsealed. To support sealing, Cerisier extends Cerise operational semantics with two instructions, `cseal` and `cunseal` to perform sealing and unsealing of capabilities. The changes to the operational semantics are shown in Figure 9.

To seal a capability, the machine takes a capability sc and a valid *sealing capability* $[sp, o_b, o_e, o_a]$, a new type of capability holding the authority to seal (or unseal). The permission sp describes the sealing permission, for either sealing (s) or unsealing (u). Instead of addresses, $[o_b, o_e]$ describes a

$$\begin{array}{ll}
o \in \text{OType} & \triangleq [0, \text{OTypeMax}] \\
sp \in \text{SealPerm} & ::= su \mid s \mid u \mid o \\
sr \in \text{SRange} & \triangleq \{[sp, o_b, o_e, o_a] \mid \\
& \quad o_b, o_e, o_a \in \text{OType}\} \\
sc \in \text{Sealable} & \triangleq \text{Cap} + \text{SRange} \\
sdc \in \text{SCap} & \triangleq \text{OType} \times \text{Sealable} \\
w \in \text{Word} & \triangleq \mathbb{Z} + \text{Cap} \text{ Sealable} + \text{SCap}
\end{array}$$

$$i \in \text{Instr} ::= \dots \mid \text{cseal } r_d \ r_1 \ r_2 \mid \text{cunseal } r_d \ r_1 \ r_2 \mid$$

i	$\llbracket i \rrbracket(\sigma)$	Conditions
$\text{cseal } r_d \ r_1 \ r_2$	$\text{updPC}(\sigma[\text{reg}.r_d \mapsto \{sc\}_{o_a}])$	$\sigma.\text{reg}(r_1) = [sp, o_b, o_e, o_a]$ and $\sigma.\text{reg}(r_2) = sc$ and $sp \in \{s, su\}$ and $o_b \leq o_a < o_e$
$\text{cunseal } r_d \ r_1 \ r_2$	$\text{updPC}(\sigma[\text{reg}.r_d \mapsto sc])$	$\sigma.\text{reg}(r_1) = [sp, o_b, o_e, o_a]$ and $sp \in \{u, su\}$ and $\sigma.\text{reg}(r_2) = \{sc\}_{o_a}$ and $o_b \leq o_a < o_e$

Fig. 9. Excerpt of the operational semantics with capability sealing extensions, extending Figure 5.

range of so-called *otypes*, acting as key identifiers. Similarly to regular capabilities, the current otype o_a needs to be in bounds to be used. The action of sealing creates a *sealed capability* $\{sc\}_{o_a}$, making it opaque and immutable. Conversely, to unseal a capability, the machine takes a sealed capability $\{sc\}_{o_a}$ and a valid (un-)sealing capability $[sp, o_b, o_e, o_a]$ with unsealing permission ($u \in sp$) and matching otype, and retrieves the initial capability sc .

Sealing capabilities cannot be forged, they can only be monotonically derived from other sealing capabilities. In Cerisier, new sealing capabilities, with fresh unique otypes, are generated when enclaves are initialized, as we detail in the next section. The operational semantics also includes a couple of other new instructions: `getwtype` for querying the type of a word (integer, regular capability, sealing capability or sealed capability) and `getotype` for querying the otype o of a sealed capability; their precise semantics can be found in the accompanying Rocq formalization.

By sharing restricted sealing capabilities with an adversary, one obtains a form of asymmetric cryptography: A component holding a private sealing capability $[su, o_b, o_e, o_a]$, can, e.g., disclose the unsealing authority $[u, o_b, o_e, o_a]$ to the adversary and keep the full authority private to obtain a form of cryptographic signatures. Conversely, making the sealing authority public, enables a form of asymmetric encryption. In fact, the public/private key pair for attestation suggested in Figure 2, corresponds in reality to a single su sealing capability used in this way for signing messages.

Following Skorstengaard et al. [39] we extend the Cerise program logic with support for sealing by adapting a technique by Sumii and Pierce [41]. The logical relation is extended by the definition marked in blue in Figure 8. Logically, we associate a value predicate P with every otype o , such that, for any capability sealed with this otype, the predicate holds. For instance, a possible predicate could be “any capability sealed with otype o will have its current address set to 42”. The association is witnessed by a predicate $\text{seal_pred}(o, P)$. When sharing o -sealed values with the adversary, there is an obligation to show that the value being sealed respects P ; and dually, it gives the guarantee that o -sealed values received from the adversary also respect P . We defer further details to Supplemental Materials, under Appendix E.

3.4 Exclusive Access and Memory Revocation

As mentioned in Section 1, when an enclave is initialized, Cerisier implements the exclusive memory ownership primitive of CHERI-TrEE by a memory sweep, to check that the entire memory (as well as the register file) does not contain a capability overlapping the new enclave’s memory. We model the memory sweep as described by the CHERI-TrEE’s operational semantics, and we discuss its hardware implementation in Section 6.1.

$$i \in \text{Instr} ::= \dots \mid \text{isunique } r_d \ r_s \mid$$

i	$\llbracket i \rrbracket(\sigma)$	Conditions
$\text{isunique } r_d \ r_s$	$\text{updPC}(\sigma[\text{reg}.r_d \mapsto z])$	$\left(\begin{array}{l} \sigma.\text{reg}(r_s) = (p, b, e, a) \\ \vee \sigma.\text{reg}(r_s) = \{(p, b, e, a)\}_{o_i} \end{array} \right)$ and $\left(\begin{array}{ll} \text{if} & \text{sweep}(\sigma)(r_s) \\ \text{then} & z = 1 \quad \text{else } z = 0 \end{array} \right)$

$$\text{overlap}(w_1)(w_2) = \begin{cases} [b_1, e_1] \cap [b_2, e_2] & \text{if } (w_1 = (p_1, b_1, e_1, a_1) \vee w_1 = \{(p_1, b_1, e_1, a_1)\}_o) \\ & \text{and } (w_2 = (p_2, b_2, e_2, a_2) \vee w_2 = \{(p_2, b_2, e_2, a_2)\}_o) \\ \perp & \text{otherwise} \end{cases}$$

$$\text{sweep}(\sigma)(r_s) = \begin{cases} \forall r \in \text{dom}(\sigma.\text{reg} \setminus r_s). \neg \text{overlap}(\sigma.\text{reg}(r), \sigma.\text{reg}(r_s)) \\ \wedge \forall a \in \text{dom}(\sigma.\text{mem}). \neg \text{overlap}(\sigma.\text{mem}(a), \sigma.\text{reg}(r_s)) \end{cases}$$

Fig. 10. Excerpt of the operational semantics with revocation-related extensions, extending Figure 5.

In addition to enclave initialization, the sweep can also be used after initialization through an `isunique` instruction, in order to support dynamically growing enclaves and memory-sharing communication. The operational semantics is shown in Figure 10, by comparison to Figure 3 and Figure 5. The instruction `isunique  $r_d \ r_s$` requires a (possibly sealed) capability (p, b, e, a) in source register r_s . The machine will sweep its entire state σ , i.e., the register file (excluding r_s) and the entire memory, to check that no capabilities overlap the source capability. The result is stored in destination register r_d as an integer 1 or 0, indicating that the source capability is unique, or has duplicates, respectively.

