Program verification on a capability machine in presence of untrusted code

Aïna Linn Georges[1], Armaël Guéneau[1], Thomas Van Strydonck[2], Amin Timany[1], Alix Trieu[1], Sander Huyghebaert[3], Dominique Devriese[3], and Lars Birkedal[1]

[1] Aarhus University, Danemark [2] KU Leuven, Belgique [3] Vrije Universiteit Brussel, Belgique

Abstract

A capability machine is a type of CPU allowing fine-grained priviledge separation using *capabilities*, machine words that represent a certain authority. In this article, we present a method that allows verifying the functional correctness of programs running on the machine, even if these call or are called by unknown (and possibly malicious) code. Such programs typically rely on the appropriate use of capabilities in order to behave as expected. From a logical point of view, our approach allows leveraging the guarantees provided by the machine, in order to formally reason about programs. The key aspects of the approach are first the definition of a program logic, and then a logical relation, which—we show—provides a specification for unknown code. The entire setup has been mechanized in Coq.

The aforementionned methodology underlies the previous work of the authors on formally reasoning about a secure calling convention [GGVS+21], but was left somewhat implicit. This work tries to present a pedagogical introduction to the methodology, in a simpler setting (no exotic capabilities), and using a minimal example.

1 Introduction

A capability machine is a type of CPU which enables fine-grained memory compartimentalization and priviledge separation through the use of *capabilities*. This type of hardware architecture has been studied since the 60s [DVH66, Lev84], and in particular more recently as part of the CHERI project [WNW+19]. The capability machine considered in this paper intends to be a simplified model of a machine from the CHERI family[1].

A capability is a value associated with some authority, allowing for instance to access a given memory region or to interact with a component of the system. In a capability machine, capabilities are represented as machine words, and can thus be stored in registers or memory. The machine guarantees the integrity of capabilities: by design, it must be impossible for a program to forge capabilities giving it more authority than it had previously. Independently from the concrete machine representation, in this paper, we model capabilities as tuples (p, b, e, a), where p represents a permission, and b, e, a memory addresses, where $[b, e)$ corresponds to the range of authority of the capability, and a is typically in the range.

There are several kinds of capabilities; we are interested here in the two kinds that are most common in CHERI. *Memory capabilities* give access to a contiguous range of memory $[b, e)$ with permission p (e.g. RW or RX). These capabilities are usable as a pointer whose bounds and permissions are automatically checked by the hardware. Given a memory capability, it is possible to derive new capabilities with restricted authority, either by restricting its permission (using the `restrict` instruction), or by restricting its range (using `subseg`), as illustrated in Figure 1.

[1]The main difference being our use of “enter” capabilities instead of CHERI’s “sealed” capabilities.

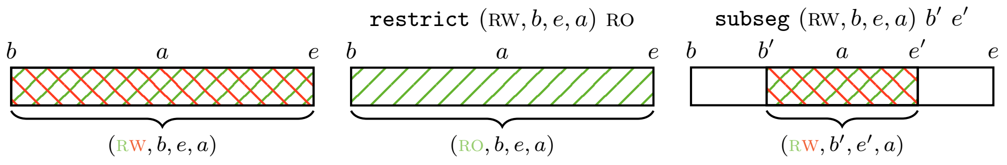

Figure 1: Behavior of `restrict` and `subseg`.

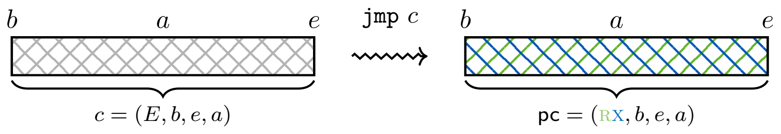

Figure 2: Behavior of `jmp` on enter capabilities.

Object capabilities provide a mechanism similar to closures in high-level languages. An object capability (associated with the "enter" permission or E) provides the authority to invoke some component, but without giving access to the private capabilities this component relies on to run. Invoking the capability (using the `jmp` instruction) gives control to the component, and changes the permission of the capability from E to RX (Figure 2). This gives access to the code and data associated to its range, which were previously inaccessible. In CHERI, object capabilities correspond to a specific mode of use of sealed capabilities, where a pair of code and data capabilities are "sealed" together and can only be accessed through a `CCall` instruction [WNW+16]. Our use of enter capabilities comes from the M-Machine [CKD94] and is conceptually similar, although a bit simpler.

Capabilities thus allow a piece of code to securely interact with an untrusted third-party, by restricting the set of capabilities the untrusted code has access to. In a system composed of mutually untrusted components (which might even contain malicious code), capabilities provide a way of enforcing that the overall system nevertheless satisfies some security properties. This is achieved by carefully initializing the components that we do trust, ensuring they properly interact with the components that we do not trust, and leveraging the capability checks performed by the machine at every step.

The question is then: which security properties can we actually enforce using these hardware capabilities? And can we formally prove that these properties do indeed hold? In other words, how do we leverage the checks of the machine in order to formally reason about the interaction of a known program with untrusted code?

This paper proposes the following answer. Any program calling—or being called by—unknown code can protect the access to some capabilities and memory cells by using the encapsulation properties provided by object capabilities. We then say that this protected data constitutes the "private state" of the program, on which it can establish and maintain some properties, provided it has been properly isolated from unknown code. For any property of the private state that we wish to guarantee, it is then enough to check: 1) that this property is an *invariant* of the program (it holds initially and is preserved through its execution), and 2) that the overall program specification guarantees that it "properly encapsulates" its private state (e.g. does not inadvertently leak it to unknown code). Then, the capability machine enforcement mechanisms guarantee that this invariant is preserved through the execution of the whole system, for any unknown code linked against the known, verified program.

