Morello-Cerise: A Proof of Strong Encapsulation for the Arm Morello Capability Hardware Architecture

ANGUS HAMMOND, University of Cambridge, UK
RICARDO ALMEIDA*, University of Glasgow, UK
THOMAS BAUEREISS, University of Cambridge, UK
BRIAN CAMPBELL, University of Edinburgh, UK
IAN STARK, University of Edinburgh, UK
PETER SEWELL, University of Cambridge, UK

When designing new architectural security mechanisms, a key question is whether they actually provide the intended security, but this has historically been very hard to assess. One cannot gain much confidence by testing, as such mechanisms should provide protection in the presence of arbitrary unknown code. Previously, one also could not gain confidence by mechanised proof, as the scale of production instruction-set architecture (ISA) designs, many tens or hundreds of thousands of lines of specification, made that prohibitive.

We focus in this paper especially on the secure encapsulation of software components, as supported by CHERI architectures in general and by the Arm Morello prototype architecture and hardware design in particular. Secure encapsulation is an essential security mechanism, for fault isolation and to constrain untrusted third-party code. It has previously often been implemented using virtual memory, but that does not scale to large numbers of compartments. Morello provides capability-based mechanisms that do scale, within a single address space.

We prove a strong secure encapsulation property for an example of encapsulated code running on Morello, that holds in the presence of arbitrary untrusted code, above a full-scale sequential model of the Morello ISA. To do so, we build on, extend, and unify three orthogonal lines of previous work: the Cerise proof of such an encapsulation property for a highly idealised capability machine, expressed using a logical relation in Iris; the Islaris approach for reasoning about known code in production-scale ISAs; and the T-CHERI security properties of arbitrary Morello code, previously proved only for executions up to domain crossing.

This demonstrates how one can prove such strong properties of security mechanisms for full-scale industry architectures.

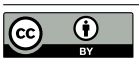
CCS Concepts: • **Security and privacy → Logic and verification**; • **Theory of computation → Program verification**; **Logic and verification**; • **Computer systems organization → Architectures**.

Additional Key Words and Phrases: Capability machines, CHERI, instruction set architectures, program logic, separation logic, mechanised proof

ACM Reference Format:
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, Article 226 (June 2025), 23 pages. https://doi.org/10.1145/3729329

*Work done while affiliated with the University of Edinburgh.

Authors' Contact Information: Angus Hammond, University of Cambridge, Cambridge, UK, angus.hammond@cl.cam.ac.uk; Ricardo Almeida, University of Glasgow, Glasgow, UK, Ricardo.Almeida@glasgow.ac.uk; Thomas Bauereiss, University of Cambridge, Cambridge, UK, Thomas.Bauereiss@cl.cam.ac.uk; Brian Campbell, University of Edinburgh, Edinburgh, UK, Brian.Campbell@ed.ac.uk; Ian Stark, University of Edinburgh, Edinburgh, UK, Ian.Stark@ed.ac.uk; Peter Sewell, University of Cambridge, Cambridge, UK, Peter.Sewell@cl.cam.ac.uk.



ACM 2475-1421/2025/6-ART226
https://doi.org/10.1145/3729329

1 Introduction

Encapsulation of software components is widely used as a security mechanism in systems, both as a fault isolation technique and as a boundary to contain the behaviour of untrusted third-party code. There are examples at many scales: from virtual machines sharing hardware in cloud computing, through operating system processes, to the in-process sandboxing used in web browsers to contain Javascript, or to protect against errors in image decoding libraries.

The CHERI architecture [18, 19] provides new hardware features to build more secure systems, which can be retrofitted to existing instruction set architectures: capabilities which are unforgeable, with fine-grained permissions to access a resource within specified limits. Arm's experimental Morello architecture has extended the Armv8.2-A architecture with CHERI features, and has been incorporated into the Morello prototype hardware implementation, based on the high-performance Neoverse-N1 processor [10]; Microsoft have developed CHERIoT, a microcontroller-scale architecture and hardware implementation of CHERI [1], and RISC-V International are currently standardising CHERI RISC-V.

One main use of CHERI capabilities is to provide spatial memory safety, e.g. by recompiling existing C or C++ as CHERI C or CHERI C++, to implement language-level pointers with capabilities, with accesses limited by compressed object bounds stored in the capability's metadata. This typically requires very minor changes to the source code. CHERI capabilities can also be used for many other purposes, including per-pointer read/write/execute permissions, temporal safety using GC-like memory sweeping for revocation [5], control-flow integrity, and, the focus of this paper, providing *fine-grained compartmentalisation between mutually untrusting components*. This can be within a single address space, obviating the typical cost of compartmentalisation implemented by managing address translation with a memory management unit (MMU), which does not scale well to large numbers of compartments and incurs a high performance cost when context-switching. For example, Narayan et al. [15] study fine-grain isolation for Firefox and partly motivate their software-based fault isolation scheme by noting that cross-process function calls are up to 300 times slower than an ordinary function call. CHERI's hardware-supported capability mechanism promises much better performance, although evaluating the performance of realistic CHERI compartmentalisation implementations in detail is ongoing work. One current prototype of CHERI compartmentalisation [6] places each shared library in a compartment, and each call to a library function becomes a cross-domain call. This way, the memory that a library can access can be restricted by the capabilities given to it by callers and at initialisation (by the dynamic linker).

A key question for any new architectural security feature is how one can know that the underlying ideas and the detailed instruction-set architecture (ISA) design actually guarantee the intended properties. This is particularly important because, if the ISA specification is flawed, then every conforming hardware implementation will be exploitable, and because late discovery of such flaws may require impractically large changes or workarounds in hardware and software.

In the small-scale setting of an academic cut-down and simplified architecture, previous work demonstrated that one can establish strong compartmentalisation properties by mechanised proof. The Cerise logic of Georges et al. [7] takes this classic approach, extracting some key ideas of capability architectures into an idealised setting where proof techniques can be developed. They developed a logical relation characterising which capabilities a compartment running known code can safely share with a compartment running unknown, untrusted code, while maintaining an internal invariant – a deep property that in some sense captures the essence of that compartmentalisation scheme. This is mechanised in the Iris separation logic framework [13]. Such a setting is also an excellent environment to study theoretical properties of proposed architecture extensions [8].

However, whether corresponding properties can be shown to hold of a full-scale ISA definition, such as that of Morello, is a very different question. The problem is the scale and intricacy of such ISAs. The Cerise ISA semantics fits on one page [7, Fig. 6], but the Morello ISA definition is around 62 000 lines of specification, which can be translated into 210 000 lines of Isabelle [3], and it includes a host of subtle aspects, including a sophisticated capability encoding scheme, multiple mechanisms for compartmentalisation, and, for example, additional metadata in virtual addresses, not used when calculating the physical address to access, that includes a bit selecting which instruction set variant a code pointer uses. Errors anywhere within this could obviate the intended security guarantees, yet mechanised reasoning about it is an intimidating prospect. Bauereiss et al. [3] successfully demonstrated that some mechanised reasoning is feasible on this scale. They proved in Isabelle key security properties of all instructions in the Morello ISA (which we refer to as the T-CHERI properties and build upon in this paper, §7.1), and on top of that a *monotonicity* property of the entire (sequential) ISA definition, that arbitrary code cannot extend its available capabilities, up to any domain transition where it calls another compartment or the operating system. This is a key sanity property of the entire production-scale ISA, but it is also in a sense a shallow property: it says nothing about the behaviour after such a domain transition, and hence nothing about any compartmentalisation scheme as a whole.

For known code, Islaris by Sammler et al. [17] provides a practical approach for reasoning about specific instructions with respect to production-scale ISAs, using Isla [2] to symbolically evaluate the full-scale instruction semantics, under SMT assumptions on the opcode and register values, and embedding the resulting symbolic traces in Iris. For known code, this symbolic evaluation can substantially simplify the semantics, e.g. by removing cases for other exception levels or unaligned accesses – but it was done only for Arm-A and RISC-V, not for capability architectures such as Morello, with their additional complications.

Contributions. The main contribution of the current paper is to show how one can combine and scale up ideas from the three above directions, to show a deep compartmentalisation property of the entire Morello (sequential) ISA definition, and thereby to demonstrate that such proofs are viable for industrial ISA designs of other security extensions in the future.

This is a two-sided problem. First, does the ISA provide us with strong enough properties with respect to arbitrary code, even when written by a malicious party? Second, can we reason about software that uses capabilities to protect itself against unknown, potentially adversarial code it calls or that calls it? The T-CHERI and monotonicity properties of Bauereiss et al. [3] are a step towards the first, showing that arbitrary code cannot extend its reach until it calls another compartment or the operating system. It is exactly at this point that the second problem takes over, because we need to ensure that our code does not provide the other compartment with too much power. We demonstrate that Cerise-style reasoning can be applied to the full Morello architecture, building a logical relation that captures the intended deep compartmentalisation property. For reasoning about the unknown arbitrary code within compartments, we adapt Bauereiss et al. [3] to supply the required properties of unknown code. For reasoning about known code, a common approach is to apply symbolic execution to the ISA specification, taking advantage of information from the context such as the instructions and system configuration to cut down the behaviours examined [14, 16, 17]. We adapt Islaris [17], which is built on the Sail toolchain that our formal model for the Morello architecture uses, and which uses the Iris separation logic framework like Cerise.