Logically, the memory sweep, akin to a stop-the-world garbage collection, corresponds to a *revocation* of ownership: memory that was previously owned by the adversary and that overlaps with the memory region of the capability in r_s is logically revoked and reallocated to the enclave. The revocation step is challenging to support in Cerisier, because it is a highly non-modular operation: the revocation affects the global state, whereas separation logics like Iris, which Cerisier builds upon, support only local (so-called frame-preserving) operations, in order to validate the celebrated frame rule of separation logic. Fortunately, we can adapt and reuse a technique by Hur et al. [23] for reasoning about garbage collection, which involves a similar global operation. They introduce an additional layer of *logical* memory, which allows addresses to be reallocated when the unique memory ownership sweep succeeds. In essence, the logical memory decorates the physical memory with version numbers, which are incremented at each successful memory sweep operation. An invariant links the physical state of the machine to logical memory, ensuring that only the latest version number of each memory address is reachable from garbage collection roots.

Because our support for revocation is an adaptation of a known technique to our setting, we omit full details and simply summarize the program logic interface for revocation. Essentially, we add versions to points-to predicates: $a \mapsto_v w$ expressing that address a at version v points to logical word w . *Logical words* are either integers or logical capabilities $(p, b, e, a)_v$, which carry a version number (note that all addressed logical addresses must be at the same version). Logical capabilities can only be used with the points-to predicate for the same version. Existing Cerise program logic rules are straightforwardly extended with versions, which they leave unmodified.

The program logic rule for the `isunique` instruction in Figure 11 shows the interface for memory revocation. Assuming we have the points-to predicate for the entire logical memory region of the swept capability, the rule describes what happens in two cases. In case of a successful sweep (i.e., the second part of the disjunction), the logical version of the capability in the source register

ISUNIQUEVALID

$$\frac{
\begin{array}{c}
\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc})_{v_{pc}} \\
p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}\} \quad \text{decode}(n) = \text{isunique } r_{\text{dst}} \ r_{\text{src}} \quad a_{pc} \notin [b, e)
\end{array}
}{
\left\langle \begin{array}{l}
pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, a_{pc})_{v_{pc}} \\
* a_{pc} \mapsto_{v_{pc}} n \\
* r_{\text{src}} \Rightarrow (p, b, e, a)_v \\
* r_{\text{dst}} \Rightarrow -
\end{array} \right\rangle \rightarrow \left\langle \begin{array}{l}
pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, a_{pc} + 1)_{v_{pc}} * a_{pc} \mapsto_{v_{pc}} n \\
* \left(\begin{array}{l}
r_{\text{dst}} \Rightarrow 0 * r_{\text{src}} \Rightarrow (p, b, e, a)_v \\
\vee \quad r_{\text{dst}} \Rightarrow 1 * r_{\text{src}} \Rightarrow (p, b, e, a)_{v+1} \\
* [b, e) \mapsto_{v+1} lw
\end{array} \right)
\end{array} \right\rangle$$

Fig. 11. WP rule for a valid case of the `isunique` instruction with distinct PC, source, and destination registers. The postcondition asserts that either the sweep was unsuccessful, writing 0 to the destination register, or it was successful and we increment the version number of the capability and all addresses within its bounds.

$$i \in \text{Instr} ::= \dots \mid \text{hash } r_d \ r_s \mid \text{hashconcat } r_d \ \rho_1 \ \rho_2$$

i	$\llbracket i \rrbracket(\sigma)$	Conditions
$\text{hash } r_d \ r_s$	$\text{updPC}(\sigma[\text{reg}.r_d \mapsto \text{hash}(w)])$	$\sigma.\text{reg}(r_s) = w$
$\text{hashconcat } r \ \rho_1 \ \rho_2$	$\text{updPC}(\sigma[\text{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \text{getWord}(\sigma, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 \parallel z_2$

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

Fig. 12. Excerpt of the operational semantics containing secure hashing extensions and assumptions, extending Figure 5. The notation $\parallel$ is the machine primitive of hash concatenation.

is incremented, and we obtain a fresh points-to predicate for the updated memory region. The points-to predicates of the old version may still exist, but are not linked to the physical state anymore. The soundness proof of the rule requires justifying that bumping the version number of the swept memory region does not invalidate the invariant linking physical and logical state. Informally, this is the case because the physical sweep ensures that there is no other capability overlapping the memory region, and so no other capabilities have to be updated.

3.5 Secure Hashing Primitives

As explained in the Introduction, attestation often involves hashing of a component's source code, etc. Therefore Cerisier also includes primitives for hashing, and we use them in Section 5.2 for implementing a mutual attestation scheme. The operational semantics of the hashing instructions is shown in Figure 12. The `hash  $r_d \ r_s$` instruction stores the hash of the word in r_s in r_d and `hashconcat  $r \ \rho_1 \ \rho_2$` takes two hash values z_1 and z_2 , and computes their concatenation $z_1 \parallel z_2$.

The hash computation is intended to model a secure hash algorithm, which we axiomatize in the form of injective and collision-free operations on natural numbers. These assumptions are not contradictory because Cerisier allows unbounded integers, and it allow us to avoid cross-cutting but orthogonal changes that would be required for a symbolic or probabilistic model. The precise assumptions we use are:

$$\begin{aligned}
\text{hash}(mem_1) = \text{hash}(mem_2) &\Rightarrow mem_1 = mem_2 \\
(z_1 \parallel z_2) = (z'_1 \parallel z'_2) &\Rightarrow z_1 = z'_1 \wedge z_2 = z'_2.
\end{aligned}$$

$$\begin{array}{ll}
ecn \in \text{ENum} & \triangleq \mathbb{N} \\
I \in \text{Eidentity} & \triangleq \mathbb{N} \\
tidx \in \text{TIndex} & \triangleq \mathbb{N} \\
etbl \in \text{ETable} & \triangleq \text{TIndex} \rightarrow \text{Eidentity} \\
\sigma \in \text{ExecConf} & \triangleq \text{Regs} \times \text{Mem} \times \text{ETable} \times \text{ENum}
\end{array}$$

$$i \in \text{Instr} ::= \dots \mid \text{einit } r_s \mid \text{edeinit } r_s \mid \text{estoreid } r_d \ r_1 \ r_2 \mid \text{hash } r_d \ r_s \mid \text{hashconcat } r_d \ \rho_1 \ \rho_2$$

i	$\llbracket i \rrbracket(\sigma)$	Conditions
$\text{einit } r_1 \ r_2$	$\text{updPC}(\sigma[$ $\text{mem}.b \mapsto (\text{RW}, b', e', a'),$ $\text{mem}.b' \mapsto [\text{SU}, o_a, o_{a+2}, o_a],$ $\text{etbl}.tidx \mapsto I,$ $\text{EC} \mapsto \sigma.\text{EC} + 1,$ $\text{reg}.r_1 \mapsto (e, b, e, b + 1),$ $\text{reg}.r_2 \mapsto 0])$	$r_1 \neq \text{pc}$ and $\sigma.\text{reg}(r_1) = (\text{RX}, b, e, a)$ and $b < e$ and $\sigma.\text{reg}(r_2) = (\text{RW}, b', e', a')$ and $b' < e'$ and $\text{sweep}(\sigma)(r_1)$ and $\text{sweep}(\sigma)(r_2)$ and $(\forall l \in [b + 1, e). \sigma.\text{mem}(l) \in \mathbb{Z})$ and $I = \text{hash}(b) \parallel \text{hash}(\sigma.\text{mem}([b + 1, e]))$ and $\text{tidx} = \text{fresh_tidx}(\sigma)$ and $o_a = \sigma.\text{EC} * 2$
$\text{edeinit } r$	$\text{updPC}(\sigma[\text{etbl}.tidx \mapsto 0])$	$\sigma.\text{reg}(r) = [\text{SU}, o_a, o_{a+2}, \dots]$ and $\text{tidx} = \text{tidx_of_otype}(o_a)$ and $\sigma.\text{etbl}(\text{tidx}) = I$
$\text{estoreid } r_d \ r_s$	$\text{updPC}(\sigma[\text{reg}.r_d \mapsto I])$	$\sigma.\text{reg}(r_s) = o_s$ and $\text{tidx} = \text{tidx_of_otype}(o_s)$ and $\sigma.\text{etbl}(\text{tidx}) = I$

$$\text{tidx_of_otype}(o_a) = \begin{cases} o_a/2 & \text{if } \text{is_even}(o_a) \\ (o_a - 1)/2 & \text{otherwise} \end{cases} \quad \text{fresh_tidx}(\sigma) = \sigma.\text{EC}$$

Fig. 13. Excerpt of the operational semantics with attestation-related extensions, extending Figure 5.