More precisely, the key aspects of this methodology are as follows:

- We define a program logic making it possible to formally verify the correctness of programs running on the capability machine. We leverage the Iris framework [JKJ+18], which provides an expressive separation logic with powerful reasoning principles, in particular the notion of logical invariant (Section 3).
- We define, using our program logic, the specification of what a "safe" capability and a "safe" program is. A capability (respectively, a program) is "safe" if it cannot be used to invalidate an invariant previously established at the logical level. Then, a capability that is safe can be shared freely with unknown code. This definition can be seen as a unary logical relation characterizing our notion of "capability safety" (Section 4).
- We show (and this is our main theorem) that, if an arbitrary program has only access to "safe" values, then running the program is itself "safe". This is a global property of the capability machine, expressing that it "works properly": it is not possible to increase one's authority more than what was available initially, independently of the sequence of instructions that one executes (Section 4).
- The last piece of the puzzle is the theorem relating invariants established in the program logic to the operational semantics of the machine (Section 3). Given a concrete scenario (typically, a complete system mixing known, verified code with unknown, untrusted code), this makes it possible to eventually obtain a theorem about the execution of the system that only depends on the operational semantics of the machine.

The goal of this paper is to illustrate this methodology by applying it to the verification of a very simple system. In Section 2, we introduce the example system that we consider, then detail its proof in Section 5, after the necessary reasoning principles have been introduced. The results and examples presented here have been fully formalized in Coq, and are available online: `https://github.com/logsem/cerise`.

2 Motivation

We consider a simple scenario: we verify the correctness of a known component interacting with an "adversary" component whose code is unknown and untrusted.

Let us start by reasoning in the setting of a high-level language with references and first-class functions (we use an ML-like syntax, but the same example would work in e.g. Javascript).

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda n.\ \text{if } n \geq 0 \text{ then } x := !x + n)$$

What can one say about the program above? The program allocates a new reference x initialized to 0. It then creates a function closure that allows increasing the reference's value by adding the integer n received as argument, as long as it is positive. This function closure is then returned by the program to its surrounding context. We can then expect that, whatever the way the (possibly malicious) context uses the closure, the value stored in x will always be non-negative. And indeed, this property can be made precise and proved formally [SGD17].

In essence, this property holds in any high-level language that forbids inspecting function closures. Then, only having access to the closure produced by the program does not give access to x. From the context perspective, the only possible action is to call the closure. In other words, the closure properly "encapsulates" the private state of the program (the reference x).

Does the property still hold ("x contains a positive integer at every step") if the closure is passed to an adversary context implemented in machine code, instead of high-level code? The answer is no: the surrounding context can simply forge a pointer to x and use it to write a negative integer in the reference, thus invalidating the property.

However, on a capability machine, it is possible to preserve this property! This can be achieved through careful use of object capabilities, which would be leveraged by a low-level equivalent of the program above so as to protect its private state. Then, it becomes possible to pass control to an arbitrary context, implemented in machine code, without it being able to modify the private reference.

In the rest of this paper, we detail how the property "x always contains a positive integer" can be verified formally, for a machine-code version of the program above, interacting with an unknown component implemented in machine code. The concrete implementation of the verified program appears later in Figure 4, and its proof will be detailed in Section 5. In the following sections, we define the two key reasoning principles that are required for the proof. First, a program logic for our capability machine, that allows verifying the correctness of known code. Second, a logical relation and its fundamental theorem, which give a "universal specification" of unknown code.

3 Program logic

We define a program logic that allows reasoning in a modular fashion on code running on the machine. More precisely, we define a separation logic by instantiating the Iris [JKJ+18] framework with the operational semantics of our capability machine. For conciseness, we do not give here the details of the operational semantics—they are somewhat verbose and unsurprising.

One noteworthy aspect of the operational semantics is its handling of errors: when a capability check fails (e.g. when a program tries to access a capability outside of its range of authority), the semantics does not "get stuck". Instead, it explicitly transitions into a "failed" state. In the program logic, we then establish specifications which do allow the program to fail during the execution. Indeed, from a security point of view, making the machine fail is in fact secure! The property that we wish to guarantee states that some invariant of the code is preserved through the execution. In that setting, having the machine fail is first, fine (invariants are trivially preserved when the machine is stuck in the failure state) and second, an eventuality that we have account for in any case (we cannot prevent the unknown code from failing some capability check).

Our program logic features the standard connectives of separation logic, including the separating conjunction ($*$) and the magic wand ($-\!\!*$, read as an implication). The assertion $\mathsf{a} \mapsto w$ expresses the ownership of a memory cell at address a and containing the machine word w. The assertion $r \Mapsto w$ denotes the ownership of a CPU register r containing w. A machine word w is either an integer or a capability. We write $\vec{a} \mapsto \vec{l}$ for the ownership of contiguous memory cells at addresses $\vec{a}$ containing $\vec{l}$. The program counter (the register containing a capability pointing to the code currently executed) is written pc.

A key feature of the logic is the notion of logical invariant (directly inherited from Iris). The (persistent) assertion $\boxed{P}$ expresses that the property P holds, and will continue to hold for every future step of the execution. The associated proof rules are standard and have been inherited from Iris.

Our notion of program specification is less standard, owning to the fact that we are reasoning about low-level machine code rather than high-level programs. We distinguish three types of program specifications:

$$\begin{array}{ll} \{w; P\} \rightsquigarrow \bullet & \text{complete execution} \\ \{w_0; P\} \rightsquigarrow \{w_1; Q\} & \text{code fragment} \\ \langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle & \text{single instruction} \end{array}$$

In each case, w, w_0 or w_1 denotes the value of the program counter (pc), and P and Q are assertions describing the state of the machine (registers and memory). The program counter typically stores a capability with permission RX, pointing to a memory region containing integers corresponding to the code of the program. These are then decoded during the execution by the machine into machine instructions (trying to decode a capability always fails).