As a simple motivating example, we have written a small assembly function that increments a cyclic counter, which we describe in detail in §4. When properly initialised, Morello's capabilities can limit access to the counter's state to the function itself, ensuring that the counter always stays in a limited range. This is analogous to the secure-counter example in Cerise [7, §2.4]. While the

example is modest, it illustrates the key point that we can prove that the invariant on the state is maintained, regardless of the unknown code's actions.

In more detail, our contributions are:

- A characterisation of sufficient conditions for a piece of encapsulated code to protect itself from adversarial unknown code on the Morello architecture, in the style of Cerise [7] using a unary logical relation (§6).
- A mechanised proof of the fundamental theorem of that logical relation (FTLR, §7.2), showing that its conditions do suffice to ensure that encapsulated code runs without interference.
- Enhancing the mechanised T-CHERI safety properties [3] to support that proof (§7.1).
- Adapting the Islaris machine code program logic [17] to support CHERI capabilities (§5).
- A small but complete example demonstrating that the above can be used to prove correctness of a program when that correctness depends on CHERI protections (§4, §8).

Collectively these establish that the protections provided by Morello can be used to safely isolate mutually distrusting code compartments running in a shared address space, and that we can in practice prove correctness of a program which depends on that isolation, even in settings with repeated calls between such compartments.

Caveats and limitations. In order to achieve this we do still consider a slightly simplified system. In particular we consider processor exceptions to terminate execution, and consider these terminations to be successful executions, under the assumption that the operating system is trusted. As a result we can assume that the values of several system registers are fixed, since they can only be modified after an exception raises the processor exception level, and in particular we pick a processor configuration in which address translation is disabled. We consider only sequential execution.

To enable reuse of the existing properties proven by Bauereiss et al. [3] we rely on a manual translation of several of their top level definitions between Isabelle and Rocq (see §7.1 for a side-by-side comparison for an example property).

2 Encapsulation Mechanism

The goal of encapsulation is to protect a piece of code and its data from unintended interference by other potentially untrusted code. Throughout this paper, we refer to the protected code and data as being encapsulated from untrusted code, although one could also see it the other way around, with the *untrusted* code being encapsulated in order to mitigate its unintended effects. As the protected code is normally intended to interact with other code in some way, a formal treatment of encapsulation might have to take into account application-specific data invariants and interaction patterns with other code. In §8 we give an example of such a formal security property for the code introduced in §4. We call it a *strong* encapsulation property due to the fact that it guarantees a data invariant of the encapsulated code even in the presence of arbitrary untrusted code. Before going into formal details, we first give some background on the capability mechanisms that CHERI provides and how they can be used for encapsulation.

Simple application of CHERI capabilities allows one to protect regions of memory, just by ensuring that all the available capabilities only cover regions of memory the currently executing code is allowed to access. A CHERI capability includes not just a virtual address, but also the base address, bounds, and permissions for the region of memory it is allowed to access, and other metadata, compressed into 128 bits (for 64-bit architectures). Capabilities are also each associated with a one-bit tag (in each capability register, and for each capability-sized and aligned unit of memory) that the hardware uses to track capabilities that have been constructed by legal operations, preventing the forging of capabilities by (for example) writing their byte representations. In CHERI C and CHERI C++, all language-level pointers are compiled to capabilities, as are implementation

pointers such as return addresses. Because capabilities include their base and bounds, the hardware can do a fast check, on each access, that the access is permitted — unlike conventional architectures and conventional C/C++.

However, because we want to allow interaction between encapsulated code and potentially adversarial arbitrary code, we also need to be able to transition from the encapsulated memory region being protected to it being accessible when the encapsulated routine is executed. CHERI *sealed capabilities* allow this behaviour. Any capability can be sealed, and a sealed capability cannot be used to authorise memory accesses until it is unsealed again.

Morello supports a number of different forms of sealed capability, identified by a type stored in the capability, each of which has a different unsealing mechanism. Any attempt to use the capability without unsealing will fault, and modification will invalidate it. To protect our encapsulated code and its memory we use *indirect sentry* (sealed-entry) capabilities, which are sealed capabilities that point to a region of memory, the first word of which is a code capability. When an indirect sentry capability is invoked it is unsealed and loaded into the register `C29`, and the code capability pointed to is loaded into the program counter. This ensures that the sealed capability can only be used to access memory by the intended piece of code.

This allows us to provide unknown code with a way to branch into our encapsulated code by providing it with an indirect sentry that covers the encapsulated memory region and points to the address of the code which is allowed to access it. We ensure that whenever we branch into unknown code all other accessible capabilities only permit access to memory we intend to allow the unknown code to modify.

An important detail of Morello sealed capabilities (differing from some other CHERI architectures) is the special handling of the `C64` register, which controls which of two instruction decoding modes the processor is in. Because this register can be manipulated by unprivileged code and can completely change the meaning of some instructions, it would be unsafe in general to allow an adversary to choose its value when our encapsulated code starts executing. To prevent this, when Morello sealed capabilities are invoked, the `C64` register is set to the least significant bit of the value of the invoked capability and the value written to the program counter has this bit cleared.

3 Proof Outline

Our proofs that programs are correctly encapsulated consist of two principal components. First we prove that the encapsulated code itself (which is all concretely known during the proof) runs safely, correctly preserving whatever invariants it relies upon. Secondly we prove that whatever other code may be run on the machine (which may be completely arbitrary, or attacker controlled) is unable to violate those invariants.

In both cases the principal challenge is the scale of the specification for Morello. While similar proofs have been carried out for simple models of capability machines, the approach of reasoning directly over the full machine semantics in Rocq does not currently scale to a realistic architecture. Instead we make use of two techniques that allow us to reason over a significantly simpler semantics, while retaining confidence that our results apply to the full machine.

For reasoning about known code we use Islaris, which allows for symbolic evaluation and SMT driven automatic simplification of the machine semantics, and provides an Iris-based separation logic for specifiying and verifying the behaviour of that known code over the resulting simplified semantics. In §5 we describe changes we made to the existing Islaris tooling to support reasoning about code using capabilities. In §8 we describe how to use Islaris to prove a property of the example program we present in §4.

To reason about arbitrary code we want to prove a general specification about the behaviour of all Morello instructions in our Islaris based separation logic. Intuitively this specification takes

ownership over all the memory that is reachable from the set of capabilities initially present in registers and tells us that the machine executes "safely", not violating any invariants stated in the separation logic. We encode this as a unary logical relation (§6), which describes a notion of safe values (those for which we have sufficient ownership to access all the memory they make accessible), and the fundamental theorem of that logical relation (FTLR, §7.2), which says that when both the program counter and all the general purpose registers have safe values, the machine executes safely.

The obvious approach to prove such a specification would be case analysis over all possible Morello instructions, and indeed the Cerise project takes this approach for its simplified capability machine. Unfortunately the scale of the Morello specification, which is tens of thousands of lines long and contains thousands of instructions, renders this direct approach of manual reasoning about it in its entirety impractical. The simplification that Islaris carries out is only possible when the instructions to be executed are known at proof time, so it does not help with reasoning about arbitrary unknown code. Instead we make use of, and extend, an existing characterisation of the behaviour of arbitrary code on CHERI systems in general and Morello in particular due to [3], which we refer to here as the T-CHERI properties.

The T-CHERI properties, which Bauereiss et al. [3] proved in Isabelle to hold for Morello, describe a collection of constraints on the set of traces of arbitrary instructions on CHERI systems, for example, that new capabilities generated by instructions must be legally derived from capabilities the instruction has already read (see §7.1 for more details). It has previously been shown that these properties are strong enough to prove that, up to any domain transition, the set of accessible capabilities cannot be increased, so attacker-controlled code cannot access memory not initially available to it. However it has not previously been shown that they can be used to prove the safety of concrete programs that depend on that protection. We augment the Islaris semantics to have two separate execution states: for parts of the known code we retain the existing Islaris semantics in which instruction memory stores pre-simplified traces describing the possible behaviours of the instruction being executed, whilst for potentially unknown code, instruction memory simply stores a marker that the behaviour of the instruction is unknown. When the machine tries to execute an instruction with unknown behaviour it picks an arbitrary trace of memory and register accesses which satisfies the T-CHERI properties.

The existing T-CHERI properties turn out to be insufficient to prove our desired fundamental theorem. However augmenting them with two additional properties, which better characterise the behaviour around domain transitions, suffices. We will discuss this in §7.1.

Figure 1 shows an overview of the components of the proof, with the existing Morello ISA specification (in ASL, Sail, and Isabelle) forming the basis for both strands of the proof. The top-level theorem Corollary 8.2 highlighted in the figure is an encapsulation property for the example incrementer program we present in the following section. The property states that, when starting with a suitably initialised machine state, the data invariant of the program is preserved even in the presence of arbitrary other code. That theorem is specific to the example program, but the fundamental theorem of the logical relation (FTLR) we prove is generic: it could be combined with an Islaris proof of any program that satisfies our calling convention, to obtain an encapsulation property for that program corresponding to Corollary 8.2.

