Proving capability safety in the presence of indirect sentries
Technical report

June Rousseau   Aïna Linn Georges   Jean Pichon-Pharabod
Dominique Devriese   Lars Birkedal

January, 2024

Abstract

This document explores the impact of extending the CHERI architecture with indirect sentries on the core security guarantee of CHERI, capability safety. Indirect sentries are a new type of capabilities that makes executing (potentially adversarial) code more straightforward, and in particular avoids boilerplate trampoline code when creating a form of compartment (closures that work with a specific context). We extend Cerise, a simplified capability machine, with indirect sentries, and develop a heap-based calling convention that does not rely on trampoline code. We then prove capability safety and illustrate it on key examples demonstrating local state encapsulation.

Keywords: hardware capabilities, capability safety, data closure, local state encapsulation, CHERI, Cerise, robust safety.

Contents

1 Indirect sentries, informally . . . **2**

2 The Cerise machine . . . **4**

3 Program Logic . . . **7**
- 3.1 Adequacy . . . 9

4 Logical Relation . . . **10**
- 4.1 Context: (unityped) logical relations for capability machines . . . 10
- 4.2 Our logical relation for indirect sentries . . . 11
- 4.3 Fundamental Theorem . . . 12

5 Case Studies . . . **17**
- 5.1 Counter closure . . . 17
- 5.2 Heap-based calling convention . . . 19
- 5.3 Sharing a sub-buffer . . . 21

6 Related Work . . . **25**
- 6.1 Capability monotonicity . . . 25
- 6.2 Points-to-PCC indirect sentry capabilities . . . 25
- 6.3 Implementing encapsulated closures with sentry capabilities . . . 25
- 6.4 Implementing encapsulated closures with sealed pairs . . . 26
- 6.5 Cerise's flavours . . . 27

7 Future Work **27**
7.1 Secure Stack-based Calling Convention 27
7.2 Technical Improvement 29

A Full Operational Semantics **35**

B Full spec example call **35**

1 Indirect sentries, informally

Capability machines Capability machines enable fine-grained memory safety by working with hardware capabilities, which are unforgeable tokens of authority. Each word in memory and in registers can be either a plain integer, or a capability, which can only be derived from other capabilities with more authority (this is called "capability monotonicity"). Typically, a capability is a fat pointer, containing not only an address, but also permissions and bounds. Memory accesses then require appropriate capabilities, and cause a fault otherwise, which preserves safety: the unauthorised action is not performed. For example, a write to memory expects, in its destination register, a capability that witnesses the authority to write to that address.

Encapsulated closures While capabilities rule out most of the traditional memory safety vulnerabilities due to the pointer-as-integer representation, capabilities can also be used to encode more elaborate abstractions (as common in higher-level languages). In particular, capabilities make it possible to create closures that enforce local state encapsulation (LSE). A more specific use case is that of a closure that encapsulate private local state, and that can be safely invoked by untrusted code. Switching to a different closure also changes the authority. That way, encapsulated closures with completely different authorities can invoke each other without putting their own private state at risk. This is the use case that we focus on in this document, and which we use in a calling convention that enforces local state encapsulation by dynamically creating closures.

Indirect sentries In this document, we explore the implementation of encapsulated closures using indirect sentries, a new type of capability developed as part of CHERI [17], a capability architecture developed over the last decade in collaboration between the University of Cambridge, SRI International, and Arm.

The purpose of indirect sentry capabilities is to facilitate the construction of closures with a fixed entry point and a specific context. The CHERI ISA-V9 [17] proposes two flavours of indirect sentry capabilities, and the Morello documentation implements both of them: points-to-pair [17, §C.8.2] [3, §4.4.10(BLR)], and points-to-PCC [17, §C.8.1] [3, §4.4.75(LDBPR)]. In this work, we focus on the 'points-to-pair' flavour. In Cerise [7], we model a *'points-to-pair' indirect sentry capability* as a capability with a new IE permission (in CHERI, this is implemented as a reserved 'otype'). The expectation is that it points to the pair of a code capability immediately followed by a data capability. Jumping to an indirect sentry capability tries to load the word that the capability points to (which is expected to be a code capability) into `pc`, and the following word (which is expected to be a data capability) into `idc`. If the addresses are not in the bounds of the capability, the jump will fail. If these addresses do not contain capabilities, the program will likely fault at a later step. Figure 1 shows a representation of an indirect sentry $(\textsc{ie}, b, e, a)$ in memory, and Figure 2 shows the evolution of the state of the machine when jumping to an indirect sentry.