We have $\{w; P\} \rightsquigarrow \bullet$ if, starting from a machine state satisfying P and with pc equal to w, then the machine runs until completion (possibly in the "fail" state) or loops forever. (This requires that the resources in P related to registers and memory are enough for the execution: we do not have $\{w; \mathsf{True}\} \rightsquigarrow \bullet$ in general). We have $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$ if, starting from a state satisfying P and with pc equal to w_0, then we can reach a state satisfying Q with pc equal to w_1. We have $\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle$ if we have the same property and furthermore the machine only executes a single instruction. (We then typically have $w_1 = w_0 + 1$, except for the jmp and jnz instructions). Additionally, these three specifications *require the logical invariants to be preserved at every step of the execution.* This requirement is implicitly inherited from Iris, but is a crucial reasoning principle what we do leverage.

In order to establish a specification of the form $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$, one typically uses in a sequence single-instructions rules ($\langle w_0; R\rangle \rightarrow \langle w_1; S\rangle$), one for each instruction of the relevant code block. More generally, these three notions of program specification compose in ways one might expect; for instance, they enjoy the following properties:

$$\textsc{SeqFrag}\quad \frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}} \qquad \textsc{SeqFull}\quad \frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet}$$

$$\textsc{StepFull}\quad \frac{\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet} \qquad \textsc{StepFrag}\quad \frac{\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}}$$

The last piece of the puzzle is the adequacy theorem of the logic, which relates a specification established in the program logic to the operational semantics of the machine. Its statement (see below) is here somewhat informal as we lack the precise definition of the semantics of the machine. It reads as follows: "Assume one establishes a specification of the form $\{w; P\} \rightsquigarrow \bullet$, with invariants $\boxed{I_0}$, ..., $\boxed{I_n}$, and for an initial machine state (*regs₀*, *mem₀*) satisfying P and the invariants. Then, these invariants are preserved at every future step of the execution.".

$$\textsc{Adequacy}\quad \frac{\begin{array}{c}\boxed{I_0}, \ldots, \boxed{I_n} \vdash \{w; P\} \rightsquigarrow \bullet \\ (\mathit{regs}_0, \mathit{mem}_0) \vDash I_0 * \ldots * I_n * \mathsf{pc} \Mapsto w * P \qquad (\mathit{regs}_0, \mathit{mem}_0) \longrightarrow^* (\mathit{regs}, \mathit{mem})\end{array}}{(\mathit{regs}, \mathit{mem}) \vDash I_0 * \ldots * I_n}$$

Let us conclude with a somewhat more technical remark, which should nevertheless be interesting for readers familiar with Iris. Our three notions of specifications are in fact defined on top of the more primitive definition of *weakest pre-condition* (wp), provided by Iris. Then, as usual, wp is directly defined in terms of the operational semantics of the machine. In a high-level language, wp typically takes as parameter an expression of the language. However, in the setting of our capability machine, there are no "expressions": programs are ordinary data (integers, in fact) laid out in memory. Instead, our notion of wp is parameterized by an "execution mode": $\mathsf{SingleStep}$ for executing a single instruction, or $\mathsf{RepeatSingleStep}$, for a complete run of the machine.

The following definitions show that the specifications for a single instruction ($\langle w_0; P\rangle \to \langle w_1; Q\rangle$) and a complete execution ($\{w; P\} \rightsquigarrow \bullet$) correspond respectively to these two execution modes of wp. Furthermore, the specification of a code fragment ($\{w_0; P\} \rightsquigarrow \{w_1; Q\}$) is defined in "continuation passing style", on top of the specification for a complete execution.

$$\begin{array}{lcl}
\langle w_0; P\rangle \to \langle w_1; Q\rangle & \triangleq & \mathsf{pc} \Mapsto w_0 * P \mathrel{-\!\!*} \mathsf{wp}\,\mathsf{SingleStep}\,\{\mathsf{pc} \Mapsto w_1 * Q\} \\
\{w; P\} \rightsquigarrow \bullet & \triangleq & \mathsf{pc} \Mapsto w * P \mathrel{-\!\!*} \mathsf{wp}\,\mathsf{Repeat\ SingleStep}\,\{\mathsf{True}\} \\
\{w_0; P\} \rightsquigarrow \{w_1; Q\} & \triangleq & \{w_0; P * \{w_1; Q\} \rightsquigarrow \bullet\} \rightsquigarrow \bullet
\end{array}$$

4 Logical Relation and its Fundamental Theorem

Our program logic is not only useful to establish the functional correctness of known programs, but also to define the key principle that we use to reason about unknown code.

We give a definition of what makes a value (i.e. a machine word) "safe to share with unknown code". Intuitively, a value is safe to share with an adversary if it satisfies the following "capability safety" contract: the capability must only give access to the memory it has authority over (as given by its range and permission), and cannot be used to increase this authority or invalidate an invariant of the logic.

The formal definition is given in Figure 3. We define both the notion of value that is "safe to share" ($\mathcal{V}$) and "safe to execute" ($\mathcal{E}$) in a mutually recursive fashion.

- *A value safe to share* only gives transitive access to other values that are safe to share, or code that is safe to execute (in the case of a closure).
- *A value safe to execute*, provided the registers contain safe values, allows the machine to run while preserving the Iris invariants (by definition of $\{\cdot;\cdot\} \rightsquigarrow \bullet$).