In practice, these conditions ensure that when a statically computed enclave identity (Figure 2, ID_{soc}) matches a dynamically computed enclave identity (Figure 2, k_p), we may conclude that the hashed memory regions must be equal.

4 Attestation

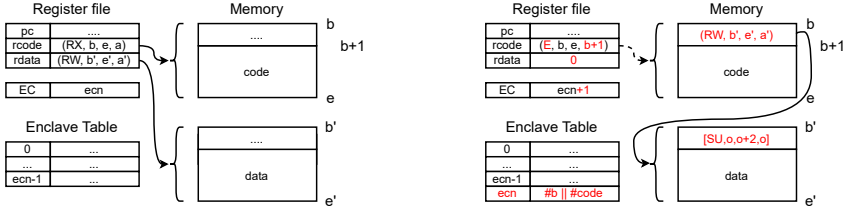
In this section we detail the operational semantics of the new attestation-related instructions included in Cerisier. The instructions are inspired by similar ones in CHERI-TrEE. Next, we show how the secure outsourced computation example from Section 2 is implemented in Cerisier, before we turn to reasoning about attestation and clients of attested enclaves in the program logic. Finally, we discuss how to reason about unknown adversarial code using the novel Cerisier universal contract, the proof of which is outlined in the final subsection.

4.1 Operational Semantics

The new attestation-related instructions in the Cerisier ISA are `einit`, `estoreid`, and `edeinit` for enclave initialization, attestation, and deinitialization, respectively. Their operational semantics are defined in Figure 13. They interact with an *enclave table* that tracks registered enclaves' identities. A special register `EC`, not directly accessible by the user, always contains the next free available table index. We first discuss the `einit` instruction, whose effects on the machine state are depicted in Figure 14.

Enclave initialization using the `einit  $r_1 \ r_2$` instruction expects the enclave to consist of two parts: a code and data region. As Figure 14 shows, `einit` requires a non-empty `RW` capability for the data region $[b', e')$, and a non-empty `RX` capability for the code region at $[b, e)$. It will check unique ownership of the two regions, similar to what was done for `isunique` in Section 3.5. Following successful sweeps, enclave initialization proceeds without further validation.

The enclave's memory layout is updated as follows. The data region capability (RW, b', e', a') is stored at the code region's base address b . The rest of the code region $[b + 1, e)$ is expected to



(a) Cerisier machine state prior to `einit` (b) Cerisier machine state following `einit`

Fig. 14. Representation of the effects in memory of the `einit` instruction.

contain the enclave instructions. A sentry capability $(E, b, e, b+1)$ to the first of these instructions will be stored in r_1 , to be safely invoked by the context. Two otypes $o_a := 2 * \text{tidx}$ and $o_a + 1$ are reserved for the enclave to use for encryption and signing (see below). This derivation from the fresh table index tidx ensures the otypes are fresh and exclusively accessible to the enclave. A sealing capability $[SU, o_a, o_a + 2, o_a]$ for the reserved otypes is written to the first address b' of the data region.

Given an enclave in this layout, Cerisier will compute the enclave identity using the hashing primitives described in Section 3.5. Specifically, it will hash the code region of the enclave, excluding the data capability; *i.e.*, the contents of the memory range $[b+1, e)$. We additionally include the address b in the enclave identity. The data region contents is not included, but the enclave code can later do integrity checks over it if necessary.

After computing the new enclave's identity, Cerisier updates its machine state accordingly. The enclave table is updated at the table index tidx to contain the new enclave with its identity and the EC register is bumped to point to the next free table index.

To allow enclaves to prove their identity to distrusting clients, enclaves receive exclusive access to sealing capabilities for two otypes, after initialization. Those clients can verify the identity of the enclave using such an otype by querying the machine state using the `estoreid` instruction. The instruction `estoreid rd rs` reads an otype o from the source register r_s , queries the enclave table for the identity of the enclave at index $\lfloor o/2 \rfloor$, if it exists, and stores the identity in the destination register r_d . Based on this identity, the client can decide whether it trusts the enclave, typically by comparing it to pre-computed identities of trusted code.

The `edeinit rs` instruction enables early deinitialization using the enclave's sealing capability. Once deinitialized, an enclave tidx cannot reappear, as the EC acts as a bump allocator. While the original CHERI-TrEE design includes it, we make no use of it in our case studies. Nevertheless, with the universal contract (Section 4.5), we prove that it does not introduce any new attack vectors.

4.2 SOC Example in Cerisier

To demonstrate use of the Cerisier attestation primitives, this section revisits the SOC example from Section 2. Figure 15 shows the example in pseudocode Cerisier assembly. The complete listing can be found in the Supplemental Materials, under Appendix B.

Recall that after the adversary initializes the enclave, the machine ensures exclusive ownership of the enclave's memory region, installs the (fresh) enclave keys in the data region, creates a new entry in the enclave table, and returns a sentry capability that can be invoked. When the SOC enclave is invoked, it first fetches the encryption and signing keys from the sealing capability

```

Enclave  $\triangleq$  keys  $\leftarrow$  Data[0]
           // keys := [su, o, o + 2, o]
           ot  $\leftarrow$  getb(keys) + 1
           sign_keys  $\leftarrow$  lea(subseg(keys, ot, ot + 1), 1)
           // sign_keys := [su, o + 1, o + 2, o + 1]
           sign_pub_key  $\leftarrow$  restrict(sign_keys, u)
           // sign_pub_key := [u, o + 1, o + 2, o + 1]
           r  $\leftarrow$  42 // computation...
           {r}ot  $\leftarrow$  seal(r, sign_keys)
           return ({r}ot, sign_pub_key)

IDSOC  $\triangleq$  hash(Addr(Enclave), Enclave)

Client  $\triangleq$  ({r}ot, sign_pub_key)  $\leftarrow$  Adversary()
         ot  $\leftarrow$  getotype({r}ot)
         I  $\leftarrow$  estoreid(ot)
         if I  $\neq$  IDSOC then abort
         r  $\leftarrow$  unseal({r}ot, sign_pub_key)
         assert r = 42

Adversary  $\triangleq$  enclave*  $\leftarrow$  einit(Enclave, Data)
           return enclave()

```

Fig. 15. SOC using Cerisier primitives, adapted from Figure 2. The *lea*, *subseg*, and *restrict* primitives modify the sealing capability’s address, bounds, and permissions, respectively. In the adversary’s initialization of the enclave, Data can be any rw memory capability at least one address wide to hold the sealing capability. Pseudocode adapted from the full listing in the Supplemental Materials, under Appendix B.

[su, o, o + 2, o] installed in the data section. Since our enclave only uses signatures, not encryption, we discard *o*, and split the sealing capability for *o* + 1 into a private signing key (with permission s), and a public key (with permission u), which will attest that the signed message originates from the enclave. Finally, the enclave signs the result using the private signing key, and shares it together with its public signing key.