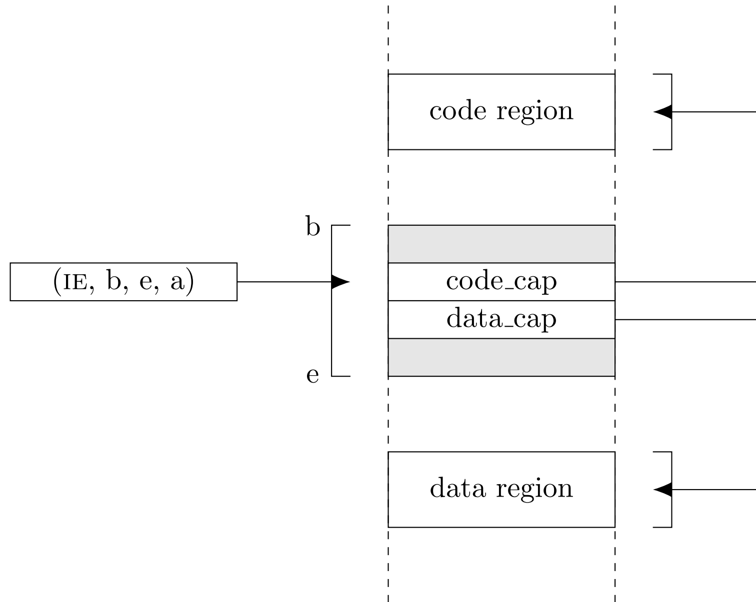
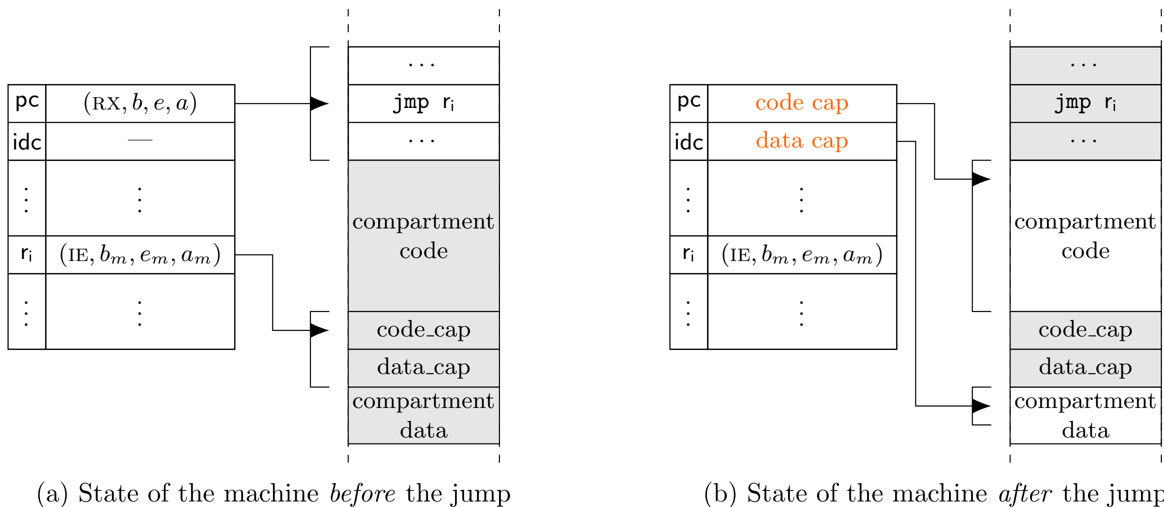


Figure 1: Indirect Sentry capability (points-to-pair flavour)



(a) State of the machine *before* the jump

(b) State of the machine *after* the jump

Figure 2: Jump to an Indirect Sentry. Parts of the memory that the machine does not have direct access to is greyed out .