Technically speaking, this definition is circular. We can nevertheless define it with the help of the $\triangleright$ modality, owing to the fact that Iris is a step-indexed logic. Outside of this technical requirement, the reader can in practice ignore the use of $\triangleright$ here.

A capability RW- gives read and write access to its range: there is no choice than to define the values in this range as being also safe to share. However, a capability with permission RO/RX cannot be used by unknown code to modify the memory words in its range. Therefore, these words can obey any invariant (P) as long as it entails safety ($\mathcal{V}$).

An object capability E is safe to share if the code it encapsulates is safe to execute. Such a capability can be executed at any moment: this fact is expressed through the use of the $\square$ modality. In Iris parlance, this means that the definition of $\mathcal{V}(\mathrm{E}, -)$ is "always true", and must be therefore established by only relying on logical invariants (since these do always hold).

Is this definition of safety trivial? In other words, isn't the definition of safety given in Figure 3 always true? The answer is no! However, convincing oneself of that is not completely obvious.

As a first step, the definition of $\mathcal{E}(w)$ is not trivial because it requires proving that, starting from w, a full execution of the machine *preserves logical invariants.* This requirement is not

$$\boxed{\mathcal{E}(w)} \quad \triangleq \forall reg,\ \left\{w; \mathop{\Large *}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Rightarrow v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z) & \triangleq \mathsf{True} \\ \mathcal{V}(\textsc{e}, b, e, a) & \triangleq \triangleright \Box \mathcal{E}(\textsc{rx}, b, e, a) \\ \mathcal{V}(\textsc{ro}/\textsc{rx}, b, e, -) & \triangleq \mathop{\Large *}_{a\in[b,e)} \exists P,\ \boxed{\exists w,\ a \mapsto w * P(w)} * \triangleright \Box \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w) \\ \mathcal{V}(\textsc{rw}/\textsc{rwx}, b, e, -) & \triangleq \mathop{\Large *}_{a\in[b,e)} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \end{cases}$$

Figure 3: Logical relation defining "safe to execute" and "safe to share" values.

visible in the definition, but is implicit in the definition of the program logic and Iris. Secondly, the definition of $\mathcal{V}(w)$ is also not trivial because, e.g. in the case of a RW capability, it requires the predicate $a \mapsto -$ to be part of a specific invariant ($\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$). Now, note that a predicate "$a \mapsto -$" is not duplicable. Every memory cell whose contents evolve according to an invariant more specific that the one above thus cannot be associated with a safe capability (according to $\mathcal{V}$). There can be only one predicate $a \mapsto -$, which cannot be simultaneously part of two different invariants.

What is a concrete example of a capability which is *not* safe? Let us consider a memory cell at address x initialized to 0. Let us assume the following Iris invariant: $\boxed{x \mapsto 0}$. This invariant expresses that x will contain the integer 0 for the rest of the execution. Then, a capability $(\textsc{rw}, x, x+1, x)$ is not safe to share with an adversary! Indeed, an adversary could use such a capability to write an arbitrary value at address x, thus invalidating the Iris invariant. (However, $(\textsc{ro}, x, x+1, x)$ would be safe.) Formally speaking, it is not possible to prove $\mathcal{V}(\textsc{rw}, x, x+1, x)$, because it is not possible to create the invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$. The resource for the memory cell x is already part of the invariant $\boxed{x \mapsto 0}$, and cannot be extracted to create a different invariant. Similarly, one cannot prove $\mathcal{E}$ of any code fragment that writes another value than 0 at address x, because the proof would not be able to guarantee that the Iris invariant related to x is preserved at every step.

Fundamental theorem. The fundamental theorem (Theorem 1) is the main result of this work: it is a non-trivial theorem, whose proof requires checking all the possible cases of every instruction in the semantics of the machine. It establishes that, according to our definition, any code that is "safe to share" is in fact "safe to execute". As a consequence, this theorem gives us a specification for the behavior of arbitrary code. As long as a capability only gives access to safe memory words (in particular, arbitrary instructions correspond to integers and are thus always safe), then it is safe to execute.

Theorem 1 (FTLR)**.** *Let* $p \in \mathrm{Perm}, b, e, a \in \mathrm{Addr}$*. If* $\mathcal{V}(p, b, e, a)$*, then* $\mathcal{E}(p, b, e, a)$*.*

Another interpretation of the fundamental theorem is that it expresses that the capability machine "works well". Running (possibly arbitrary) instructions cannot create more authority than what was initially available. If that was the case, then this would reveal a design or implementation bug of the capability machine.

The astute reader will have noticed that our definition of safety does not distinguish between permissions -X and those without X. This is a direct consequence of the fundamental theorem! Indeed, our logical relation model expresses the "authority" that a fragment of code has over memory. And following Theorem 1, being able to execute code does not yield additional authority compared to only being able to read it.

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

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

5 Reasoning in presence of unknown code: an example

(The code and proofs presented in this section correspond to the files `adder.v` and `adder_adequacy.v` in the `theories/examples/` folder of the Coq formalisation.)

Let us come back on the example introduced in Section 2. To make our life slightly simpler and avoid relying on an auxiliary memory allocation routine, we assume that our verified component is given (exclusive) access to a memory cell x. We then build a closure that encapsulates access to x. At a high level, the implementation of our component is thus equivalent to:

$$(\lambda n.\ \text{if}\ n \geq 0\ \text{then}\ x := !x + n)$$

We then wish to verify the following property: *for any adversarial component linked against this program, if x initially contains a positive integer, then, at any point of the execution, the value of x remains positive.* Indeed, the adversary does not have direct access to x, but only of the closure written above. Assuming a proper implementation of this closure (relying on an object capability), then the adversary can only modify x by calling the closure, and one can check that the closure can only increase the value of x.