4 Motivating Example: A Capability-Encapsulated Incrementer

We now present sample Morello assembly code that demonstrates the encapsulation mechanism in action, protecting the trusted code and its local data using sealed capabilities allowing us to ensure that an invariant is maintained on the local data. Our code is based on a Cerise example [7, §2.4] of a small encapsulated routine that manages a counter. The Cerise example then uses *sentry*

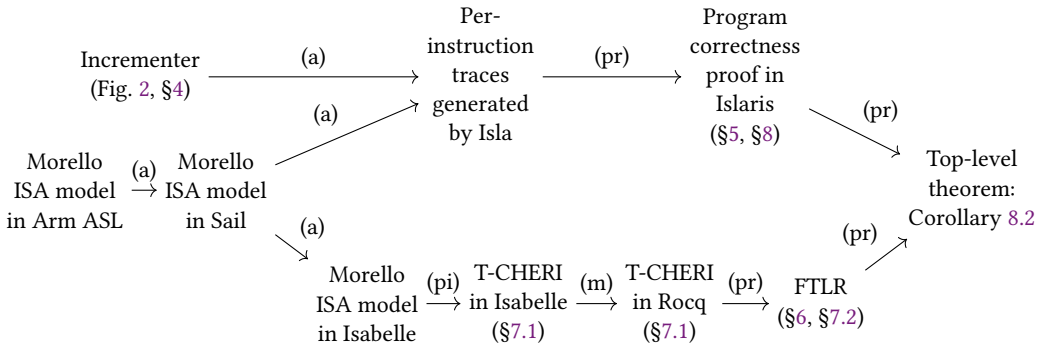


Fig. 1. Overview of the proof, with edges labelled to indicate the relationships between artifacts: (a) for automatic generation of an artifact from another, (m) for manual translation with equivalence checking by manual inspection, and (pi)/(pr) for mechanised proofs in Isabelle/Rocq, respectively.

```
1 incrementer:          // On entry, c29 is the data capability unsealed
2     ldr  w0,[c29,16]  // Fetch counter and store it in 32-bit register 0
3     add  w0,w0,1      // Increment the counter locally
4     and  w0,w0,0x3F   // Cycle back to zero if equal to 64
5     str  w0,[c29,16]  // Write back the counter
6     clrtag c29,c29    // Invalidate the data capability
7     ret               // Return to untrusted caller
```

Fig. 2. The encapsulated incrementer routine maintains a counter which is increased by 1 each time it is called, wrapping back to 0 when it reaches 64. The routine returns the current value of the counter in register 0. The adversary may invoke this routine repeatedly by keeping a copy of the sealed data capability. Any attempt to access the counter except through calling the incrementer routine with `blr [c29,0]` will fault.

(sealed-entry) capabilities provided by the Cerise abstract machine to protect the encapsulated routine from untrusted code and ensure that only the encapsulated code itself can modify the value of the counter in memory.

Figure 2 shows our Morello implementation of this incrementer example. A minor difference from the Cerise example is that our code increments from 0 to 63 and then wraps back to 0, because real architectures do not have unbounded integers as words.

Another difference is that the original Cerise example stores a capability to the data alongside the code so that it can be accessed with a PC-relative load, whereas we separate the protected code and its data, allowing us to put the former into a read-only region of memory. This is compatible with the 'write xor execute' attack mitigation feature found in most operating systems, which forbids directly writing to code memory. Strictly speaking, we do not need this mitigation because our encapsulation using a sealed capability prevents external alteration of the code, but the CHERI architecture aims for high compatibility with existing operating system features. Hence, we maintain this compatibility and use the more sophisticated *indirect* sentry type of capability introduced in Section 2 above, setting it up to point to a writable region of memory containing the data of the incrementer as well as a *pointer* (i.e., capability) to its code in a different memory region.

Our formal encapsulation result presented in §8 assumes that untrusted code starts executing in a machine state where this sealed capability has been set up correctly and is the only capability to

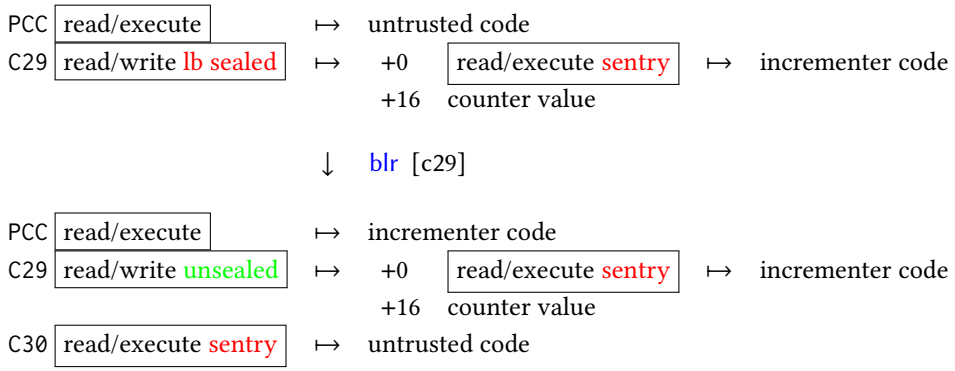


Fig. 3. Informal effect of invoking the sealed capability

the encapsulated code and data available to the untrusted code. More concretely, we assume that this capability is sealed with the lb (load-and-branch) sealing type, making it an indirect sentry capability, and that it points to a 20-byte region of memory that contains a 16-byte capability pointing to the incrementer code followed by the 4-byte counter value. For running our code, we have implemented a test harness (omitted here for brevity, but available in the artifact accompanying the paper [11]) that initialises the machine state by constructing the sealed capability in register C29 and then clearing the other capability registers before jumping to untrusted code.

The untrusted code therefore has no direct access to the encapsulated code or its data, but it can call the incrementer by placing the sealed capability into register C29 and invoking it using the memory-indirect branch-and-link instruction blr [C29], which unseals the capability, fetches the code capability stored at the start, branches to it, and stores a return capability to the caller in register C30. This is illustrated in Figure 3. The incrementer code in Figure 2 then updates the counter before invalidating C29 to avoid leaking the unsealed data capability (line 6) and then returning to the caller.

The code capabilities in Figure 3 also happen to be sealed as plain *sentry* capabilities. This is not essential to the encapsulation of the function, but rather normal practice on CHERI architectures to provide a form of control flow integrity.

Our code makes no attempt to conceal the location of the trusted routine or its private data (sealing capabilities or wiping their validity tags still leaves their bitwise content visible to untrusted code). We are concerned only with whether untrusted code is able to access protected memory locations.

The running example we chose for this paper is intentionally small and simple to focus our presentation. In particular, the code does not have loops or other more complex control flow, but we note that reasoning about control flow within the known code is orthogonal to the challenge of reasoning about encapsulation and can be handled by the existing Islaris approach. Notice that the traces Islaris reasons over describe the behaviour of individual instructions, and after executing an instruction trace, Islaris uses the program counter and an "instruction memory" (mapping addresses to traces) to pick a new trace to execute. This allows reasoning about programs which contain unbounded loops or other complex control flow, as such control flow is realised by regular instructions modifying the program counter. This trace based approach is viable because the specifications of single instructions do not contain unbounded loops. See Sammler et al. [17]

for a demonstration of how Islaris can be used to verify a memcpy implementation, a hypervisor call and return, and other examples.

5 Machine Model

We make several adaptations to the Islaris machine model to support reasoning about both capabilities and partially unknown code. First we extend Islaris's state to include a capability tag for every 16 byte aligned memory address. Instruction traces can explicitly read and write from these tags, and they are implicitly cleared by any writes that overlap with a capability in memory.

Second, we augment Islaris with a notion of processor exceptions, which we consider to immediately terminate execution. These terminations are considered to be a safe behaviour because on hardware they would result in control passing to the operating system, which we assume to be trusted. Capability errors (such as attempts to access memory using a capability that is untagged or which does not cover the accessed address) cause such exceptions, and we want to consider this behaviour to be safe because the exception is the mechanism by which hardware prevents the unsafe memory access.

Third, we modify Islaris's instruction fetch to fail safely if the program counter contains a capability which is not tagged, is out of bounds or which does not have the execute permission. Any of these errors would cause an exception on Morello hardware, but they are specially handled here because Islaris treats instruction fetch separately to instruction execution.

Finally we augment Islaris's instruction memory, which previously contained a map from addresses that might be executed to instruction traces. We instead allow addresses to be mapped either to "known" code, which is still represented using instruction traces, or to "unknown" code, which is simply marked as unknown. As in existing Islaris, when an instruction is fetched from an address containing known code a trace is loaded from instruction memory, then executed event by event using the Islaris operational semantics. When an instruction is fetched from an address containing unknown code, an arbitrary trace satisfying the T-CHERI properties is picked nondeterministically, and is similarly executed using the Islaris operational semantics.

In addition to these modifications we assume a number of system registers that can only be modified at raised exception levels take fixed values. This allows us to simplify the traces of known instructions under the assumption that these registers have their assigned values. This is justified by the fact that we consider executions starting in exception level 0 (user mode) where these registers are inaccessible, and consider all executions that change exception level (by taking an exception) to have safely terminated.