Indirect sentries makes it straightforward to encode an encapsulated closure, namely by writing the closure's code capability and data capability to two adjacent addresses, and restricting a capability to these addresses to the IE permission.

Why indirect sentries? / Other types of closure capabilities CHERI supports two other ways to encode closures: pairs of sealed capabilities, and sentry capabilities. Pairs of sealed capabilities are very flexible, and make it possible to invoke the same code with different private states, but rely on a limited pool of identifiers (otypes), and they are moreover quite difficult to formally reason about. On the other hand, sentry capabilities require boilerplate trampoline code mixing code and data. Trampoline code mixing code and data is undesirable, because it breaks the principle of least privilege. Moreover, code and data are linked together,

and as a consequence, multiple instances of the same code for different data require duplication of code. Indirect sentry capabilities are designed to address these shortcomings in the case of encapsulated closures, where the code is called with the same data.

In this work, we want to prove that indirect sentries provide the expected security guarantee, namely capability safety. To do so, we extend Cerise, a simplified, CHERI-like capability machine, with indirect sentry capabilities. One of the purposes of Cerise is to provide a platform for researchers to explore new features for capability machines and study their security properties, and thus inform the evolution of CHERI. Cerise is defined in the Coq Proof Assistant, which has made it possible to mechanise proofs of functional correctness of programs, even in interaction with unknown, untrusted code. To verify such programs, Cerise comes with a program logic built on top of Iris, which makes it possible to write and prove the specification of programs with known code. In addition, Cerise comes with a characterisation of an upper bound on the behaviour of arbitrary instructions, in the form of a logical relation. In a sense, this logical relation works as a universal contract given by the machine, and intends to capture capability safety. This way, this logical relation makes it possible to reason about the interaction of known code with unknown code.

Contributions In this work, we extend the Cerise machine with indirect sentries, prove new program logic rules to reason about them, and extend the logical relation to capture their behaviour. In particular, the logical relation captures what it means for an indirect sentry to be safe to share with unknown code. Finally, we illustrate how this extension allows to reason about key representative programs involving non-trivial use of data closures with indirect sentries: a counter library, and a secure heap-based calling convention. We demonstrate that our calling convention is secure by using it in a sub-buffer example. Those examples are an adaptation of earlier work [7].

Our work should extend to other secure stack-based calling convention, as the one based on uninitialised and local capabilities [8], the one based on uninitialised and directed capabilities [6], or the one based on local capabilities and stack clearing [14]. They all strictly reduce the memory footprint of the calling convention, without much disadvantages, thus strengthening the motivation to support indirect sentries.

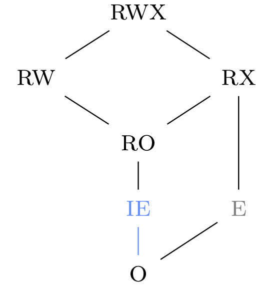
Mechanisation The results are mechanised in Coq, and are available online [13]: `https://github.com/logsem/cerise/tree/bastien/indirect-sentry`.

Plan In the following sections, we define the operational semantics of Cerise extended with indirect sentries (§2), a program logic to reason about known code (§3), and a logical relation that captures the effect of unknown code (§4). In these sections, the original 'Vanilla' Cerise model is described in black, and our additions related to indirect sentries are highlighted in blue.