The concrete code that we verify is a sequence of machine code instructions. It appears in Figure 4, in pseudo-assembly syntax. The code that is specific to our component is composed of two routines: `g`, which is executed once at the beginning of the execution and creates the closure, and `f` which implements the closure body itself.

Figure 5 illustrates the memory layout before and after executing `g`. The code at `g` is assumed to receive in register r_2 a capability to x, and then creates the closure encapsulating this capability, eventually passing control to the adversary by jumping to the capability in register r_0, which we know nothing about. In order to create the closure, `g` uses the macro-instruction `crtcls` (which unfolds into a sequence of instructions not detailed here). This macro writes in memory the activation code of the closure, the capability to x, and the capability to the code of `f`. It then encapsulates the whole region into an object capability (with permission E), that it writes in r_1. The activation code of the closure appears in Figure 4. This is the code that gets executed every time the closure is invoked and which passes control to `f` after some initial setup. It is not specific to the concrete component executed here, and simply corresponds to the specific implementation of closures that we consider here.

Whenever the closure is invoked by the adversary, the activation code (`a`) is executed. The activation code copies into registers the capabilities stored in the closure after the activation code (for x and `f`). It then jumps to `f` (the closure's body), which uses the capability to x to (possibly) modify its value, and then takes care of cleaning up temporary capabilities from the registers, before passing control back to the adversary (by convention, this is done by jumping to the return pointer in r_0).

```
;; r1: capability to a memory region where
;;  to write the closure activation code
;; r3: capability to address x
g: move r2 pc
   lea r2 23 ;; offset to f
   subseg r2 f f_end ;; restrict to f's code
   ;; crtcls (destination) (code) (data)
   crtcls r1 r2 r3
   ;; r1 = closure (E-capability), r2, r3 = 0
   jmp r0
g_end:

a: move r1 pc
   lea r1 7
   load renv r1
   lea r1 -1
   load r1 r1
   jmp r1
   data 0 ;; will be: code capability
   data 0 ;; will be: data capability
a_end:
```

```
;; renv: capability to address x
;; r2: value given as argument by the caller
;;      (supposed to be an integer)
f: move r1 pc ;; / r1: address to the end
   lea r1 7   ;; \      of the program
   lt r3 r2 0 ;; do we have r2 ≥ 0?
   jnz r1 r3 ;; if not: exit
   load r3 renv  ;; / if yes: add the integer
   add r3 r3 r2  ;; | to the private state
   store renv r3 ;; \ ...
   move renv 0 ;; / clean up capabilities
   move r1 0   ;; \ pointing to private state
   jmp r0
f_end:
```

Figure 4: Implementation of our verified component.
g: code creating the closure; f: body of the closure; a: activation code of the closure. For simplicity, we assume that the code of g and f follow each other in memory, i.e. g_end = f.

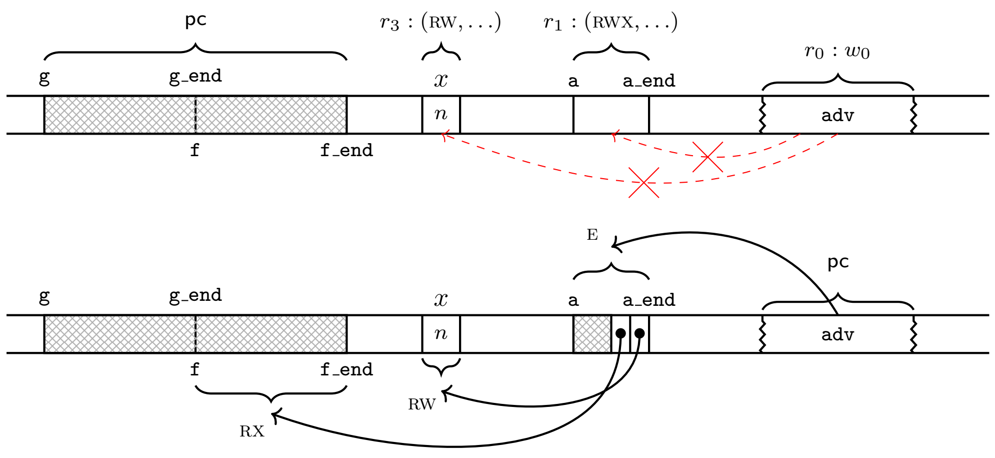

Figure 5: Memory layout: 1) initially, and 2) after executing g.

Firstly, one needs to state and prove a specification for each routine presented. These specifications appear in Figure 6. We do not detail the proofs: they are a "standard" exercise in program verification. Each specification is proved by stepping through the code of f, g or a, and using in a sequence the respective specification of each instruction encountered. Note the use of several invariants. There is one invariant for the code of each program (which does indeed need to be in memory and readable). More importantly, the specification for f is proved under an invariant requiring the value stored at address x to always be a non-negative integer.