6 Logical Relation

We capture the notion of a capability being safe using two separation logic predicates. The full Rocq definitions are available in the artifact accompanying this paper [11]. Here we simplify somewhat for exposition, showing conventional paper-maths versions which elide many minor details including the treatment of register values that are not capability sized, address manipulations to handle Morello's support for discarding high order address bits and the more detailed handling of direct sentry capabilities. $\mathcal{V}$, shown in Figs. 5 and 6, captures the idea of a capability being safe to share with adversarial code, and $\mathcal{E}$, shown in Fig. 4, captures the idea of a capability being safe to execute, i.e. that the machine will execute safely if the capability is placed in the program counter.

$\mathcal{E}(pcc)$ takes ownership of all general purpose registers, fixing the value of `PCC` (the program counter capability) to *pcc*, and requiring that all other registers that may contain capabilities contain values satisfying $\mathcal{V}$. Because some kinds of sealed capability (including indirect sentry capabilities) set the values in registers `C29` (sometimes also called `IDC` or *invoked data capability*) and `C64` when they are invoked, we also define $\mathcal{E}_{gen}$ which further specifies the exact values of those registers. Both

$$\begin{aligned}\mathcal{E}_{gen}(pcc, idc, c_{64}) = {} & \text{“PCC”} \mapsto_R pcc \twoheadrightarrow \\ & \text{“C29”} \mapsto_R idc \twoheadrightarrow \\ & \text{“C64”} \mapsto_R c_{64} \twoheadrightarrow \\ & \textstyle\bigstar_{r \in \text{gprs}} (\exists v, \mathcal{V}(v) * r \mapsto_R v) \twoheadrightarrow \\ & \text{WP Fetch}\end{aligned}$$

$$\mathcal{E}(pcc) = \forall idc\ c_{64}, \mathcal{V}(idc) \twoheadrightarrow \mathcal{E}_{gen}(pcc, idc, c_{64})$$

Fig. 4. The definition of $\mathcal{E}$, which asserts that a capability *pcc* is safe to execute. We also define $\mathcal{E}_{gen}$ which is additionally parameterised on the values of the registers C29 and C64 because we want to be able to describe capabilities that are only safe to execute with particular values of these registers. gprs is the set of general purpose registers other than C29, C64 and PCC. Fetch is the state of the machine as it has finished executing the previous instruction, immediately before it fetches the next instruction to execute. *pcc* and *idc* range over capabilities and c_{64} ranges over one bit bitvectors.

variants of $\mathcal{E}$ then require WP Fetch, which asserts that the machine executes safely indefinitely, starting from a state in which the previous instruction has finished executing and a new instruction is about to be fetched. Note that WP is a separation logic assertion which holds only if the machine executes without violating any existing Iris invariants or memory ownership that it might be framed with.

$\mathcal{V}(c)$ is defined differently depending on whether and how c is sealed. If c is an unsealed capability, $\mathcal{V}(c)$ is defined to be $\mathcal{V}_{\text{unsealed}}(c)$. The most important aspect of $\mathcal{V}_{\text{unsealed}}$ is $\mathcal{V}_{\text{data}}$ which takes ownership of the memory region accessible using c, plus any extra bytes before and after this region required to give 16-byte alignment. We expand the owned region in this way because Morello capabilities are stored in memory at 16-byte-aligned addresses, and writes to any subset of the bytes of a capability stored in memory will result in that capability being invalidated. To account for multiple capabilities covering the same region of memory we store all this memory ownership in Iris invariants. The invariant is also allowed to impose an additional persistent constraint P on the stored value v (the precise definition of Persistent is provided by Iris, but for our purposes it suffices to note that persistent propositions are duplicable, i.e. Persistent(P) implies $P \twoheadrightarrow P * P$). To ensure memory accesses performed using c are safe, if c permits reading memory P must imply $\mathcal{V}$ and if c permits writing to memory $\mathcal{V}$ must imply P. Intuitively these constraints capture the notion that unknown code must only be allowed to read safe values, and should only be able to write safe values. All recursive uses of $\mathcal{V}$ are guarded by a later modality, $\triangleright$ which ensures the fixpoint $\mathcal{V}$ is defined.

In addition we enforce $\mathcal{V}_{\text{unsealing}}(c)$, which requires that c does not permit directly unsealing other sealed capabilities, and $\mathcal{V}_{\text{executable}}$, which requires that if c has the executable permission then all addresses in bounds of c correspond to unknown code.

We forbid capabilities that allow unsealing in order to straightforwardly prevent the protection mechanisms of sealed capabilities being sidestepped. This prevents us from working with unknown code which expects to make use of unsealing capabilities, but we note that unsealing capabilities are only one of several mechanisms Morello supports for working with sealed capabilities, the rest of which we support. We believe it would be straightforward to relax this restriction to only forbid

$$\mathcal{V}_{\text{data}}(c) = \mathop{\Large *}_{a \in \text{addrs } c} \exists P,$$
$$\boxed{\exists v, a \mapsto_M v * P\ v} *$$
$$\text{read\_cond}(c, P) * \text{write\_cond}(c, P) * \text{persistent\_cond}(P)$$

$$\text{read\_cond}(c, P) = \text{permits\_read}(c) \longrightarrow \triangleright \Box \forall v, P(v) \mathrel{-\!\!*} \mathcal{V}(v)$$
$$\text{write\_cond}(c, P) = \text{permits\_write}(c) \longrightarrow \triangleright \Box \forall v, \mathcal{V}(v) \mathrel{-\!\!*} P(v)$$
$$\text{persistent\_cond}(P) = \forall v, \text{Persistent}(P(v))$$
$$\mathcal{V}_{\text{unsealing}}(c) = \neg\text{permits\_unsealing(c)}$$
$$\mathcal{V}_{\text{executable}}(c) = \text{permits\_execute}(c) \mathrel{-\!\!*} \mathop{\Large *}_{a \in \text{instr\_addrs } c} (\text{instr } a \text{ Unknown})$$
$$\mathcal{V}_{\text{unsealed}}(c) = \mathcal{V}_{\text{data}}(c) * \mathcal{V}_{\text{unsealing}}(c) * \mathcal{V}_{\text{executable}}(c)$$

Fig. 5. The definition of $\mathcal{V}_{\text{unsealed}}$, which describes whether an unsealed capability is safe to share with an adversary.
addrs computes the set of 16 byte aligned addresses such that any byte of a capability stored at that address is accessible using c. instr_addrs computes the set of addresses with the appropriate alignment for instructions within the bounds of c.
instr a i is an Islaris predicate asserting that address a contains instruction i, so instr a Unknown asserts that branching to a will execute an unknown instruction, i.e one which has an arbitrary T-CHERI compliant trace.

unsealing capabilities that have permission to unseal indirect sentries, but did not attempt to prove this to avoid complicating the proof of the fundamental theorem of the logical relation.

We require that only unknown code is executable through capabilities satisfying $\mathcal{V}$ because we know by definition that executing unknown instructions in our modified Islaris semantics gives a trace that satisfies the T-CHERI properties. It would suffice to require that each executable instruction is either unknown or corresponds to a trace that satisfies the T-CHERI properties, but allowing this would further complicate the definition of $\mathcal{V}$ for minimal gain. Note that because the known encapsulated code is accessed through a sealed indirect sentry capability this constraint does not apply to it.

If c is a sealed indirect sentry capability then $\mathcal{V}(c)$ is defined to be $\mathcal{V}_{\text{indirect}}(c)$. $\mathcal{V}_{\text{indirect}}(c)$ also takes ownership of all capability sized regions of memory partially within the bounds of c, again using invariants to allow multiple capabilities to reference the same region. Because the sealed capability cannot be used to read or write to memory in that region we do not need to insist that values stored there satisfy $\mathcal{V}$, but we do need to know that it is safe to execute from all machine states possible after the indirect sentry is invoked. We enforce this by requiring $\mathcal{E}_{\text{gen}}(pcc, idc, c_{64})$ holds, with idc set to the result of unsealing the original indirect sentry capability, pcc set to the value pointed to (with its least significant bit cleared) and c_{64} set to the least significant bit of that value. It is consequently possible for invoking the sealed capability to give access to capabilities that do not satisfy $\mathcal{V}$, and so would not be safe to share with an adversary, so long as it can be proven that invoking the sealed capability results in executing code that is safe to execute (and in particular that does not go on to leak any such unsafe capabilities to potentially adversarial code).

$$
\begin{aligned}
\mathcal{V}_{\text{indirect}}(c) &= \text{permits\_read}(c) \rightarrow \\
&\quad \ast_{a \in \text{addrs } c} \exists P, \\
&\qquad \boxed{\exists v, a \mapsto_M v * P\ v} * \\
&\qquad \forall v, \triangleright \Box (P\ v \mathrel{-\!\!*} \mathcal{E}_{\text{gen}}(\text{clear\_lsb}(v), \text{unseal}(c), \text{lsb}(v))) \\
\mathcal{V}_{\text{sealed}}(c) &= \triangleright \mathcal{V}(\text{unseal}(c)) \\
\mathcal{V}(c) &= \text{match}(\text{tagged}(c), \text{seal\_type}(c)) \\
&\quad \mid (\text{true}, \text{Unsealed}) \qquad \Rightarrow \mathcal{V}_{\text{unsealed}}(c) \\
&\quad \mid (\text{true}, \text{IndirectSentry}) \Rightarrow \mathcal{V}_{\text{indirect}}(c) \\
&\quad \mid (\text{true}, \_) \qquad \Rightarrow \mathcal{V}_{\text{sealed}}(c) \\
&\quad \mid (\text{false}, \_) \qquad \Rightarrow \text{True}
\end{aligned}
$$

Fig. 6. The definitions of $\mathcal{V}$, $\mathcal{V}_{\text{indirect}}$, and $\mathcal{V}_{\text{sealed}}$. The latter two describe whether sealed capabilities are safe to share with an adversary. $\mathcal{V}_{\text{indirect}}$ is used for indirect sentries and $\mathcal{V}_{\text{sealed}}$ is used for all other cases. unseal clears the seal type of a capability, giving an unsealed capability with the same bounds and permissions. $\text{lsb}(c)$ returns the least significant bit of c and $\text{clear\_lsb}(c)$ returns c with its least significant bit set to 0.