2 The Cerise machine

Syntax Cerise models a capability architecture based on the CHERI architecture. Figure 3 shows the model of the machine. The machine state Σ keeps track of the state of the register file, the memory, and the current state of the machine, whether it is currently running, halted or entered a failed state due to an illegal operation. The memory of the machine Mem ranges over a finite set of addresses (up to AddrMax). The machine does not have any distinction between

$$
\begin{array}{llll}
a & \in \text{Addr} & \triangleq & [0, \text{AddrMax}] \\
p & \in \text{Perm} & ::= & \text{O} \mid \text{IE} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
c & \in \text{Cap} & \triangleq & \{(p, b, e, a) \mid b, e, a \in \text{Addr}\} \\
w & \in \text{Word} & \triangleq & \mathbb{Z} + \text{Cap} \\
reg & \in \text{RegFile} & \triangleq & \text{RegName} \to \text{Word} \\
m & \in \text{Mem} & \triangleq & \text{Addr} \to \text{Word} \\
s & \in \text{ExecState} & ::= & \mathsf{Running} \mid \mathsf{Halted} \mid \mathsf{Failed} \\
\sigma & \in \text{ExecConf} & \triangleq & \text{Reg} \times \text{Mem} \\
\Sigma & \in \text{MachineState} & \triangleq & \text{ExecState} \times \text{ExecConf}
\end{array}
$$



Lattice defining the $\preccurlyeq$ relation.

(We have $p_1 \preccurlyeq p_2$ if there is a path going up from p_1 to p_2 in the diagram.)

$$
\begin{array}{lll}
r \in \text{RegName} & ::= & \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_{31} \\
\rho \in \text{InstructionArg} & \triangleq & \mathbb{Z} + \text{RegName} \\
i \in \text{Instruction} & ::= & \texttt{jmp}\ r_{\text{dst}} \mid \texttt{jnz}\ r_{\text{dst}}\ r_{\text{cond}} \mid \\
& & \texttt{mov}\ r_{\text{res}}\ \rho_{\text{expr}} \mid \texttt{add}\ r_{\text{res}}\ \rho_{\text{e1}}\ \rho_{\text{e2}} \mid \texttt{sub}\ r_{\text{res}}\ \rho_{\text{e1}}\ \rho_{\text{e2}} \mid \texttt{lt}\ r_{\text{res}}\ \rho_{\text{e1}}\ \rho_{\text{e2}} \mid \texttt{lea}\ r_{\text{res}}\ \rho_{\text{expr}} \mid \\
& & \texttt{load}\ r_{\text{res}}\ r_{\text{src}} \mid \texttt{store}\ r_{\text{res}}\ \rho_{\text{src}} \mid \texttt{restrict}\ r_{\text{res}}\ \rho_{\text{src}} \mid \texttt{subseg}\ r_{\text{res}}\ \rho_1\ \rho_2 \mid \\
& & \texttt{isptr}\ r_{\text{res}}\ r_{\text{src}} \mid \texttt{getp}\ r_{\text{res}}\ r_{\text{src}} \mid \texttt{getb}\ r_{\text{res}}\ r_{\text{src}} \mid \texttt{gete}\ r_{\text{res}}\ r_{\text{src}} \mid \texttt{geta}\ r_{\text{res}}\ r_{\text{src}} \mid \\
& & \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Figure 3: Base definitions for the machine's words, state, and instructions. The Coq definition also has capability types and instructions for sealing and unsealing, but do not support sealed pairs. We omit those here as they are not relevant for this work.

regions of memory. In particular, it does not have a specific region for the stack. The register file RegFile ranges over the registers pc and r_0 to r_{31}. We use the name idc to refers to the register r_0. The pc register is the *program counter* register, and idc is the *invoked data* register, hereafter called the *data register*. The data register is reserved, by convention, to receive the data word of an invoked indirect sentry.