Given proofs for these specifications, the most interesting part of the proof remains. We now

$$\boxed{[\mathsf{g}, \mathsf{g\_end}) \mapsto g_{instrs}}$$
$$\vdash \begin{array}{l} \left\{ (\mathrm{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g});\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto - * \\ r_3 \Mapsto (\mathrm{RW}, x, x+1, x) * [\mathsf{a}..\mathsf{a\_end}] \mapsto [-] \end{array} \right\} \rightsquigarrow \\ \left\{ w_0;\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{E}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}..\mathsf{a\_end}] \mapsto (a_{instrs} +\!\!+ [(\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] +\!\!+ [(\mathrm{RW}, x, x+1, x)]) \end{array} \right\} \end{array}$$

$$\boxed{[\mathsf{f}, \mathsf{f\_end}) \mapsto f_{instrs}}\ ,\ \boxed{\exists n,\ x \mapsto n \wedge n \geq 0}$$
$$\vdash \begin{array}{l} \left\{ (\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f});\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto - * r_2 \Mapsto k * r_3 \Mapsto - * \\ r_{\mathrm{env}} \Mapsto (\mathrm{RW}, x, x+1, x) \end{array} \right\} \rightsquigarrow \\ \left\{ w_0;\ \exists k'\ n',\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto 0 * r_2 \Mapsto k' * r_3 \Mapsto n' * \\ r_{\mathrm{env}} \Mapsto 0 \end{array} \right\} \end{array}$$

$$\boxed{[\mathsf{a}, \mathsf{a\_end}) \mapsto (a_{instrs} +\!\!+ c_{code} +\!\!+ c_{data})}$$
$$\vdash \quad \{ (\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ \ r_1 \Mapsto - * r_{\mathrm{env}} \Mapsto - \ \} \ \rightsquigarrow\ \{ c_{code};\ \ r_1 \Mapsto c_{code} * r_{\mathrm{env}} \Mapsto c_{data} \ \}$$

Figure 6: Specifications for routines g, f et a.
We write g_{instrs}, f_{instrs} and a_{instrs} the lists containing their respective instructions encoded as machine words.

need to combine the specifications for the different routines with the specification of unknown code (as given by Theorem 1), in order to obtain a specification for a full execution of the complete system. Then, by applying the adequacy theorem, we can obtain that the memory cell at x indeed contains a non-negative integer at every step, by preservation of the relevant Iris invariant. The main steps of this proof are as follows.

The goal is to show the following specification describing a full execution of the machine. Since this specification is established under the logical invariant $\boxed{\exists n, x \mapsto n \wedge n \geq 0}$, from the adequacy theorem, it directly entails the property about x that we wish to obtain in the end.

$$\boxed{[\mathsf{g}, \mathsf{g\_end}) \mapsto g_{instrs}}\ ,\ \boxed{[\mathsf{f}, \mathsf{f\_end}) \mapsto f_{instrs}}\ ,\ \boxed{\exists n, x \mapsto n \wedge n \geq 0}$$
$$\vdash \left\{ (\mathrm{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g});\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto - * \\ r_3 \Mapsto (\mathrm{RW}, x, x+1, x) * [\mathsf{a}, \mathsf{a\_end}) \mapsto [-] * \\ \mathcal{V}(w_0) * \mathop{\Large\ast}_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0..r_3\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

This specification requires the resources necessary to the execution of g (see g's precondition in Figure 6). It also requires w_0 (the pointer to unknown code) and the values contained in the rest of the registers to be safe machine words. One can for example check that w_0 is a capability to a memory region only containing code and integers (i.e. no other capabilities), and initialize the rest of registers to zero. Indeed, from the definition of $\mathcal{V}$, integers are always safe, and so are capabilities pointing to a region that only contains integers.

By composition with g's specification (Figure 6), it is in fact enough to reason about g's continuation. It is thus enough to prove the following:

$$\boxed{[\mathsf{f}, \mathsf{f\_end}) \mapsto f_{instrs}}, \boxed{\exists n, x \mapsto n \land n \geq 0}$$
$$\vdash \left\{ w_0;\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\textsc{e}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}, \mathsf{a\_end}) \mapsto (a_{instrs} \mathbin{+\!\!+} [(\textsc{rx}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] \mathbin{+\!\!+} [(\textsc{rw}, x, x+1, x)]) * \\ \mathcal{V}(w_0) * \mathop{\Large *}_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0..r_3\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Since w_0 is safe (we assume $\mathcal{V}(w_0)$), following the fundamental theorem (Theorem 1), it is safe to execute (thus we have $\mathcal{E}(w_0)$). By unfolding the definition of $\mathcal{E}$, we obtain the property above, provided we can prove that the value in each register is itself safe. This holds trivially for all registers except r_1, which points to our closure. Since we are sharing the closure with adversary code, it falls onto us to prove that it is safe.

We are thus left with proving $\mathcal{V}(\textsc{e}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$. We first take care of putting the resources corresponding to the instructions of the activation code (between a and $\mathsf{a\_end}$) in a new invariant. Then, without detailing the technical details related to the $\square$ and $\triangleright$ modalities, it is enough to show $\mathcal{E}(\textsc{rx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$.

We now need to make use of the specifications of a and f. One can check that they do not assume anything about the values contained initially in the registers (and indeed we only know that they are safe). Similarly, the values contained in the registers after the execution of f are also safe (they are either integers or haven't been modified). By composing those two specifications, we thus obtain:

$$\boxed{[\mathsf{f}, \mathsf{f\_end}) \mapsto f_{instrs}}, \boxed{\exists n,\ x \mapsto n \land n \geq 0},$$
$$\boxed{[\mathsf{a}, \mathsf{a\_end}) \mapsto (a_{instrs} \mathbin{+\!\!+} (\textsc{rx}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f}) \mathbin{+\!\!+} (\textsc{rw}, x, x+1, x))}$$
$$\vdash \begin{array}{l} \{(\textsc{rx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ r_0 \Mapsto w_0 * \mathcal{V}(w_0) * \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}, r_0} r \Mapsto v * \mathcal{V}(v)\ \} \rightsquigarrow \\ \{w_0;\ \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\ \} \end{array}$$

In order to establish $\mathcal{E}(\textsc{rx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$, it is then enough to reason about the continuation of f (where f returns to the adversary by executing the return pointer w_0). In other words, it is enough to establish: $\left\{w_0; \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$. Moreover, w_0 is assumed to be safe (by definition of $\mathcal{E}$): we have $\mathcal{V}(w_0)$. Theorem 1 then gives us $\mathcal{E}(w_0)$, which is exactly what was left to prove.