It is possible to define similar notions of safety for the other kinds of sealed capability supported by Morello, but, because we do not make use of them in our encapsulation mechanism, for simplicity we define $\mathcal{V}$ to be $\mathcal{V}_{\text{sealed}}(c) = \triangleright \mathcal{V}(\text{unseal}(c))$ for all other kinds of sealed capability.

Finally, because untagged capabilities do not grant any permissions, all untagged values are safe.

Notice that $\mathcal{V}$ is persistent, because an adversary may make repeated usage and multiple copies of any capability they are granted, so we must be able to reuse and duplicate $\mathcal{V}$.

The logical relation defined by $\mathcal{V}$ and $\mathcal{E}$ is very similar to the one used by Cerise, but also differs in several ways. Most importantly, we are forced to consider alignment when taking ownership of memory, because capabilities do not fit into a single byte. For the same reason, each invariant in $\mathcal{V}_{\text{data}}(c)$ and $\mathcal{V}_{\text{indirect}}(c)$ owns several bytes of memory, not just one, potentially including some just outside the bounds of c to ensure we own entire capabilities.

In addition, we must handle the Morello-specific feature of using the least significant bit of invoked capabilities to set `C64`, which complicates the definitions of $\mathcal{E}_{gen}$ and $\mathcal{V}_{\text{indirect}}$.

Finally, because we are considering the compressed encoding of capabilities into a bitvector, rather than a structured representation, we rely on a number of helper functions to extract information from the capability instead of being able to simply pattern-match on it.

7 Unknown Code

While Islaris can substantially simplify reasoning about known code, that simplification is largely driven by knowing exactly which instructions that code contains. Confronted with unknown code Islaris is generally unable to simplify it at all. Instead we turn to an existing result providing an upper bound on the behaviour of arbitrary Morello instructions, the T-CHERI properties.

7.1 T-CHERI

The T-CHERI properties were designed by Bauereiss et al. [3] as a thin abstraction layer (a "Theoretician's CHERI") capturing essential properties that any instruction in any CHERI architecture must satisfy. They have two main thrusts. First, resources must be protected by capabilities: before accessing memory or a privileged register, an instruction has to obtain a valid capability with

sufficient permissions. For a memory access, this means that the accessed memory region has to be within the bounds of the capability, and the capability has to have the right load or store permissions. Second, new capabilities generated by instructions must be legally derived from existing capabilities that the instruction has already obtained. Legal derivation includes operations removing permissions or shrinking the bounds, but not growing them. Hence, software is not able to escalate its privileges by manipulating capabilities. The only allowed cases of producing capabilities that the currently running code might not be able to construct normally are capability invocation, where a sealed capability belonging to another compartment may be unsealed (but then control is transferred to that compartment), or processor exceptions, where a capability pointing to the exception handler is copied from a privileged register to the PCC (but then the exception handler takes control).

The formalisation of T-CHERI states these properties in terms of traces of register and memory read and write events of instructions. For example, for any event writing a capability to a register in any possible trace of any instruction, one of the following must hold:

- either the capability is derived from capabilities that have been read earlier in the trace,
- or the instruction is an invocation instruction and the register write is part of the invocation process (spelled out in some detail in the formalisation),
- or the trace ends in a processor exception, and the capability was read from an exception handler register and is being written to PCC.

Bauereiss et al. formalised this and the other T-CHERI properties in Isabelle and proved them for Morello, using the Sail model of the architecture derived from Arm's ASL specification and translated into Isabelle.

The T-CHERI properties are sufficient to prove the headline result of [3], reachable capability monotonicity, stating that arbitrary code cannot escalate its privileges by forging capabilities that it couldn't have legally derived from the capabilities it has available from the start. The property holds for arbitrary sequences of instructions, unless and until a capability invocation or processor exception occurs, passing control to another security domain with a potentially different set of reachable capabilities.

However, when we tried to use the existing T-CHERI properties in our proofs, we found that their characterisation of capability invocation is not precise enough for our purposes. For example, in the case of invocations involving a pair of sealed code and data capabilities, the original T-CHERI properties specify when and how each of the two capabilities may be unsealed and invoked, but it does not specify that both capabilities must be invoked together atomically. This leaves open the possibility of an architecture bug where an invocation instruction only unseals the data capability without invoking the code capability, thereby giving the currently running code access to the data of another compartment. This issue is technically out of scope of the original monotonicity proof by Bauereiss et al., because that property stops making guarantees at the point of capability invocations, but for our proofs we need to go beyond those points of security domain transitions to be able to reason about back and forth interactions between known and unknown code.

We therefore extend T-CHERI by adding two properties that specify capability invocation in more detail. First, in order to solve the above problem with the invocation of pairs of capabilities, we require that for any execution of an instruction that may invoke a data capability in addition to a code capability, one of the following holds:

- either the instruction writes both the expected code and data capabilities to the PCC and C29 registers, respectively,
- or the instruction raises an exception and does not write to the C29 register (or writes an untagged capability to PCC, which will cause an exception on fetching the next instruction).

```
definition "idc_write_axiom CC ISA t =
    (trace_invokes_data_caps ISA t = {} ∨
     (∃cc cd. trace_writes_pcc_caps ISA t = {cc} ∧ trace_writes_idc_caps ISA t = {cd} ∧
              (is_tagged_method CC cc ⟶ cc ∈ trace_invokes_code_caps ISA t ∧
                                         cd ∈ trace_invokes_data_caps ISA t)) ∨
     (∃cc. trace_raises_ex ISA t ∧
           trace_writes_pcc_caps ISA t = {cc} ∧
           trace_writes_idc_caps ISA t = {}) ∨
     trace_has_assertion_failure ISA t)"
```

Fig. 7. Isabelle formalisation of the capability pair invocation property

```
Definition idc_write_prop (isa : ISA) (t :  isa_trace) :=
    cap_set_empty (trace_invokes_data_caps_contains isa t) ∨
    (∃ cc cd, trace_writes_pcc_caps isa t = [cc] ∧ trace_writes_idc_caps isa t = [cd] ∧
              (Cap.cap_is_valid cc → (trace_invokes_code_caps_contains isa t cc ∧
                                      trace_invokes_data_caps_contains isa t cd))) ∨
    (∃ cc, isa.( trace_raises_ex) t  ∧
           trace_writes_pcc_caps isa t = [cc] ∧
           trace_writes_idc_caps isa t = [])  ∨
    isa.( trace_has_assertion_failure) t.
```

Fig. 8. Rocq formalisation of the capability pair invocation property

In the latter case, control will pass to the exception handler in the operating system (which we trust in the context of this work).

The second property we add is specific to Morello and spells out the different kinds of invocation more precisely than the original T-CHERI properties. For example, in the case of an indirect sentry invocation instruction, the original properties state that the instruction may invoke a code capability loaded from memory, but they do not stop the instruction from loading multiple capabilities and invoking either one of them. Our new property states that such an instruction performs exactly one memory load, which becomes the invoked code capability, and the invoked data capability is the indirect sentry in the source register. Similarly, for invocation instructions that load both code and data capabilities from memory, we state that the instruction performs exactly two loads from adjacent memory addresses, with the capability loaded from the lower address becoming the invoked data capability and the other becoming the invoked code capability. Other kinds of invocation are similarly spelled out in the form of a big case split. Additionally, we state another Morello-specific requirement, namely that all invocation instructions will set `C64` to the least significant bit of the address of the invoked code capability (controlled by the sealer), ensuring that after invocation execution resumes in the expected processor mode.

We formalised these new properties in Isabelle. Figure 7 shows the Isabelle formalisation of the property about invocations of pairs of code and data capabilities described above. It is a property of traces of register and memory read and write effects of instructions, and the parameter t contains such a trace along with an annotation specifying which instruction has generated it. The property can be instantiated for different CHERI architectures, and takes the relevant information about

architecture-specific details in the parameters CC and ISA. The CC parameter is a record containing implementations of methods of a capability type-class. For example, the capability representation in the Morello specification is just a 129-bit word, and is_tagged_method checks the tag in the most significant bit. The ISA parameter contains architecture-specific information such as details about which instructions are allowed to perform capability invocations and how, used here in particular by the functions trace_invokes_{code,data}_caps. The Morello branch-and-link-register instruction blr [c29], for example, invokes the capability in register C29 as a data capability if it is an indirect sentry, and the capability loaded from memory as the code capability. The property uses all this information to spell out the different allowed cases described above: either the instruction does not invoke a data capability, or it successfully invokes the expected pair of code and data capabilities, or it raises an exception, or the given trace ends prematurely due to an assertion failure in the specification (we have not verified the absence of assertion failures in the original Morello specification, and assume that they do not happen).

