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

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

[1] Aarhus University, Denmark
[2] KU Leuven, Belgium

Abstract

A capability machine is a type of CPU allowing fine-grained privilege separation using *capabilities*, machine words that represent certain kinds of authority. We present Cerise, a mathematical model and accompanying proof methods that can be used for formal verification of functional correctness of programs running on a capability machine, even when they invoke and are invoked by unknown (and possibly malicious) code. Our work has been entirely mechanized in the Coq proof assistant using the Iris program logic framework.

The methodology we present underlies recent work of the authors on formal reasoning about capability machines [5, 9, 11], but was left somewhat implicit in those publications. This extended abstract is intended as a teaser for a longer paper currently under submission [4], which exposes in further details a pedagogical introduction to the methodology, in a simple setting (no exotic capabilities), and starting from minimal examples.

1 Introduction

A capability machine is a type of CPU that enables fine-grained memory compartmentalization and privilege separation through the use of *capabilities*. This type of hardware architecture has been studied since the 60ies [2, 8], and in particular more recently as part of the CHERI project [13], including a commitment from Arm to develop industrial prototypes of capability machines based on CHERI[1]. Capability machines are promising both as a way of improving the security of existing software systems [6], and as the basis for the design of new systems built with security in mind (e.g. as a target for secure compilation [10, 3, 12, 1]).

Capability machines provide fine-grained and scalable privilege separation mechanisms. They distinguish, at the level of hardware, between machine integers and capabilities. Capabilities can be understood as pointers with associated metadata—such as a range in memory and permission over this range (read-only, read-write, ...). A capability is a native machine word, and can be stored in a CPU register or in memory. Then, a machine word containing a capability can be used to access a given portion of memory, depending on the metadata contained in the capability. On the other hand, a machine word containing a bare integer can only be used for numerical computations and cannot be used as a pointer to access memory.

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

Note, however, that capabilities are low-level, flexible, building blocks, which operate at the level of the machine code and whose metadata "just" trigger some additional runtime checks by

[1] https://morello-project.org

the machine. This means that the *properties* we can actually enforce using capabilities crucially depend on how we *use* capabilities: the variety of properties that can be enforced stems from how one can use and combine capabilities.

In this work we show how we can formally *prove* that security properties are enforced for some known verified code, *even when* that code is linked with unverified untrusted third-party code. Our model of interaction between the known and unknown code is very simple: we assume the code is in the same address space and that control is transferred from one to the other using an ordinary jump instruction (see Figure 1). The security properties we focus on are centered around memory compartmentalization, in particular, local state encapsulation. In other words, we wish to formally reason about the fact that a trusted component can protect the integrity of some private data while interacting with untrusted code.

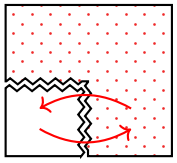

Figure 1: Scenario where a trusted component (plain background) interacts with its untrusted context (dotted background). The component may call or be called by the context, possibly many times.

The key components of our methodology are as follows:

- We define the formal operational semantics of a capability machine, for which we then define a *program logic* based on separation logic, using the Iris framework [7]. This gives us an expressive framework to verify the correctness of *trusted* low-level programs running on the machine, and in particular to establish *logical invariants* they satisfy.
- We define, using our program logic, the specification of what a "safe" capability and a "safe" program is. These are our key tools for reasoning about *untrusted* code. A capability (resp. program) is "safe" if it cannot be used to invalidate an invariant at the logical level. We then show that if a program only has access to "safe" values, then running the program itself is also "safe". This is a global property of the machine, which effectively provides us with a *universal contract* that holds for arbitrary code.
- We illustrate, on concrete scenarios, how manual proofs for trusted code can be combined in the program logic with our universal contract. As a result, one obtains a full-execution theorem about the execution of both trusted and untrusted code, which only depends on the operational semantics of the machine (not on the program logic).

We focus here on a restricted set of capabilities, but the same methodology has been deployed to reason about more advanced security properties, in presence of additional kinds of capabilities or their combination with architectural features such as MMIO [5, 9, 11]. Our work has been fully formalized in Coq, and is available online: `https://github.com/logsem/cerise`.