End-to-end theorem. By combining the result above with the adequacy theorem, we obtain the following theorem, specifying a full execution of the machine running our scenario, and expressed in terms of the operational semantics of the machine:

Theorem 2 (Complete machine execution)**.** *Starting from an initial state of the machine* (reg, mem) *where:*

- *mem has been initialized with the code of* g *and* f*, and unknown code for the adversary (between addresses* adv *and* adv_end*) (Figure 5);*
- $reg(\mathit{pc}) = (\textsc{rx}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g})$*,* $reg(r_0) = (\textsc{rwx}, \mathsf{adv}, \mathsf{adv\_end}, \mathsf{adv})$*,* $reg(r_1) = (\textsc{rwx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$*,* $reg(r_3) = (\textsc{rw}, x, x+1, x)$*, and* $reg(r) \in \mathbb{Z}$ *otherwise;*
- $mem(x)$ *is a positive integer.*

Then, for any reg', mem'*, if* $(reg, mem) \longrightarrow^* (reg', mem')$ *then* $mem'(x)$ *is a positive integer.*

In other words, for a properly initialized machine, then the invariant about x (that has been established in the logic) is upheld at every step of the execution. At every step, the value stored at address x is a positive integer.

Is this theorem satisfactory? We can anticipate two possible comments with respect to the above theorem, that might suggest that it is not as general as one might wish. We expand on these and answer below.

Isn't the theorem making assumptions on the initial state that are a bit too specific to the scenario at hand? Indeed, Theorem 2 does not detail how the machine memory might have been initialized with the code of `g`, `f` or the adversary code. Similarly, one might wonder where do the capabilities assumed to be present in registers `pc`, r_0, r_1 and r_3 come from. In fact, what is the initial state of a capability machine, immediately after it is powered on?

The exact details depend on the concrete implementation of the capability machine. Nevertheless, in any case, a capability machine must provide at startup an "omnipotent capability" giving authority over the whole memory (here, that would be $(\textsc{rwx}, 0, \mathsf{addr\_max}, 0)$). It is then the role of the boot code of the machine to restrict and split this omnipotent capability, and redistribute it between the different components of the machine. Theorem 2 specifies what happens after the boot code has run and performed the initial setup, so as to abstract itself from the implementation details related to machine initialization. Given a concrete implementation of such a boot code, it would be a standard exercise that to check its correctness using our program logic, and connect it to the theorem established here.

Wouldn't it be more general to first execute the adversarial code rather than the trusted code? For the reasons outlined above, we need to trust *some* of the code that runs at machine startup. If we consider the entire boot code to be unknown (or untrusted), then the machine is insecure, and no guarantee can be upheld, since the untrusted boot code is given access to an omnipotent capability. It is thus necessary for *some* trusted code to be part of the initialization code run at machine startup, before giving control to the adversary. The concrete scenario considered here requires from the boot code (not provided) that it prepares a number of memory regions (for the memory cell at x, the region where to store the activation record), and then uses those to setup functions closures before calling to the adversary. The exact details need not be exactly the same from one scenario to another. The counter example presented next in Section 6 makes different assumptions about the boot code. In particular, the counter initialization code allocates memory dynamically (instead of requiring the boot code to setup fixed memory regions for its use). In turn, it requires that the system provides an additional memory allocation routine (`malloc`): the job of the boot code is then to provision such a routine and make it accessible to the counter implementation.

One can think of various scenarios, in which responsibilities for machine initialization are distributed differently between the code run initially, and code indirectly invoked by the unknown code. But it is in any case necessary to run some amount of trusted (and verified) code at machine startup.

6 Case studies

On top of the example detailed in the previous section, we implement and verify a few extra examples, presented thereafter. To simplify the presentation, we only show a high-level version of their implementation, and refer to the Coq formalization for further details. For each example, we verify that the assertions do not fail. More precisely, each example makes use of an auxiliary routine "`assert`". The routine maintains a private memory cell, initialized to 0, and set to 1 when the routine is called on a condition that evaluates to false. We then verify that the private cell held by `assert` contains 0 at every step of the execution.

Counter with increment, read and reset methods. We verify a component implementing a counter, holding a private memory cell (storing the current value of the counter), and which exposes three closures allowing to respectively increment, read, and reset the counter.

$$\begin{array}{l}\text{let } x = \text{alloc } 0 \text{ in}\\ \quad (\lambda().\ x := !x + 1),\ (\lambda().\ \text{assert } (!x \geq 0);\ !x),\ (\lambda().\ x := 0)\end{array}$$

The three closures are verified independently, and use the same invariant for x than in the example of the previous section. Unlike the previous example, the memory cell for x is here allocated dynamically, by calling to the `alloc` auxiliary routine. The final theorem only requires some memory allocation routine to be initialized in memory, and does not require the boot code to pre-allocate memory regions for the counter. As previously, the three closures are passed to an unknown adversary. We can then prove that the assertion never fails. We also prove that the entry-point to the“alloc” routine is safe, and give it to the adversary (so it can also allocate memory if it needs to).