We proved our new properties for the instructions in Morello, strengthened the existing T-CHERI properties in a few places (e.g. adding some more details about the different kinds of capability invocation), and fixed the existing proofs. Due to the size and complexity of the full Morello ISA specification (about 200,000 lines of Isabelle definitions), any such proof requires serious engineering effort. For their monotonicity proof, Bauereiss et al. [3] report 24 person-months of effort writing around 17,000 lines of manual proof, plus 37,000 lines of auto-generated lemmas about the instructions and auxiliary functions in the specification. Their automation is tailor-made to the T-CHERI properties rather than aiming to be generic, and it proved effective in handling the bulk of the architecture, but the proof did require significant manual effort and tuning of the automation. In order to further break down the whole-instruction T-CHERI properties into properties that can be proved of individual auxiliary functions used by the instructions, Bauereiss et al. defined helper predicates capturing the T-CHERI properties for partial traces, taking as a parameter a ghost state summarising the preceding part of the trace, in particular which capabilities have been accessed. They then manually proved these properties for the builtin Sail functions and combinators as well as key helper functions in the Morello specification, e.g. for accessing memory or capability registers, or manipulating capabilities. They also defined other useful helper predicates, e.g. stating that a given function does not access memory or capability registers, and therefore trivially satisfies the T-CHERI properties. They then developed custom tactics and tools to prove the properties for the bulk of the architecture. This includes a tool to automatically generate lemmas about all helper functions and instructions in topological order, with the ability to manually tune selected lemmas and proofs with additional preconditions or hints. This is an external tool that generates theory files which are checked by Isabelle, so the tool itself does not need to be trusted.

We reuse much of this existing tooling and the lemmas generated by it, e.g. about the absence of memory accesses in auxiliary functions, but not all of it is applicable to our new properties, which have a different shape than the original T-CHERI properties: rather than checking all events in traces with respect to their trace prefix, we have to show that specific events (like the PCC and C29 writes in our first property above) are guaranteed to happen at some point in the trace. Rather than generalising the existing tooling, we formalised a small Hoare logic to prove the new properties, again w.r.t. a ghost state that keeps track of capability reads and writes. We manually proved Hoare triples for the key auxiliary functions, including in some cases different triples for the same function for use in different invocation instructions (for example, the code capability given to BranchXToCapability has to come from different sources depending on the kind of invocation the instruction is performing). We used basic proof tactics that require manual hints on which of those triples to apply. Fortunately, we had to prove our new properties only for the 16 instructions that can perform capability invocation (as the properties hold trivially for the remaining instructions),

so a more manual approach was feasible. The proofs of our new properties consist of about 3,600 lines, and we spent several person-months writing these proofs and integrating them into the existing ones.

While the proofs are large, the T-CHERI properties are relatively compact. We manually transcribed both the original T-CHERI properties and our new ones from Isabelle into Rocq and assumed them there instead of re-proving them in Rocq. Figure 8 shows the Rocq translation of the Isabelle property in Figure 7. There are some differences in the representation types, to make each version idiomatic for the respective prover, but we took care to otherwise keep the formalisations as close to each other as possible to enable straightforward checking that there are no transcription errors.

7.2 Fundamental Theorem of the Logical Relation (FTLR)

We express the safety of unknown code as the fundamental theorem of the logical relation described in Section 6:

Theorem 7.1 (FTLR). $\forall pcc, \mathcal{V}(pcc) \twoheadrightarrow \mathcal{E}(pcc)$.

We prove theorem 7.1 by Löb induction, taking $\triangleright(\forall c, \mathcal{V}(c) \twoheadrightarrow \mathcal{E}(c))$ as the induction hypothesis.

Unfolding the definition of $\mathcal{E}$, theorem 7.1 essentially says that if all registers contain safe values then the machine executes safely. If the value in the program counter, *pcc*, is sealed or lacks the executable permission, the machine will fault immediately, which is safe, so we need only consider cases where *pcc* is unsealed and has the executable permission. Therefore, we can extract $\mathcal{V}_{\text{executable}}(pcc)$ from $\mathcal{V}(pcc)$ and conclude that the instruction to be executed is unknown, so the machine's first step of execution will load an arbitrary trace satisfying the T-CHERI properties. Taking this step allows us to clear the later modality in the induction hypothesis, so we are required to prove that given safe register values and an instruction trace satisfying the T-CHERI properties the machine executes safely, assuming that the FTLR holds. Note that we cannot immediately apply the induction hypothesis at this point to conclude the proof, because we are trying to prove safe execution from a state where the machine has loaded a trace to execute, while $\mathcal{E}$ is stated in terms of safe execution from a state where the machine has finished executing a previous trace and is about to load a new one.

This proof has three major components. First we prove that any capability derived from a collection of capabilities satisfying $\mathcal{V}$ also satisfies $\mathcal{V}$ (Lemma 7.2). Second we prove that each event of a trace satisfying the T-CHERI properties executes safely, keeping track of various properties of the prefix of the trace that has been executed (Lemma 7.3). Finally we prove that if all the events of a T-CHERI compliant trace have executed the machine will go on to execute safely (Lemma 7.4). All these lemmas are proven under the assumption of the above induction hypothesis.

Lemma 7.2. *For any set of capabilities S, if $\forall c \in S, \mathcal{V}(c)$ then any capability c' derivable from S satisfies $\mathcal{V}(c')$.*

We prove Lemma 7.2 by case analysis on capability derivations. A new capability, c', is derivable from a set, S, of capabilities if it corresponds to a narrowing of the bounds or permissions of an unsealed capability in S, in which case it is clear that $\mathcal{V}_{\text{unsealed}}$ is preserved; or it is the result of unsealing a capability in S using an unsealed capability in S with the unsealing permission, which cannot happen because $\mathcal{V}$ forbids unsealed capabilities with the unsealing permission; or it is the result of sealing a capability, $c \in S$, where c is unsealed and $c = \text{unseal}(c')$.

For this last case if the capability is sealed with any seal type other than that of an indirect sentry capability $\mathcal{V}_{\text{sealed}}(c')$ is just $\mathcal{V}(\text{unseal}(c'))$ which holds because we have $\mathcal{V}(c)$ and $\text{unseal}(c') = c$. If the capability is sealed as an indirect sentry we prove $\mathcal{V}_{\text{indirect}}(c')$ by assuming c' permits reads and extracting P such that $\boxed{\exists v, a \mapsto_M v * P\ v}$ and $\forall v, P\ v \twoheadrightarrow \mathcal{V}(v)$ for all a in the bounds of c' from

$$\text{trace\_correct\_cap } c \text{ writes} = (\text{last}(\text{writes}) = \text{Some}(c)) \lor (\text{last}(\text{writes}) = \text{None} * \mathcal{V}(c))$$

$$\begin{aligned}\text{memory\_reads\_safe } t\ t' = \forall c, c \in \text{trace\_reads\_mem\_cap } t \twoheadrightarrow \\ \exists s,\ (\text{trace\_load\_auth } t'\ s * \\ (\text{seal\_type}(s) = \text{IndirectSentry} \twoheadrightarrow \\ (s \in \text{invokes\_indirect\_cap}(t') * \\ \mathcal{E}_{gen}(\text{clear\_lsb}(c), \text{unseal}(s), \text{lsb}(c)))) * \\ (\text{seal\_type}(s) \neq \text{IndirectSentry} \twoheadrightarrow \mathcal{V}(c)))\end{aligned}$$

$$\begin{aligned}\text{trace\_prefix\_safe } t\ t' = \ & \mathop{\Large *}_{r \in \text{gprs}} (\exists v, \mathcal{V}(v) * r \mapsto_R v) * \\ & (\forall c, \text{available\_caps } t \twoheadrightarrow \mathcal{V}(c)) * \\ & (\exists c, \text{“PCC”} \mapsto_R c * (\mathcal{V}(c) \lor c \in \text{invokes\_code\_cap } t') \\ & \quad * \text{trace\_correct\_cap } c\ (\text{pcc\_writes } t)) * \\ & (\exists c, \text{“C29”} \mapsto_R c * (\mathcal{V}(c) \lor c \in \text{invokes\_data\_cap } t') \\ & \quad * \text{trace\_correct\_cap } c\ (\text{idc\_writes } t)) * \\ & (\exists b, \text{“C64”} \mapsto_R b * \text{last } (\text{c64\_writes}) \in \{\text{Some } b, \text{None}\}) * \\ & (\forall c, c \in \text{trace\_reads\_reg\_cap} \twoheadrightarrow \mathcal{V}(c)) * \\ & \text{memory\_reads\_safe } t\ t'\end{aligned}$$

Fig. 9. trace_prefix_safe t t′ describes safe machine states when the machine has executed a prefix, t, of a trace, t', which satisfies the T-CHERI properties. It takes ownership of all registers; ensures the values of “PCC”, “C29” and “C64” are sensible; and requires all capabilities that might be used at this point in the trace satisfy $\mathcal{V}$ or are part of safe invocations.
available_caps is the set of capabilities previously read by a trace, excluding those read from memory if the instruction is an invocation.
invokes_{code, data, indirect}_cap are the sets of capabilities that may be invoked by a trace.
trace_reads_{reg, mem}_cap are the sets capabilities read from registers and memory respectively by a trace, excluding those read after a write to the same location in the same trace.
trace_load_auth t s means s is a capability used to justify memory reads in trace t.
While parts of this definition may initially seem redundant, subtle differences in the meaning of the various T-CHERI helper functions mean this form aligns very well with the properties we use while proving the FTLR.