Separately, the client will contain the (pre-computed) static identity of the SOC enclave to check the enclave’s attest. Eventually, the client is resumed with a sealed result and public signing key from the SOC enclave. The former is really a word sealed with the SOC’s signing otype *ot*, while the latter is an unsealing capability [u, *ot*, *ot* + 1, *ot*]. Using the *estoreid* instruction, the client first queries the enclave table to get the dynamic identity corresponding to the otype of the sealed result. Then, it verifies that the enclave that signed the value is indeed the SOC enclave. Finally, the sealed value {*r*}_{*ot*} is unsealed, and verified to be 42. As the SOC enclave only signs (seals) the value 42, we know that the *assert* will not fail.

4.3 Attestation Reasoning

Formally capturing the rather special intuitive reasoning model for attestation, as described in Figure 1, is the key challenge for this paper. Initially, only the client is trusted, that is, we consider a system that is only known to contain the trusted code and unknown, adversarial code (which may or may not contain an enclave). When the client successfully validates an attested message, the trust model changes. The validation tells us that an enclave with the specified identity was successfully initialized on the system and has received exclusive authority over the otypes. Then, we additionally verify the attested enclave, assuming the information captured in the attested identity. If we can verify that the enclave properly protects the received sealing capabilities and only uses them to sign messages containing correct results, we finally learn something about the received message.

Our approach to formalize this reasoning is to extend the universal contract with an additional guarantee about sealed capabilities produced by adversarial code. As discussed in Section 3.3, the program logic allows associating an otype with a sealing predicate that must be satisfied by all

$$\begin{array}{c}
\text{ESTOREIDATTESTVALID} \\
\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc})_{v_{pc}} \quad \text{decode}(n) = \text{estoreid } r_d \ r_s \quad \text{tidx_of_otype}(o_a) = \text{tidx} \\
\hline
\left\langle \begin{array}{l} pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, a_{pc})_{v_{pc}} * \text{EC}(ecn) \\ * a_{pc} \mapsto_{v_{pc}} n * r_s \Rightarrow o_a * r_d \Rightarrow _ \end{array} \right\rangle \rightarrow \left\langle s. \left(\begin{array}{l} \exists I. [s = \text{Running}] \\ * pc \Rightarrow (p_{pc}, b_{pc}, e_{pc}, a_{pc} + 1)_{v_{pc}} \\ * \text{EC}(ecn) * a_{pc} \mapsto_{v_{pc}} n \\ * r_s \Rightarrow o_a * r_d \Rightarrow I \\ * \text{tidx} \mapsto_E^\square I * [0 \leq \text{tidx} < ecn] \\ \vee ([s = \text{Failed}] * \dots) \end{array} \right) \right\rangle
\end{array}$$

Fig. 16. Rules for the new enclave instructions. The “...” represent precondition resources being returned.

capabilities sealed with the otype. If a client learns by querying the enclave table that a pair of otypes (o_1, o_2) corresponds to a specific enclave identity I , our program logic guarantees that the sealing predicate corresponding to o_i must be a specific pair of predicates P_{o_1}, P_{o_2} .

For all enclaves of interest, these enclave sealing predicates P_{o_1}, P_{o_2} must be provided to the universal contract in an argument $\mathcal{M}$, mapping enclave identities to the sealing predicates they enforce. A global *enclave identity invariant* $I(\mathcal{M})$ ties the sealing predicates and the identities of the custom enclaves together, ensuring that for any enclave in $\mathcal{M}$ ever initialized, the sealing predicate will effectively be P_{o_1}, P_{o_2} .

In addition, each entry in $\mathcal{M}$ must be accompanied by program logic proofs that any enclave instance with identity I respects the sealing predicates P_{o_1}, P_{o_2} for the otypes o_1, o_2 it receives at initialization. This requires that the enclave itself uses its sealing capabilities only to seal capabilities satisfying P_{o_1} and P_{o_2} , but also that the enclave adequately protects its private state and the sealing capabilities to prevent the adversary from using them to seal other capabilities.

4.4 Reasoning about Attested Enclaves’ Clients

To model this intuitive reasoning, our program logic (Figure 6) can represent knowledge about the currently registered enclaves and their identity. For brevity, we restrict ourselves to the assertions and relations that are necessary to reason about initially untrusted enclaves communicating with initially trusted clients, even though Cerisier offers additional rules for alternative trust models. Moreover, we do not discuss enclave deinitialization.

Enclave clients can establish trust in the identity of enclaves using the `estoreid` instruction. The persistent $\text{tidx} \mapsto_E^\square I$ resource represents the knowledge about an enclave with identity I that has been initialized at table index tidx .⁵

Enclave resources $\text{tidx} \mapsto_E^\square I$ can be obtained by clients using the `EStoreIdAttestValid` rule in Figure 16. The rule specifies the contract for the enclave attestation instruction `EStoreId`. The instruction queries the CPU for the identity of the enclave corresponding to a given otype o_a . It has two possible outcomes, depending on whether this otype corresponds to the enclave sealing capabilities of an initialized enclave. If attestation succeeds, the identity of enclave I was found in the enclave table and is stored in the destination register r_d . A token $\text{tidx} \mapsto_E^\square I$ is created, witnessing that an entry tidx for the otype o_a , with identity I exists in the enclave table.

⁵The resource $\text{tidx} \mapsto_E^\square I$ does not guarantee that the enclave is still live, i.e. it may have already deinitialized itself. The resource continues to be useful because messages from the enclave may still reach communication partners after deinitialization.

$$\begin{aligned}
\mathcal{I}(\mathcal{M}) &\triangleq \exists ecn. (\textcolor{teal}{A}) \text{EC}(ecn) * (\textcolor{teal}{B}) (*_{o \in [0, 2*ecn+1)} \exists P. \text{seal_pred}(o)(P)) \\
&\quad * (\textcolor{teal}{C}) \text{custom_enclaves_predicates}(\mathcal{M})(ecn) \\
\text{where } \text{custom_enclaves_predicates}(\mathcal{M})(ecn) &\triangleq \\
&\quad \square \forall I \in \text{dom}(\mathcal{M}), 0 \leq \text{tidx} < ecn, o. [\text{tidx_of_otype}(o) = \text{tidx}] * \text{tidx} \mapsto_{\text{E}}^{\square} I \multimap \\
&\quad \text{seal_pred}(o)(\mathcal{M}(I).P_{enc}) * \text{seal_pred}(o+1)(\mathcal{M}(I).P_{sign})
\end{aligned}$$

Fig. 17. Global invariant $\mathcal{I}$ parameterised by user-provided map $\mathcal{M}$ of enclaves of interest (*custom enclaves*).

$$\begin{aligned}
\text{custom_enclaves_contract}(\mathcal{M}) &\triangleq \forall I \in \text{dom}(\mathcal{M}), \text{data}, b, e, v, p', b', e', a', v', o. \\
&\quad \left(\begin{array}{l} [b = \mathcal{M}(I).addr] * [\text{tidx} = \text{tidx_of_otype}(o)] \\ * (\textcolor{teal}{1}) b \mapsto_v (p', b', e', a')_{v'} * [b+1, e] \mapsto_v \mathcal{M}(I).code \\ * (\textcolor{teal}{2}) b' \mapsto_{v'} [\text{su}, o, o+2, o] * [b'+1, e'] \mapsto_{v'} \text{data} \\ * (\textcolor{teal}{3}) \text{seal_pred}(o, \mathcal{M}(I).P_{enc}) * (\textcolor{teal}{4}) \text{seal_pred}(o+1, \mathcal{M}(I).P_{sign}) \\ * (\textcolor{teal}{5}) \text{tidx} \mapsto_{\text{E}} I \end{array} \right) \multimap \mathcal{V}((\text{E}, b, e, a)_v)
\end{aligned}$$

Fig. 18. Contract of custom enclaves, parameterised by user-provided map $\mathcal{M}$, stating that all enclaves of interest are safe to execute, and that they respect the sealing predicate defined in $\mathcal{M}$.