The memory and the registers manipulate machine words, that are either integers or capabilities. The machine uses integers for the sole purpose of arithmetic operations, whereas it uses capabilities to address the memory. A capability c is of the form (p, b, e, a). It points to the address a, and ranges over the addresses $[b, e)$ with the permission p. The permissions are ordered by the relation $\preccurlyeq$ given by the lattice in Figure 3. A capability permission allows to read (R), write (W) or execute (X) the content of the address it points to. On top of these usual permissions, the machine has two additional permissions to opaque closures: the entry E and the indirect sentry IE permissions. Entry and indirect sentry capabilities cannot be used to load or store in the memory, and their bounds and their address cannot be changed. The only way to use them is to jump to them. They are a way to perform a controlled domain transition, *i.e.,* change non-monotonically the authority. The execution of the machine is described in details by the operational semantics.

i	$[\![i]\!](\sigma)$	Conditions
`store` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{mem}.a \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a) \wedge b \leq a < e$ $\wedge\ p \in \{\mathrm{RW}, \mathrm{RWX}\} \wedge w = \mathrm{getWord}(\sigma, \rho)$
`restrict` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a)$ $\wedge\ p' = \mathrm{decodePerm}(\mathrm{getWord}(\sigma, \rho)) \wedge p' \preccurlyeq p$ $\wedge\ w = (p', b, e, a)$
`jmp` r	$\left(\mathsf{Executable}, \sigma\begin{bmatrix} \mathrm{reg}.\mathsf{pc} \mapsto \mathit{newPc}, \\ \mathrm{reg}.\mathsf{idc} \mapsto \mathit{newIdc} \end{bmatrix}\right)$	if $\sigma.\mathrm{reg}(r) = (\mathrm{IE}, b, e, a)$ then $b \leq a < a+1 < e$ $\wedge\ \mathit{newPc} = \sigma.\mathrm{mem}(a)$ $\wedge\ \mathit{newIdc} = \sigma.\mathrm{mem}(a+1)$ else $\mathit{newPc} = \mathrm{updatePcPerm}(\sigma.\mathrm{reg}(r))$ $\wedge\ \mathit{newIdc} = \sigma.\mathrm{reg}(\mathsf{idc})$
`jnz` $r_{\mathrm{dst}}\ r_{\mathrm{cond}}$	$\left(\begin{array}{l} \mathsf{Executable}, \\ \text{if } \sigma.\mathrm{reg}(r_{\mathrm{cond}}) \neq 0 \\ \text{then } \sigma\begin{bmatrix} \mathrm{reg}.\mathsf{pc} \mapsto \mathit{newPc}, \\ \mathrm{reg}.\mathsf{idc} \mapsto \mathit{newIdc} \end{bmatrix} \\ \text{else } \mathrm{updPC}(\sigma) \end{array}\right)$	if $\sigma.\mathrm{reg}(r_{\mathrm{dst}}) = (\mathrm{IE}, b, e, a)$ then $b \leq a < a+1 < e$ $\wedge\ \mathit{newPc} = \sigma.\mathrm{mem}(a)$ $\wedge\ \mathit{newIdc} = \sigma.\mathrm{mem}(a+1)$ else $\mathit{newPc} = \mathrm{updatePcPerm}(\sigma.\mathrm{reg}(r_{\mathrm{dst}}))$ $\wedge\ \mathit{newIdc} = \sigma.\mathrm{reg}(\mathsf{idc})$

Figure 4: Excerpt of the operational semantic of Cerise with Indirect Sentries.

Operational semantics At every step of execution, if the machine is $\mathsf{Running}$, the machine checks that the pc register contains a valid PC capability, *i.e.,* an in-bounds, executable capability. It then fetches and decodes the instruction contained in the address the PC capability points to, and executes the instruction according to its semantics. If the PC capability is not valid, if the fetched word cannot be decoded into a valid instruction, or if the instruction is illegal, the machine fails.

Figure 4 shows an excerpt of the operational semantics, for the execution of one instruction. The full operational semantics can be found in Appendix A. For example, assuming that register r contains an in-bounds write capability pointing to address a (that is, a capability (p, b, e, a) with the address a in the bounds $[b, e)$, and with the permission $p \ni \mathrm{W}$), the instruction `store` $r\ \rho$ stores the word from ρ at address a. An instruction argument $\rho \in (\mathbb{Z} + \mathrm{RegName})$ can either be an immediate (*i.e.,* an integer) or a register name.