$\mathcal{V}_{\text{data}}(c)$ (using the fact that sealing a capability does not change its bounds or permissions). We use our induction hypothesis to conclude $\forall v, P\ v \twoheadrightarrow \mathcal{E}(v)$ and unfold $\mathcal{E}$ to $\mathcal{E}_{gen}$ to conclude the proof.

Lemma 7.3. *If t' is a trace satisfying the T-CHERI properties and t is a prefix of t', t executes safely from an initial state in which all registers satisfy $\mathcal{V}$, and once t has executed,* trace_prefix_safe t t' *holds.*

We prove Lemma 7.3 by induction on the structure of t'. The base case is straightforward, since in the initial state all registers including “PCC” and “C29” satisfy $\mathcal{V}$ and available_caps, trace_reads_{reg, mem}_cap and {pcc, idc, c64}_writes are all empty. The inductive case is proved

by case analysis over the possible types of event. For simplicity we will describe only one representative example, of a register write. The T-CHERI properties tells us that a register write after prefix t must either write a capability derivable from available_caps t or must write an invoked capability to PCC or C64. In the first case, since every value of available_caps t satisfies $\mathcal{V}$, by Lemma 7.2 the written value also satisfies $\mathcal{V}$. In the second case we learn the written value is in one of invokes_{code, data}_cap t. In either case it is therefore possible to update the appropriate clause of trace_prefix_safe, noting that the Some branch of trace_correct_cap holds if required.

Lemma 7.4. *If t' is a trace satisfying the T-CHERI properties, and* trace_prefix_safe t' t' *holds, then* WP Fetch *holds.*

Finally we prove Lemma 7.4 by case splitting on whether or not the trace t' describes an invocation. If t' does not describe an invocation then invokes_{code, data}_cap are empty, so we have capabilities *pcc* in the program counter and *idc* in C29 with $\mathcal{V}(pcc)$ and $\mathcal{V}(idc)$. Then by the induction hypothesis of the FTLR we have $\mathcal{E}(pcc)$ which we can apply with $\mathcal{V}(idc)$ and our ownership of C29 and C64 to conclude WP Fetch.

If t' does describe an invocation a more nuanced argument is needed, the exact form of which depends on the kind of invocation described. To handle this we use the Morello specific large case split described in Section 7.1, which describes the possible behaviours of each kind of invocation.

Because the definition of $\mathcal{V}$ is most interesting for indirect sentry capabilities we will examine the proof for that case as an example. We learn from the case split that a data capability, s with the IndirectSentry seal type and some address a has been invoked, and that a code capability, c, has been read from address a and invoked. We know from idc_write_prop (Fig. 8) that since the instruction did not raise an exception these two invoked capabilities were the only values written to C29 and PCC, and so from trace_correct_cap we know that they are the current values in those registers. We also learn from the invocation case split that the invoked data capability is the only capability satisfying trace_load_auth for this trace. Since c was read from memory by the trace we can unfold memory_reads_safe to learn $\mathcal{E}_{gen}(\text{clear\_lsb}(c), \text{unseal}(s), \text{lsb}(c))$ which we can combine with the register ownership in trace_prefix_safe to get WP Fetch as required.

Using Lemma 7.3 we prove that the initially loaded instruction executes safely, and then apply Lemma 7.4 to prove WP Fetch and therefore that the machine executes safely, proving the FTLR.

Formalising this proof of the fundamental theorem in Rocq required around 6.5k lines of proof and intermediate definitions, and approximately six person months of work. We were surprised that most of the Rocq proofs did not require detailed reasoning about the capability representation. This reasoning is largely confined to the Isabelle proofs of the T-CHERI properties and the proof of Lemma 7.2, with the rest of the Rocq proofs being mostly phrased in terms of helper decoding functions that rarely need to be unfolded.

8 Known Code

To prepare the known code for verification we follow a similar process to the one used for standalone program verification in Islaris [17]. We compile the assembly program in Figure 2 on a Morello board, and use Islaris's tooling to generate a Rocq embedding of the Isla instruction traces for the instructions in the resulting binary.

We focus our formalization on the compartmentalization of the routine in Figure 2 and prove that the stored counter only takes values between 0 and 63, by proving an Iris specification for the code which includes invariant (1), in which bv 128 is the type of a bitvector of length 128 used to represent a capability minus its *tag*, $data + 16 \mapsto_M (tag, v)$ takes ownership of the 16 bytes of memory at address $data + 16$, asserting that they have tag *tag* and store value v. $v[0..31]$ is the integer representation of the lowest 32 bits of v and $\lceil\ \rceil$ embeds a pure Rocq proposition into Iris:

$$\boxed{\exists(v : \text{bv } 128)(tag : \text{bool}), (data + 16) \mapsto_M (tag, v) * \lceil tag = \text{false} \wedge 0 \leq v[0..31] < 64 \rceil} \quad (1)$$

Assuming that the code and data capabilities to the encapsulated routine have been initialised correctly as described in the following theorem, we can prove that the sealed data capability is safe to share and the code capability is safe to execute.

Theorem 8.1. *If*

- *cd is a valid indirect sentry capability that has been sealed with the* lb *sealing type, has load and store permission, contains a 16-byte aligned address, and points to a 20-byte memory region containing a code capability cc and a 4-byte counter value between 0 and 63, where*
- *cc is a valid capability that has execute permission, points to the first instruction of the incrementer routine (Figure 2, line 2), and has the least significant bit set to* 1 *(so that invoking it sets the* C64 *mode bit also to* 1*),*

then $\mathcal{V}(cd)$ *holds.*

Our proof takes the following approach: 1) For each instruction of the routine, we define and prove a suitable specification for the generated Islaris trace. As in the existing Islaris work these specifications are written in a continuation passing style, where the specification of each instruction takes as an assumption that execution from the following instruction is safe. 2) We then prove that a code capability *cc* satisfying our initialisation assumptions is safe to execute (i.e., $\mathcal{E}(cc)$), preserving invariant (1). 3) Finally, we prove that a data capability *cd* satisfying our initialisation assumptions is safe to share (i.e., $\mathcal{V}(cd)$).

To make this verification effort feasible, several challenges needed to be addressed. As discussed in [17], in a real-world architecture even relatively simple instructions such as add have a large and complex definition. Isla helps simplify these significantly by pruning away unreachable execution paths. Our assumption that a number of system registers have fixed values significantly helps Isla to prune away paths that are not relevant to our proof. In addition, since we assume exceptions to be safe, we can have Isla prune the tail of traces once they have begun to take an exception. Finally we can manually annotate instructions in the known code with additional constraints that help simplify the generated traces, although for soundness Islaris will then require us to prove that those constraints hold when executing those instructions. In our incrementer example, we annotate the instructions ldr w0,[C29,16] and str w0,[C29,16] with the constraint that the data capability in C29 has load permission and a suitable alignment, which follows from our invariant. This allows Isla to rule out a number of traces, for example, traces that split up a misaligned access into a number of smaller requests and then reassemble the data.

Even with these simplifications in place, we are left with the complexity inherent to some instructions' behaviour that cannot be automatically simplified away. In general, instruction traces are more complex for Morello than for Arm-A because instruction behaviour often depends on if and how the different fields in a given capability are set, so apparently simple instructions can have many subtly different behaviours. For example, the instruction ret, used in the incrementer routine to return execution to the caller, has different behaviour depending on whether the return pointer (C30) is tagged or sealed and how it is sealed among other things. This results in a trace with a total of 23 different branches even after simplification, which we handle by adapting Islaris's proof search for Morello.

Islaris's proof search is designed to automatically find necessary resources in the separation logic context. For memory accesses and instruction fetch this requires Islaris to determine whether bitvectors representing addresses are equal. This is done using a bitvector equality solver principally based on the linear arithmetic solver *lia*. Unfortunately because CHERI capabilities are stored in a compressed format, for which the decompression calculations are complicated and non-linear, Islaris

is prone to hanging trying to search the separation logic context. This is particularly troublesome for instruction fetch. Because the program counter is a capability, it has to be fully decompressed and recompressed every time it is incremented. The correct calculations for this are generated by Isla and evaluated by Islaris, but they are too complex for Islaris's automation to handle in general, meaning automation stalls at the end of every instruction. To handle this we wrote a more fine tuned simplifier in Rocq, which takes advantage of the known structure of the compression to pre-simplify addresses before Islaris searches for matching resources.