To use the knowledge learned using `EStoreId`, enclave clients can combine the $\text{tidx} \mapsto_{\text{E}}^{\square} I$ resource with user-provided information about a set of enclaves of interest. This information is represented in the user-defined map of *custom enclaves* $\mathcal{M} : \text{Eidentity} \rightarrow \text{List}(\text{Word}) \times \text{Addr} \times i\text{Prop} \times i\text{Prop}$, which maps each known enclave identity I to its code $\mathcal{M}(I).code$, the base address $\mathcal{M}(I).addr$ and the sealing predicates for encryption $\mathcal{M}(I).P_{enc}$ and signing $\mathcal{M}(I).P_{sign}$. The map is assumed to be well-formed, *i.e.*, the identity should correspond to the hash of the code and base address:

$$\forall I. \text{hash}(\mathcal{M}(I).addr) \parallel \text{hash}(\mathcal{M}(I).code) = I.$$

Given this user-provided map, the system-wide *enclave identity invariant* $\mathcal{I}(\mathcal{M})$ in Figure 17 ties the sealing predicates and the identities of the custom enclaves together. It owns the resource $\text{EC}(ecn)$ tracking the number of registered enclaves (A), and states that the seal predicates for all otypes up-to the current enclave counter must be initialized (B). Moreover, *custom_enclaves_predicates* (C) states that for all identities in the custom enclaves map $\mathcal{M}$, the associated sealing predicates are registered for the enclave's otypes.

Enclave clients that have obtained a $\text{tidx} \mapsto_{\text{E}}^{\square} I$ resource for the enclave owning a given otype o , may use the enclave identity invariant to learn that $\text{seal_pred}(o)(\mathcal{M}(I).P_{enc})$, stipulating that safe capabilities sealed with o necessarily satisfy the predicates associated with I . Usually, the sealed capabilities that the client uses have to be safe, because the adversary acts as intermediary between the enclave and the client. The case studies in Section 5 explain how this is used in practical examples.

4.5 Reasoning about the Adversary with the Universal Contract

The `EStoreId` instruction, the enclave identity invariant, and $\text{tidx} \mapsto_{\text{E}}^{\square} I$ provide enclave clients with extra information they can rely on when receiving sealed values from untrusted code. However,

this information comes at the cost of an additional proof obligation when invoking the adversary. In particular, we need to prove that the enclaves of interest registered in the custom enclaves map, actually respect the sealing predicates we registered for them.

This condition on registered enclaves is expressed as *custom_enclaves_contract* in Figure 18. This contract essentially requires that when an enclave is initialized by adversarial code and its identity matches that of an enclave in the custom enclaves map, then it must respect the sealing predicates registered for it. More technically, the *einit* instruction does not invoke the initialized enclave itself, but instead creates a sentry capability for it, with exclusive access to the sealing capability for the enclave's otypes. The custom enclaves contract essentially describes the state of the enclave at that point and requires that it is safe to share with the adversary: the code region contains the capability pointing to the data region, followed by the instructions themselves (1), the data region contains the sealing capability containing the encryption and signing otypes, followed by the content of the data (2). Importantly, the enclave receives exclusive ownership of this memory in the form of points-to-predicates. The sealing predicates associated with the enclave's keys correspond to those found in *custom_enclaves_contract* ((3) - (4)). Additionally, the enclave receives the *tidx* $\mapsto_E I$ resource representing authority to deinitialize itself (5). Proving this contract amounts to showing that the enclave only ever seals values that satisfy the predicates defined in $\mathcal{M}$, and that it does not leak its private keys to untrusted parties.

Apart from this assumption about the registered enclaves, the Ceriesier universal contract is similar to the Cerise one, described in Theorem 4.1.

THEOREM 4.1 (CERISIER UNIVERSAL CONTRACT). *Let $\mathcal{M}$ be a map of custom enclaves, such that *custom_enclaves_contract*($\mathcal{M}$) holds. Then, $\boxed{\mathcal{I}(\mathcal{M})} \vdash \left\{ \bigstar_{r \in \text{RegName}} (\exists w. r \Rightarrow w * \mathcal{V}(w)) \right\} \rightsquigarrow \bullet$.*

The contract captures all possible behaviors of arbitrary, adversary code, obeying the guarantees provided by the capability machine, and thus captures capability and attestation safety. It is a *universal contract* [24] since it has no assumptions about the particular code that is executed, i.e., it applies to arbitrary code. As such, the contract can be used to reason about invocations of untrusted adversarial code.

Notice that proving the contract *custom_enclaves_contract* typically requires recursively invoking the universal contract, for example, to reason about the enclave returning control to its untrusted caller. Fortunately, this recursion can be resolved by proving the contract using Löb induction. This proof principle, inherited from Iris, allows proving a predicate P by showing that $\triangleright P \vdash P$ [26], i.e., to prove P , one may assume that P holds at later logical steps.

4.6 Proving the Universal Contract

The original Cerise universal contract is proven by Löb induction and case analysis on the first instruction executed by the adversary. For every instruction in the ISA, one shows that the necessary authority follows from the safety of capabilities that are available. Furthermore, one proves that the machine state after executing this instruction is also safe, which is done in one of two ways. In most cases, it suffices to prove that the resulting machine state still contains only safe-to-share values and invoke the Löb induction hypothesis.

However, in cases where control is potentially passed to non-adversarial code, e.g., when invoking a sentry capability, we may reach a state where not all capabilities are known to be safe to share. In such cases, we require another argument to prove the safety of the resulting state. After invoking a sentry capability, it is converted into an rx capability that may not be safe to share (see Figure 5, in particular, the *jmp* instruction). However, in this case, we can appeal to the safe-to-execute fact for this rx capability that we get from the safety of the original ϵ capability (see Figure 8).

The Cerisier proof of the universal contract proceeds along similar lines, adding cases for all the new instructions. Cases for `getwtype`, `getotype`, `estoreid`, `hash`, and `hashconcat` are trivial, since they only produce integers and do not modify system state. The cases for `cseal` and `cunseal` appeal to the `safe_to_seal` and `safe_to_unseal` properties that we learn from the safety of the sealing capabilities (see Figure 8) to justify that the sealed or unsealed capabilities produced by the instruction remain safe. Similarly, the case for `edeinit` relies on the authority to deinitialize an enclave that is included in safety of the sealing capability (see Figure 8). For brevity, we do not explain the case for `isunique` which involves dealing with the revocation machinery from Section 3.4.

By far the most interesting case is the `einit` instruction, essentially because it creates a new situation where control is passed from the adversary to potentially trusted code. The operational semantics provides us with a measured identity, a new table entry and a sealing capability for two fresh otypes. For the fresh otypes, we can register a sealing predicate of choice. In the case where the measured identity I is not present in the custom enclaves map $\mathcal{M}$, we can take $\mathcal{V}$ as sealing predicates and store the enclave ownership predicate in a suitable invariant. This amounts to treating the new enclave as untrusted, and indeed, we can then prove all produced capabilities safe and appeal to the induction hypothesis.

However, if the measured identity is present in $\mathcal{M}$, the invariant $\mathcal{I}(\mathcal{M})$ forces us to register the corresponding sealing predicates (rather than $\mathcal{V}$), and the new sealing capability cannot be considered safe-to-share. It is in this situation a new form of security-boundary-crossing appears, as the untrusted adversarial code metamorphoses into a trusted enclave. Fortunately, we can make use of *custom_enclaves_contract*, which tells us that the freshly defined enclave is safe, at least if we can satisfy the preconditions from Figure 18.