The introduction of the indirect sentry changes the semantics of several instructions, but most of them are quite straightforward. As such, we focus on the jumping instructions (*i.e.,* `jmp` and `jnz`). The only way to use an indirect sentry is to jump (via a `jmp` or a `jnz` instruction), which performs a domain transition. Upon jumping, the machine unseals the indirect sentry pointing to a and performs two capability loads, of the two consecutive addresses a and $a+1$. For each capability load, the machine checks that both addresses a and $a+1$ are in the bounds of the capability. To construct an indirect sentry, we can use the `restrict` instruction on any capability with the read permission. In the spirit of delaying errors as much as possible, no other check is performed: the current address of the capability does not have to be in between the bounds, and the words to which the capability points to does not have other requirements.

Stack Vanilla Cerise [7], on which we build, does not have an explicit stack. Given that we are exploring calling conventions, this might seem unusual. However, the aspect we are concerned with in this work, namely Local State Encapsulation and Capability Safety, are orthogonal to the concerns of a system with explicit stack, such as Well Bracketed Control Flow and Capability Revocation. These last aspects require another features, a *locality* bit [14], to be enforced. In this document, we oppose *heap-based* calling convention §5.2 to *stack-based* calling convention §7. The former uses heap allocation to store the local state, while the latter uses an explicit stack with the locality bit.

3 Program Logic

The Cerise program logic [7, §4], as well as our extension, is build on top of the Iris Separation logic [10]. Separation logic has been widely used to reason about programs with mutable state, such as memory. Moreover, it enables modular reasoning, which allows composition of program with disjoint state. Thus, the Iris logic framework is well-suited to our purpose. Figure 5 presents the syntax of our program logic.

Base logic from Iris Iris is an impredicative higher-order separation logic. As such, Iris propositions include the usual proposition of higher-order logic: conjunction $\wedge$ and disjunction $\vee$, universal $\forall$ and existential $\exists$ quantification, *etc.*. And those of usual separation logic: the separating conjunction $*$, and the magic wand $-\!*$. The pure proposition $\lceil \varphi \rceil$ holds if the proposition φ from the meta logic holds. In addition, Iris features a persistency modality $\square$, a 'later' modality $\triangleright$, and invariants, which we quickly describe below.

- Iris propositions can be either ephemeral or persistent. Ephemeral propositions might

$$
\begin{array}{lr}
P, Q \in iProp ::= & \\
\quad \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \dots & \text{higher-order logic} \\
\quad \mid P * Q \mid P -\!\!* Q \mid \lceil \varphi \rceil \mid \square\, P \mid \triangleright P & \text{separation logic} \\
\quad \mid \boxed{P} & \text{invariants} \\
\quad \mid \mathsf{a} \mapsto w & \text{memory points-to} \\
\quad \mid r \Mapsto w & \text{register points-to} \\
\quad \mid \langle P \rangle \rightarrow \langle s.\, Q \rangle \mid \{P\} \rightsquigarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \bullet & \text{Hoare triples}
\end{array}
$$

We also write $[b, e) \mapsto \vec{l} \triangleq \mathop{\Huge *}_{i \in 0..e-b-1} (b+i) \mapsto \vec{l}[i]$ for contiguous memory region points-to.

Figure 5: Syntax of assertions

be invalidated at some point and are not duplicable. On the other hand, persistent propositions always hold, and can thus be duplicated. The proposition $\square P$ describes the persistent, duplicable part of P.

- The $\triangleright P$ proposition is largely a technicality, and we refer the reader to [10, §5.5] for details, but it roughly means that P holds "after one logical step of execution".
- An invariant $\boxed{P}$ states that P holds now and at all future steps of execution. It can be opened in one execution step to access P, but must be restored by the end of the step. For the exact details of opening and closing invariants, and for examples, we refer the reader to [10, §2.2].