Sharing a read-only (RO) capability. This next example illustrates how one might use and reason about RO (read-only) capabilities.

$$\begin{array}{l}\text{let } x = \text{alloc } 1 \text{ in}\\ \text{let } y = \text{restrict } x \text{ RO in}\\ \text{unknown\_code}(y);\\ \text{assert } (!x = 1);\\ \text{halt}()\end{array}$$

In this example, “unknown_code” is an unknown function, and “restrict x RO” restricts the capability x (which has permission RWX as it has been returned by alloc) to have permission RO. There, adversary code corresponds to the unknown_code function. According to our logical relation model, we can reason about the execution of unknown_code provided we can prove that y is a safe capability. Since y has permission RO, following the definition of safety (Figure 3), we must show:

$$\exists P,\ \boxed{\exists w,\ y \mapsto w * P(w)} * \triangleright \Box \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w).$$

That is, we can describe the contents of the memory pointed by y by choosing any predicate P *at least as restrictive* as $\mathcal{V}$. Intuitively, the value stored in the memory cell need to be safe, because it can be read by the adversary. However, since the adversary cannot modify it, it is possible to enforce any invariant stronger than safety. In order to show that the later assertion does not fail, we pick $P(w) \triangleq (w = 1)$. This predicate does satisfy the condition $\triangleright \Box \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w)$ (since 1 is always safe), and allows us to later show that x does point to 1 after the call to unknown_code.

7 Related work

This paper presents a simplified version of the methodology previously deployed by the authors to reason (also using Iris) about a secure calling convention [GGVS+21]. By itself, the use of object capabilities is indeed not sufficient to provide safe "function calls" at the machine code level, which are fully faithful to the high-level notion of a function call. Object capability do allow implementing some form of local state encapsulation. However, they do not enforce calls and returns to be well-bracketed. In particular, they do not prevent an adversary from invoking a return pointer several times (something that is not possible in a high-level language without control operators).

Skorstengaard et al. [SDB19] show that it is possible to implement a faithful secure calling convention by using an additional kind of "local" capabilities. Their work follows a similar methodology as the one described here, by defining a logical relation which characterizes some notion of safety. Their proofs have however not been mechanized, and the details of the logical relation can be quite hard to follow on paper.

Later, the work by Georges et al. [GGVS+21] introduces, on one hand, a new type of capabilities ("uninitialized") to improve the runtime efficiency of Skorstengaard et al.'s calling convention, and on the other hand, uses Iris to formulate safety as a logical relation and mechanize the corresponding proofs. The use of Iris allows the logical relation to be expressed in a more concise and high-level way than in Skorsteengard et al's. In comparison with the present work, it is still significantly more complicated, because it is more expressive: the logical relation in Georges et al. allows reasoning about well-bracketedness properties of machine-code "function calls", on top of local-state encapsulation.

A number of high-level programming languages allow for programming patterns similar to object capabilities, that enable preserving some local state while interacting with unknown code. (Typically, this can be achieved by leveraging the encapsulation properties provided by closures of the language.) Devriese et al. [DBP16] give a definition of "capability safety" for a subset of Javascript (including some notion of observable effects) using a logical relation, and show that it allows reasoning about a number of concrete examples. Their relation is more expressive than the one we present here, and closer to the logical relation for a safe calling convention [GGVS+21]. It has however only been defined on paper and has not been mechanized.

More recently, Swasey et al. [SGD17] present a program logic which allows reasoning modularly about "object capabilities patterns" in a high-level language. Their methodology is extremely close to the one presented here: the reasoning principles used at the logical level are essentially the same as ours, but are used there in the setting of a high-level language, while we reason about object capabilities on a low-level machine.

For instance, Swasey et al. define two predicates to describe a reference: a predicate for "high integrity" locations ($\ell \hookrightarrow v$), and one for "low integrity" ($\mathsf{lowloc}\ v$) locations. The first predicate gives exclusive access to the corresponding reference, and therefore is not shareable with an adversary; the second is shareable with an adversary, but can only be used to read and write "low integrity" values. In our setting, "high integrity" directly corresponds to the predicate $a \mapsto w$ for a memory cell, and "low integrity" corresponds to the invariant used in the definition of $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Correspondingly, our definitions satisfy the same reasoning rules as the one established by Swaysey et al.; and in particular, we believe that the various "object capability patterns" they verify could be implemented and verified in a similar way in the setting of a capability machine, using the principles presented in this paper.

Nienhuis et al. [NJB+20] formally verify a number of "architectural" properties of CHERI capability machines. This constitutes a significant mechanization effort: the authors tackle the full generality of a realistic operational semantics of CHERI, which is significantly more complex than the minimal machine we consider here. The approach followed by Nienhuis et al. is different from ours: they state the properties they establish as trace properties, over a trace of "abstract actions" describing the various capabilities transiting through the machine during the execution. This approach makes it possible to state the desired properties in a very explicit and concrete fashion. For instance, the authors state and prove a property of "capability monotonicity": during the execution, the authority of available capabilities cannot increase (in other words, the machine does not allow forging new authority). Intuitively, this seems indeed like a very reasonable property, required for the capability machine to work properly. However, things are more subtle in practice: calls between components (in our case, calling to an E-capability) do allow for some restricted form of non-monotonicity. The property proved by Nienhuis et al. is thus restricted to trace fragments that do not include calls to a different component. Our methodology is less explicit, but more expressive. In our setting, the fundamental theorem can be seen as our formulation that "the machine works well". Its (very extensional) statement is admittedly hard to understand in terms of the operational semantics of the machine, but it makes it possible to eventually derive correctness statements that do apply to a full execution of the machine in terms of the operational semantics, including calls between an arbitrary number of components.

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

References

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994.

[DBP16] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about Object Capabilities Using Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*. IEEE, 2016.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, March 1966.

[GGVS$^+$21] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and Provable Local Capability Revocation using Uninitialized Capabilities. In *POPL*, 2021. (Conditionally accepted).

[JKJ$^+$18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[NJB$^+$20] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, May 2020.

[SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42(1):5:1–5:53, December 2019.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM, 2017.

[WNW$^+$16] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5):38–49, September 2016.

[WNW$^+$19] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.