Many of these preconditions simply describe the resulting machine state after `einit` and are hence easy to satisfy. However, the points-to predicates for the code and memory region need to be created freshly, intuitively by revoking pre-existing authority over the corresponding memory. For every address a in these regions, the memory revocation infrastructure from Section 3.4 allows us to do this by bumping the current version v of a to $v + 1$, allocating $(a, v + 1)$ as a fresh logical address and reestablishing correspondence of physical and logical memory by designating $(a, v + 1)$ as current. The memory sweep's positive result provides the necessary evidence that no more references to (a, v) exist in the system.

4.7 Adequacy

Our specifications are written and proven in the Cerisier program logic. To draw end-to-end conclusions about executions of our case studies in terms of only the operational semantics, we prove an *adequacy theorem*.

THEOREM 4.2 (END-TO-END THEOREM FOR ENCLAVE ATTESTATION — INFORMAL). *Suppose a good initial machine state with an initialized client and an (arbitrary) adversary, but no enclave. Further suppose that for some custom enclave map $\mathcal{M}$, the specification $\mathcal{I}(\mathcal{M}) \vdash \{P\} \rightsquigarrow \bullet$ holds, where P corresponds to the interpretation of the initial state in the program logic.*

Then, the assert in the client program will never fail.

We provide the formal version of Theorem 4.2 in the Supplemental Material, under Appendix G.

When proving the end-to-end properties of our case studies Section 5, we always assume a good initial state (which can be obtained, e.g., after booting the machine), and the program logic specification is then proved with a $\mathcal{M}$ carefully chosen. When the program interacts with untrusted code, the program logic specification requires proving that *custom_enclaves_contract*($\mathcal{M}$) holds for all registered custom enclaves in $\mathcal{M}$.

5 Case Studies

In this section, we present case studies of three systems relying on enclaves, implemented using Cerisier features and verified using the Cerisier program logic. Two case studies represent trusted execution applications: secure outsourced computation (SOC, Section 5.1) and trusted sensors in a stateful enclave (Section 5.3). Both model representative applications of well-known trusted execution applications, studied in the systems literature by, for example, Barbosa et al. [4], Li et al. [27] for the former, and Hu et al. [22], Shepherd et al. [38] for the latter. Our third case study concerns mutual attestation, where two mutually distrusting enclaves establish trust in each other. To avoid circularity, such enclaves' identities cannot include the identity that they use to attest each other. In the case study, we model *MAGE*: a technique proposed by Chen and Zhang [10] to resolve the circularity by attesting enclave code separately from an identity table used to attest other enclaves (Section 5.2).

5.1 Verifying the SOC Example

The correctness of the SOC example from Figure 15 follows from our adequacy result (Theorem 4.2) by instantiating the client's program with the SOC client and proving the Cerisier specification. The derived property guarantees that the assertions never fail, we can thus conclude that the unsealed result is 42. In a bit more detail, our proof follows the approach sketched in Section 2.

First, we instantiate the custom enclave map $\mathcal{M}$ to contain a single entry that maps the SOC enclave's static identity ID_{SOC} to the signing predicate $\lambda w. w = 42$. We need to prove that the property associated with the enclave identity holds, formally by proving that *custom_enclaves_contract*($\mathcal{M}$) holds. In essence, we must argue that the enclave does not leak its private signing key, and that any values signed equal 42.

Then, we need to prove the specification of the SOC client. When the client calls the adversary, we invoke the universal contract Theorem 4.1. When the adversary yields to the known client, we expect to receive a sealed result $\{r\}_{ot}$ and the corresponding unsealing key $[u, ot, ot + 1, ot]$. Moreover, $\mathcal{V}(\{r\}_{ot})$ holds because $\{r\}_{ot}$ was passed by the adversary. For the call to *estoreid*, we apply *EStoreIdAttestValid*. In the success case, we obtain the enclave resource $tidx \mapsto_E^\square I$, for the table index $tidx$ linked to the otype ot , for some identity I . If $I = ID_{SOC}$, the system invariant combined with our *custom_enclaves_predicates* and $tidx \mapsto_E^\square I$ yields $seal_pred(ot)(\lambda w. w = 42)$. By combining $seal_pred(ot)(\lambda w. w = 42)$ and $\mathcal{V}(\{r\}_{ot})$, we know that $r = 42$. After unsealing, we can thus safely conclude that the assert holds.

5.2 Mutual Attestation

Naively attesting two (or more) enclaves with each other induces a circularity problem since they must each contain a table of static identities for the other enclave(s). Nevertheless, many applications require a mutual attest to establish secure communication. Chen and Zhang [10] propose a solution to break the circularity. First, all enclaves involved in the mutual attestation compute a *pre-identity* which hashes only the code, not the identity table. These pre-identities are gathered in an *identity table*, that is located in memory after the enclaves' code. When an enclave E_1 wants to attest an enclave E_2 , it can no longer simply look up the expected identity of E_2 in its identity table, only the pre-identity. However, E_1 can still reconstruct E_2 's identity at runtime by combining E_2 's pre-identity with its own copy of the pre-identity table (which should be equal to E_2 's copy). Concretely, it will hash its pre-identity table and combine the result with E_2 's pre-identity. The result is the identity of E_2 and can be used to attest E_2 . Note that this method requires some algebraic assumptions about hash, i.e., hashing the concatenation of the memory contents is equal to concatenating the hash of each, separately: $\text{hash}(m_1 ++ m_2) = \text{hash}(m_1) \parallel \text{hash}(m_2)$.

Our mutual attestation case study models this proposed MAGE protocol. The full code can be found in the Rocq artifact, and simplified pseudocode is provided in the Supplemental Materials, under Appendix C. Two enclaves A and B, compute 42 and 43, respectively, and attest each other. When A executes, it signs 42, and yields to the adversary. It expects the adversary to call B, passing the signed integer. When B is executed, it expects the adversary to pass the signed value from A. B then attests A, from which it learns that the signed value must be 42. Now, B signs 43, and yields to the adversary. When the adversary then executes A, it expects the signed value from B. A finally attests B, and learns that the signed value was 43. Mutual attestation is thus established.

Correctness is analogous to the SOC example Section 5.1. In addition, for the enclaves to mutually attest each other, *custom_enclaves_contract* holds for enclave A (resp. B) iff it holds for enclave B (resp. A). This follows from the Löb induction principle.

5.3 Trusted Sensor Readout

The previous two examples only read from memory without writing to it. We now consider an example that writes to its data section. The example is an adaptation of the trusted sensor readout example in CHERI-TrEE [43]. The code can be found in the Supplemental Materials, under Appendix D. Since we do not model I/O, a memory read acts as a stand-in for reading from an MMIO sensor. The setup is as follows: the adversary creates a reader enclave that reads from the sensor address, while a separate transformer enclave processes the raw sensor data. The transformer enclave attests the reader enclave. Furthermore, a verifier can then attest the transformer enclave to be assured that the values it receives are indeed processed sensor readings. For both enclaves, we use the initial entry points, created after the adversary executes *einit*, to initialize private state. We create another sentry to serve as a second entry point.

The initialization of the reader enclave expects a capability (*rw*, *mmio_b*, *mmio_e*, *mmio_a*) to access the sensor. After ensuring exclusive access with *isunique*, the reader initializes the sensor (here modeled by writing 21 to the sensor's address), and stores the sensor capability in its data section after the sealing keys. It then creates a sentry capability to a *read* entry point in its own code section, signs this sentry, and returns it with the public signing key to the adversary. The *read* entry point fetches the sensor capability from its data, and loads and returns the value at address *mmio_a*.

The initialization of the transformer enclave expects the sealed sentry to the sensor's *read* entry point and the sensor's public signing key which it uses to attest the sensor enclave. Like the reader enclave, it stores the sentry in its data section and returns a new sentry to a *use* entry point. This second entry point calls the *read* entry point of the sensor and processes the value (multiply by 2) before returning. Finally, the main program expects a signed sentry to the transformer enclave's *use* entry point, which it attests, and asserts that it returns 42.