2 Reasoning about Untrusted Code in Cerise

Let us give a preview of the key concepts that we define to reason about untrusted code. A noteworthy fact is that these can be elegantly defined in a few lines using our program logic and standard Iris constructs. This is quite remarkable, given that these definitions allow reasoning about arbitrary low-level machine-code programs, which for instance make no distinction between code and data or have no notion of structured control flow.

Figure 2 shows the (mutually recursive) definitions of "safe to share" ($\mathcal{V}$) and "safe to execute" ($\mathcal{E}$) machine words. A machine word is either an integer z, or a capability (p, b, e, a) giving access to memory range $[b; e)$ with permission p, and pointing to address a.

$$\mathcal{V}(w) \quad \begin{cases} \mathcal{V}(z) & \triangleq \mathsf{True} \\ \mathcal{V}(\text{E}, b, e, a) & \triangleq \triangleright \, \square \, \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(\text{RW/RWX}, b, e, -) & \triangleq \ast_{a \in [b;e)} \boxed{\exists w,\, a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(\text{RO/RX}, b, e, -) & \triangleq \text{(omitted for simplicity, similar to the RW/RWX case)} \end{cases}$$

$$\mathcal{E}(w) \quad \triangleq \; \forall reg, \left\{ w; \ast_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

Figure 2: Logical relation defining "safe to share" and "safe to execute" values.

The first case of the definition of $\mathcal{V}$ asserts that all integers are safe to share: indeed, they do not give access to memory. Capabilities with permission E correspond to so-called "sentry" capabilities, which provide a form of low-level closures with data abstraction guarantees. A sentry capability is thus safe to share as long as it contains code that is safe to execute (by jumping to the sentry capability). Then, capabilities giving direct access to memory (with permission RW/RWX) are safe to share as long as they give access to words that are themselves safe to share at every point of the execution (where $\boxed{\ldots}$ denotes an Iris invariant).

A machine word w that is safe to execute ($\mathcal{E}(w)$) allows the machine to run while preserving logical invariants, provided the registers contain safe values. ($\{\cdot;\cdot\} \rightsquigarrow \bullet$ is defined by our program logic and specifies a full execution of the machine.)

Then, the Fundamental Theorem of our Logical Relation (Theorem 1) establishes that any capability that is "safe to share" is in fact "safe to execute". In other words, if a capability only gives transitive access to safe capabilities, then it is safe to use it as a program counter capability and execute it. Since the theorem is independent of the actual code that the capability points to, this results is used to specify the execution of *arbitrary code*.

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

Theorem 1 directly entails the following reasoning principles, which are key for reasoning about interactions between trusted and untrusted code:

- A capability pointing to a memory region containing arbitrary instructions (encoded as integers) is always safe to execute;
- It is always safe to jump to an unknown safe machine word (e.g. a "return pointer" provided by the untrusted context) as long as machine registers contain only safe values;
- A sentry (E) capability pointing to trusted code is always safe to share with untrusted code, as long as the encapsulated trusted code only assumes that it receives safe values, and always returns safe values.

3 Case Studies

We apply our methodology to a number of case studies of increasing complexity. These demonstrate how one reasons about security guarantees upheld by concrete programs, but also show that our approach scales up to the (modular) verification of larger programs.

Simpler examples demonstrate how to protect the integrity of a static memory location: by simply not giving a capability to it to the context (effectively proving the impossibility of a buffer overflow attack), or by encapsulating it in a sentry capability, which allows (only) trusted code to update the protected value. We then consider more sophisticated examples, which involve dynamic memory allocation, and build up higher-level constructs such as a heap-based calling convention and so-called object capability patterns.

References

[1] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. CHERI JNI: Sinking the Java Security Model into the C. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 569–583. ACM, 2017.

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

[3] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. CapablePtrs: Securely Compiling Partial Programs using the Pointers-as-Capabilities Principle. May 2020.

[4] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. *Submitted for publication*, 2021. https://cs.au.dk/~birke/papers/cerise.pdf.

[5] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021.

[6] Nicolas Joly, Saif ElSherei, and Saar Amar. Security analysis of CHERI ISA. Technical report, Microsoft Security Response Center, 2020.

[7] 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.

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

[9] 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.

[10] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.*, 3(POPL):19:1–19:28, January 2019.

[11] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. volume CSF, 2022.

[12] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, ICFP, 2019.

[13] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020.