One subtle incompatability between Islaris's existing automation and this work, caused by the fact that Islaris is designed to reason about entirely known code, is that the definition of safe execution from a machine state understood by Islaris's automation assumes that the trace of the first instruction to be executed is known concretely. This is not the case for the return instruction from our incrementer routine, since the next instruction is expected to be unknown code. To handle this we modified Islaris's automation to be able to use a subtly different statement of safe execution, at the cost of slightly worse automated handling of later modalities. As in Islaris these specifications can be thought of as Hoare doubles, which are essentially standard Hoare triples with a postcondition of True. Islaris uses Hoare doubles instead of Hoare triples because of the ill defined nature of termination for realistic ISAs. Note that despite the lack of a post condition Islaris Hoare doubles are not vacuous, as they still assert that the program executes safely, without accessing resources it does not have ownership of or violating invariants.

As an example of the instruction specifications proven for our example, we consider the specification of the first instruction as the following Hoare double: $\{P\}\ a_{ldr}$, where a_{ldr} is the address of instruction ldr w0,[c29,16] and

$$P = \exists v_{pcc}, v_{29}, v_0.\, c_{29} \mapsto_R v_{29} * r_0 \mapsto_R v_0 * PCC \mapsto_R v_{pcc} * \texttt{C64} \mapsto_R 1 * \ldots * \tag{2}$$

$$\lceil v_{pcc}[0..63] = a_{ldr} \rceil * \boxed{\exists (v : \text{bv } 128).\, c29 + 16 \mapsto_M (v, \text{false}) * \lceil 0 \le v[0..31] < 64 \rceil} * \tag{3}$$

$$\{\, c_{29} \mapsto_R v_{29} * \exists v'_{pcc}, v_0.\, r_0 \mapsto_R v_0 * PCC \mapsto_R v'_{pcc} * \lceil v'_{pcc}[0..63] = v_{pcc}[0..63] + 4 \rceil * \tag{4}$$

$$\lceil 0 \le v_0[0..31] < 64 \rceil * \texttt{C64} \mapsto_R 1 * \ldots \}\ (a_{ldr} + 4). \tag{5}$$

The pre-condition P starts by taking ownership of the registers used by ldr, including C64 to determine the instruction decode mode (2). We omit here ownership of system registers, as well as assumptions on $c29$'s alignment, validity, bounds, permissions and that it is unsealed (recall that blr automatically unseals $c29$ on invocation). It also requires PCC to be pointing to the ldr instruction (3) and that invariant (1) holds. Finally, it assumes safe execution from the next instruction, add (4-5), the pre-condition of which essentially acts as ldr's post-condition: v_{29} remains accessible and unchanged (which is necessary for the correct execution of str w0,[c29,16] on line 5), PCC has been incremented correctly, r_0 is owned and its value is bounded appropriately, C64 has not changed and we still own the system registers (omitted here). Note that we need not assert that v_0 will match the v pointed to by $c_{29} + 16$, as for safety it suffices to show it preserves the invariant condition.

Corollary 8.2. *From an initial machine state in which all capabilities reachable from the set of capabilities initially in registers are either capabilities for unprotected memory, or correctly initialised data capabilities as described in Theorem 8.1, after executing the machine for any number of steps, the value in the protected counter will be between 0 and 63.*

To prove Corollary 8.2, we establish $\mathcal{V}$ for each of the capabilities initially in registers, then if the value in the program counter is *pcc* apply Theorem 7.1 to $\mathcal{V}(pcc)$ to establish $\mathcal{E}(pcc)$. Unfolding $\mathcal{E}$ and specialising with $\mathcal{V}$ for each register value gives WP Fetch which can be combined with the Islaris adequacy theorem to prove that the data invariant holds at every step of execution, and in particular that the protected counter's value is between 0 and 63.

9 Related Work

Serval [16] builds on the Rosette symbolic execution framework to provide 'push-button' verification of isolation properties for systems code, rather than connecting to a general-purpose interactive theorem prover like Rocq or Isabelle to direct the proof manually. The machine model is written specifically for Rosette, hints are used to control state space explosion, and the approach exploits the fact that the execution paths of interest (such as exception handlers) are bounded. They use a more symbolic architectural model and have a more restricted logic for expressing properties.

Closer to our work, Huyghebaert et al. [12] use a verified symbolic execution in Rocq called Katamaran for proving similar properties of small systems, including a minimal capability machine. The ISA semantics are written in μSail, a core calculus of Sail deeply embedded in Rocq. This provides a more foundational result because all of the reasoning takes place inside Rocq, but it still remains to scale it up to full Sail models, both in terms of the specification language and the ISA.

VeriCHERI [4] proves that a CHERI hardware design is secure, in the sense of enforcing non-interference between mutually distrusting compartments. They establish this property directly over an RTL design and so are able to identify microarchitectural information leaks not present in the ISA definition. Like the existing T-CHERI work, VeriCHERI considers safety up to domain transition, so does not prove interacting compartments remain secure after capability invocation.

While the simplest versions of Cerise achieve very similar properties to this work, for significantly simplified machines, later versions of Cerise [8, 9] establish much stronger properties. These developments use capabilities to protect the stack in such a way that well bracketed control flow is enforced, and prove that this gives rise to temporal safety, rather than just the spatial safety we consider. To achieve this in addition to still considering a highly simplified machine they also use specialised capabilities that are not part of existing CHERI ISAs.

10 Conclusion and Future Work

We present a mechanised proof (available in the artifact accompanying the paper [11]) that the Morello ISA's architectural security mechanisms can be used to encapsulate software compartments, such that they can run safely while sharing an address space with arbitrary adversarial code. By mechanising this result we achieve a high degree of confidence that the security properties hold regardless of the behaviour of the adversary, which would be impossible with testing alone. In addition to demonstrating that such a proof is possible on a small example we provide a reusable proof that any compartment which satisfies reasonable isolation conditions, expressed in the form of a logical relation, is secure in a similar way. By building our proof on top of the existing T-CHERI properties we demonstrate that a modest enhancement of the T-CHERI properties does in fact strongly constrain an architecture to provide useable security guarantees, even in the presence of domain transitions between distrusting compartments.

Since this work demonstrates that the basic version of Cerise can be adapted to a realistic hardware architecture, one natural extension would be to adapt the more sophisticated temporal safety forms of Cerise to realistic CHERI architectures. Other than increased scale and complexity, the principle challenge here is that Temporal Cerise makes use of complex capability types that are not a part of the Morello architecture, and alternative designs require extremely slow stack clearing procedures. Consequently efficient implementation of Temporal Cerise on Morello requires either new ISA design or efficient software emulation of these unusual capabilities.

Finally, while this work demonstrates that such proofs are feasible, it also highlights the need for better proof automation for reasoning about real-world ISA semantics, so that similar work could be done more routinely.

Acknowledgments

We thank Aïna Linn Georges and June Rousseau for useful conversations about the details of the Cerise formalisation, and Simon Spies for his help with a number of Iris technical questions.

This work was funded in part by Arm. This work was supported in part by the Innovate UK project Digital Security by Design (DSbD) Technology Platform Prototype, 105694. This work was funded in part by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for ERC-AdG-2022, EP/Y035976/1 SAFER. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 789108, ERC-AdG-2017 ELVER). Funded by the European Union. 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. The authors would like to thank the Isaac Newton Institute for Mathematical Sciences, Cambridge, for support and hospitality during the programme Big Specification, where work on this paper was undertaken. This work was supported by EPSRC grant EP/Z000580/1.

References

[1] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert M. 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, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 641–653. doi:10.1145/3613424.3614266

[2] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification.* Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[3] 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, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 174–203. doi:10.1007/978-3-030-99336-8_7

[4] Anna Lena Duque Antón, Johannes Müller, Philipp Schmitz, Tobias Jauch, Alex Wezel, Lucas Deutschmann, Mohammad Rahmani Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2024. VeriCHERI: Exhaustive Formal Security Verification of CHERI at the RTL. In *Accepted for publication at the 43rd International Conference on Computer-Aided Design (ICCAD '24)* (New York, USA). Association for Computing Machinery. doi:10.1145/3676536.3676841

[5] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert M. 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 251–268. doi:10.1145/3620665.3640416

[6] Dapeng Gao and Robert N. M. Watson. 2023. Library-based Compartmentalisation on CHERI. Extended abstract, PLARCH 2023. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/202306-plarch-library-based-compartmentalisation.pdf

[7] 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 (2024), 3:1–3:59. doi:10.1145/3623510

[8] 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. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434287

[9] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. doi:10.1145/3527318

[10] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform - Validating CHERI-Based Security in a High-Performance System. *IEEE*

Micro 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[11] Angus Hammond, Ricardo Almeida, Thomas Bauereiss, Brian Campbell, Ian Stark, and Peter Sewell. 2025. *Morello Cerise Artifact.* doi:10.5281/zenodo.15047228

[12] 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

[13] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[14] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.* 6, ICFP (2022), 194–224. doi:10.1145/3547628

[15] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[16] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 225–242. doi:10.1145/3341301.3359641

[17] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 825–840. doi:10.1145/3519939.3523434

[18] 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 Markettos, 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 UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[19] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st annual international symposium on Computer architecuture.* IEEE Press, 457–468.

Received 2024-11-15; accepted 2025-03-06