During the proof, invariants specify the integrity of the data sections. For instance, after initialization the sensor enclave's data contains a RW-capability pointing to 21. However, this invariant must be allocated during *einit* before any code runs. Hence, we make use of a one-shot update to stipulate in a single invariant the shape of the data section before and after calling the enclave. The second entry point *read*, has persistent knowledge that the data section has been initialized.

The sensor enclave only ever signs the *read* entry point, but in the sealing predicate we specify a triple that must hold for the capability. Furthermore, we must show that the sealed sentry capability to *read* is safe to share with the adversary, i.e., that it satisfies the sealing predicate, or in other words, that it satisfies its triple. The transformer enclave is verified similarly to the sensor enclave, and the verification of the main program is also analogous to the SOC example in Section 5.1.

6 Discussion

6.1 Implementing CHERI-TrEE Primitives and Exclusive Memory Ownership

The CHERI-TrEE authors [43] implement the primitives in hardware as an ISA extension to the open-source RISC-V Proteus processor [7] and describe how the memory sweep integrates into the processor's 5-stage instruction pipeline. They also implement the primitives in software on the ARM Morello [3] capability machine as a trusted hypervisor component providing the primitives as hypercalls. Both implementations perform an un-optimized eager memory sweep and in this paper, we follow suit to keep verification simple.

Even though this is costly in practice, the CHERI-TrEE authors [43] envision it to be acceptable in certain embedded system scenarios. They also point to work on reducing the cost of the sweep [25] and how it could be run concurrently on multi-core systems [14]. The exclusive memory ownership guarantees can also be implemented with other schemes. For instance, CheriOS [14] lets a trusted nanokernel manage and hand out *reservations*, from which one can obtain guaranteed exclusively-owned memory. It uses both a (fast) virtual-memory-remapping and a (slow) GC-like mechanism (similar to CHERIoT's [2] and Cornucopia [16, 17]) to reclaim memory.

6.2 Reasoning about Attestation Generally

Our technical development involves several non-trivial techniques not previously present in Cerise, e.g., for reasoning about sealed capabilities and memory revocation. Nevertheless, the main innovation is our approach to reasoning about attestation, discussed in Sections 4.3-4.5. The core idea is to treat attestation formally as an additional form of control transfer from untrusted to trusted code.

This core idea is visible in the universal contract (see Section 4.5), which justifies safety of a control transfer to untrusted code, provided any control transfer from the adversary back to trusted code is safe. The contract of custom enclaves in Figure 18 models exactly that requirement for the transfer that conceptually happens from untrusted to trusted code when an enclave is freshly attested: safety of the callback produced by `einit` must follow from ownership of the authority the attested enclave obtains (particularly exclusive memory ownership and access to enclave attestation seals). This is similar to the "safe to share"-requirement $\mathcal{V}(E, b, e, a)$ for an enter capability in Figure 8, since an enter capability lets the adversary transfer control back to trusted code, and this transfer must be proven safe when invoking the universal contract. Huyghebaert et al. [24] have shown that the control transfers of ISA security primitives in non-capability-machine settings (particularly RISC-V PMP ecalls and interrupts) can be treated similarly.

Reasoning about other systems offering attestation will require a program logic that can express the guarantees offered by underlying mechanisms (e.g. the lazy GC-like mechanism of Esswood [14] or SGX-like lookaside tables [8, 11, 31]). Insofar as those techniques involve a form of revocation of memory ownership, we expect that one should be able to adapt the logical memory technique of Hur et al. [23]. After defining a suitable program logic, a universal contract should be defined, proven and applied, which has assumptions about the control transfer from untrusted code to freshly-attested trusted enclaves, formalized with a form of contract for custom enclaves. We believe that our approach generalizes to other attestation systems, because any attestation system involves a form of control transfer from untrusted code to attested code.

7 Related Work

We have discussed the most closely related work along the way. In this section, we discuss other related work; we restrict ourselves to work that considers verification of low-level software using trusted computing primitives (not just software that implements them) and verification of capability machine security properties.

Some other work verifies architectural and micro-architectural properties about models of trusted execution environments (TEE) (micro-)architectures or their implementation in hardware and firmware. However, none of these papers can be used to verify end-to-end results about TEE enclaves (or equivalent) and their client(s) in the presence of arbitrary adversary code. Witharana et al. [46] use model-checking to verify some security properties about a model of Intel TDX (micro-)architectures. Nunes et al. [32] use a combination of verification techniques to verify hardware and software components of the VRASED TEE system, but Bognar et al. [6] later discovered various errors in their verification approach. They did not aim to prove end-to-end results about software relying on attestation. Ferraiuolo et al. [15] use Dafny to prove information flow properties about enclaves in their Komodo system, but as other mentioned work and unlike our work, they do not aim to prove that application software uses the TEE primitives correctly. Although such work strengthens confidence in correctness of TEE implementations, the authors do not consider how to effectively reason about software making use of TEE primitives.

The Cerise capability machine and program logic has been discussed and extended with various features [18–21, 42] but no attestation primitives have been considered until now. Other authors have used different approaches for proving weaker properties on larger ISAs [5, 30].

8 Conclusion and Future Work

We have presented Cerisier, the first program logic for modular reasoning about trusted, untrusted, and attested code. We have proved Cerisier sound with respect to a novel operational semantics of a capability machine model with primitives for attestation and sealing, and we have used Cerisier for defining a universal contract, which captures both capability safety and local enclave attestation. The universal contract is formalized using a novel kind of parameterized logical relation. Finally, we have used Cerisier to prove non-trivial end-to-end properties of representative examples of trusted computing applications relying on local attestation. Our work is fully mechanized in Rocq, comprising ~35k LoC, including 2k LoP for the proof of the FTLR case for `einit` sketched in Section 4.6. We believe our formal techniques capture intuitive reasoning of trusted execution systems, and thus that our techniques can inform the design of program logics for other trusted execution systems. There are several interesting other lines of future work.

Remote attestation is an important feature of trusted computing that Cerisier does not support. This would require to change our operational semantics and program logic, introducing either a probabilistic [see, e.g., 1] or symbolic [see, e.g., 41] model of cryptography. The resulting changes would be important, but orthogonal to the contribution of this paper.

Similarly to trusted computing, confidential computing derives confidentiality guarantees about the systems' data and I/O behavior. We believe that Cerisier's reasoning techniques extend to confidential computing, and could be based on Cerise's existing binary program logic, which has previously been demonstrated to support confidentiality results.

Finally, it is worth noting that Cerisier currently does not support I/O, but support for memory-mapped I/O (MMIO) [40] would enable *trusted* and *confidential* I/O, i.e., establishing trust in enclaves performing I/O. The trusted sensor implementation from Section 5.3 could then be adapted to read from an external sensor rather than from memory.

Acknowledgments

We thank the anonymous PLDI reviewers for their valuable feedback and suggestions. Denis Carnier holds a PhD fellowship (1S46525N) of the Research Foundation – Flanders (FWO). This work was supported in part by a Villum Investigator grant (no. 25804 and VIL73403), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This research is partially funded by the Internal Funds KU Leuven, the Cybersecurity Research Program Flanders, and by ERC grant

(UniversalContracts, 101040088). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Data-Availability Statement

Our results are publicly available [35, 36]. The artifact contains the mechanization in Rocq of all definitions, case studies, theorems and proofs discussed in the paper. The Supplemental Material is also publicly available [34].

References

- [1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–15. doi:10.1109/CSF51468.2021.00048
- [2] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (Toronto, ON, Canada) (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 641–653. doi:10.1145/3613424.3614266
- [3] ARM Ltd. 2022. *Arm Architecture Reference Manual Supplement Morello for A-Profile Architecture*. ARM Ltd. <https://documentation-service.arm.com/static/61e577e1b691546d37bd38a0?token= Document number DDI0606>.
- [4] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. 2016. Foundations of Hardware-Based Attested Computation and Application to SGX. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 245–260. doi:10.1109/EUROSP.2016.28
- [5] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.). Springer, 174–203. doi:10.1007/978-3-030-99336-8_7
- [6] Marton Bognar, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1638–1655. doi:10.1109/SP46214.2022.9833735
- [7] Marton Bognar, Job Noorman, and Frank Piessens. 2023. Proteus: An Extensible RISC-V Core for Hardware Extensions. In *RISC-V Summit Europe '23*.
- [8] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 34:1–34:6. doi:10.1145/2744769.2744922
- [9] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 319–327. doi:10.1145/195473.195579
- [10] Guoxing Chen and Yinqian Zhang. 2022. MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4095–4110. <https://www.usenix.org/conference/usenixsecurity22/presentation/chen-guoxing>
- [11] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. Technical Report 086. <https://eprint.iacr.org/2016/086>
- [12] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. doi:10.1145/365230.365252
- [13] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. 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*. IEEE, 147–162. doi:10.1109/EUROSP.2016.22
- [14] Lawrence Esswood. 2021. *CheriOS: Designing an Untrusted Single-Address-Space Capability Operating System Utilising Capability Hardware and a Minimal Hypervisor*. Ph.D. Dissertation. University of Cambridge. <https://www.repository.cam.ac.uk/handle/1810/326715>

- [15] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Symposium on Operating Systems Principles (SOSP '17)*. ACM, 287–305. doi:10.1145/3132747.3132782
- [16] 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 D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 608–625. doi:10.1109/SP40000.2020.00098
- [17] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (La Jolla, CA, USA) (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 251–268. doi:10.1145/3620665.3640416
- [18] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (Feb. 2024), 1–59. doi:10.1145/3623510 <https://dl.acm.org/doi/10.1145/3623510>.
- [19] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 6:1–6:30. doi:10.1145/3434287
- [20] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 74:1–74:30. doi:10.1145/3527318
- [21] Angus Hammond, Ricardo Almeida, Thomas Bauereiss, Brian Campbell, Ian Stark, and Peter Sewell. 2025. Morello-Cerise: A Proof of Strong Encapsulation for the Arm Morello Capability Hardware Architecture. *Proc. ACM Program. Lang.* 9, PLDI (June 2025), 226:1961–226:1983. doi:10.1145/3729329
- [22] Shengtuo Hu, Qi Alfred Chen, Jiwon Jeong, Can Carlak, Yiheng Feng, Z. Morley Mao, and Henry X. Liu. 2020. CVShield: Guarding Sensor Data in Connected Vehicle with Trusted Execution Environment. In *Proceedings of the Second ACM Workshop on Automotive and Aerial Vehicle Security (AutoSec '20)*. Association for Computing Machinery, 1–4. doi:10.1145/3375706.3380552
- [23] Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2011. Separation Logic in the Presence of Garbage Collection. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, Toronto, ON, Canada, 247–256. doi:10.1109/LICS.2011.46 <http://ieeexplore.ieee.org/document/5970244/>.
- [24] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. 2023. Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2083–2097. doi:10.1145/3576915.3616602
- [25] Alexandre Joannou, Jonathan Woodruff, Robert Kovácsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N.M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*. 641–648. doi:10.1109/ICCD.2017.112
- [26] Ralf Jung, Robbert Krebbers, Jacques-henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. doi:10.1017/S0956796818000151
- [27] Xiaoguo Li, Bowen Zhao, Guomin Yang, Tao Xiang, Jian Weng, and Robert H. Deng. 2023. A Survey of Secure Computation Using Trusted Execution Environments. arXiv:2302.12150 doi:10.48550/ARXIV.2302.12150
- [28] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2018. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* 67, 3 (March 2018), 361–374. doi:10.1109/TC.2017.2647955
- [29] James H. Morris, Jr. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21. doi:10.1145/361932.361937
- [30] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. 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. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1003–1020. doi:10.1109/SP40000.2020.00055

- [31] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security* 20, 3 (July 2017), 7:1–7:33. doi:10.1145/3079763
- [32] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1429–1446. <https://www.usenix.org/conference/usenixsecurity19/presentation/de-oliveira-nunes>
- [33] Andrew M. Pitts. 2010. Step-Indexed Biorthogonality: a Tutorial Example. In *Modelling, Controlling and Reasoning About State, 29.08. - 03.09.2010 (Dagstuhl Seminar Proceedings, Vol. 10351)*, Amal Ahmed, Nick Benton, Lars Birkedal, and Martin Hofmann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. <http://drops.dagstuhl.de/opus/volltexte/2010/2806/>
- [34] June Rousseau, Denis Carnier, Thomas Van Strydonck, Steven Keuchel, Dominique Devriese, and Lars Birkedal. 2026. Cerisier: A Program Logic for Attestation in a Capability Machine. arXiv:2604.13638 doi:10.48550/arXiv.2604.13638
- [35] June Rousseau, Denis Carnier, Thomas Van Strydonck, Steven Keuchel, Dominique Devriese, and Lars Birkedal. 2026. Cerisier - Code repository. <https://github.com/logsem/cerisier>
- [36] June Rousseau, Denis Carnier, Thomas Van Strydonck, Steven Keuchel, Dominique Devriese, and Lars Birkedal. 2026. Cerisier: A Program Logic for Attestation in a Capability Machine (Artifact). doi:10.5281/zenodo.19062990
- [37] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. *IEEE Access* 9 (2021). doi:10.1109/ACCESS.2021.3087421
- [38] Carlton Shepherd, Raja Naeem Akram, and Konstantinos Markantonakis. 2017. Establishing Mutually Trusted Channels for Remote Sensing Devices with Trusted Execution Environments. In *ARES (ARES '17)*. Association for Computing Machinery, 1–10. doi:10.1145/3098954.3098971
- [39] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2021. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Journal of Functional Programming* 31 (2021), e9. doi:10.1017/S095679682100006X
- [40] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. 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*. IEEE, 80–95. doi:10.1109/CSF54842.2022.9919645
- [41] Eijiro Sumii and Benjamin C. Pierce. 2003. Logical relations for encryption. *Journal of Computer Security* 11, 4 (2003), 521–554. doi:10.3233/JCS-2003-11403
- [42] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. Proving Full-System Security Properties under Multiple Attacker Models on Capability Machines. In *IEEE Computer Security Foundations Symposium (CSF)*. 80–95. doi:10.1109/CSF54842.2022.9919645
- [43] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. 2023. CHERI-TrEE: Flexible Enclaves on Capability Machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, Delft, Netherlands, 1143–1159. doi:10.1109/EuroSP57164.2023.00070 <https://ieeexplore.ieee.org/document/10190507/>.
- [44] 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, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Marketos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report. Computer Laboratory, University of Cambridge. 523 pages pages. doi:10.48456/TR-987
- [45] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 20–37. doi:10.1109/SP.2015.9
- [46] Hasini Witharana, Hansika Weerasena, and Prabhat Mishra. 2024. Formal Verification of Virtualization-Based Trusted Execution Environments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (Nov. 2024), 4262–4273. doi:10.1109/TCAD.2024.3443008
- [47] Sara Zain, Jannik Mähn, Stefan Köpsell, and Sebastian Ertel. 2025. Formally-Verified Security against Forgery of Remote Attestation Using SSProve. arXiv:2502.17653 [cs] doi:10.48550/arXiv.2502.17653

Received 2025-11-12; accepted 2026-04-03