On top of Iris, in Cerise, we define logical resources that allow us to relate the operational semantics with the logic:

Points-to assertions

- The resource $a \mapsto w$ is the usual points-to of separation logic, which expresses exclusive ownership of the *address* a, and knowledge that the *memory* contains the word w at that address.
- $r \Mapsto w$ is the corresponding assertion for registers: it expresses exclusive ownership of the *register* r, and knowledge that the *register file* contains the word w in that register.

Hoare Triples In order to specify programs, we define three kind of triples, similar to *Hoare triples*. Here, we write P for a proposition describing the machine state in the precondition of the triple, and $s.\, Q$ for the postcondition of the triple, where s binds the execution state in Q.

- $\langle P \rangle \rightarrow \langle s.\, Q \rangle$ describes the specification a single instruction execution. It holds if the machine starts in a state satisfying P, and ends in a state satisfying Q after one execution step.
- $\{P\} \rightsquigarrow \{s.\, Q\}$ describes the specification of a code fragment. It holds if the machine starts in a state satisfying P, and either diverges or reaches a state satisfying Q.
- $\{P\} \rightsquigarrow \bullet$ describes the complete, safe execution. It holds if the machine starts in a state satisfying P, and either diverges or runs until it halts of fail.

It is possible to sequence and interleave the different kind of specification. We refer the reader to Cerise [7, §4.2] for the exact sequencing rules.

We use the notations

$$\begin{aligned} \{P\} \rightsquigarrow \{Q\} &\triangleq \{P\} \rightsquigarrow \{s.\, \lceil s = \mathsf{Running} \rceil * Q\} \\ \langle P \rangle \rightarrow \langle Q \rangle &\triangleq \langle P \rangle \rightarrow \langle s.\, \lceil s = \mathsf{Running} \rceil * Q \rangle \end{aligned}$$

in case the execution does not stop. Moreover, we use the notation

$$w; P \triangleq \mathsf{pc} \Mapsto w * P$$

as a shorthand for ownership of the pc register, because the specifications always requires the register points to resource for *pc*.

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \text{decode}(n) = \texttt{jmp}\ r \qquad \text{not_IE_cap}(w)}{\begin{array}{l}\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ \ \mathsf{r} \Mapsto w * a_{pc} \mapsto n \rangle \rightarrow \\ \langle \text{updatePcPerm}(w)\ ;\ \mathsf{r} \Mapsto w * a_{pc} \mapsto n \rangle\end{array}}\ \textsc{Jmp-not-IE}$$

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \text{decode}(n) = \texttt{jmp}\ r \qquad b \leq a < a+1 < e}{\begin{array}{l}\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ \mathsf{idc} \Mapsto _ * \mathsf{r} \Mapsto (\text{IE}, b, e, a) * a_{pc} \mapsto n * a \mapsto w_1 * (a+1) \mapsto w_2 \rangle \rightarrow \\ \langle w_1\ ;\ \qquad \mathsf{idc} \Mapsto w_2 * \mathsf{r} \Mapsto (\text{IE}, b, e, a) * a_{pc} \mapsto n * a \mapsto w_1 * (a+1) \mapsto w_2 \rangle\end{array}}\ \textsc{Jmp-IE}$$

$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \text{decode}(n) = \texttt{jmp}\ r \qquad (a \notin [b, e) \lor (a+1) \notin [b, e))}{\begin{array}{l}\{(p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ \mathsf{r} \Mapsto (\text{IE}, b, e, a) * a_{pc} \mapsto n\} \rightsquigarrow \\ \{s.\ \ \lceil s = \mathsf{Failed} \rceil * \mathsf{r} \Mapsto (\text{IE}, b, e, a) * a_{pc} \mapsto n\}\end{array}}\ \textsc{Jmp-IE-not-in-bounds}$$

Figure 6: Example of weakest-precondition rules.

Figure 6 shows two specifications for the `jmp r` instruction. For both rules, the differences between the post and pre condition is highlighted in blue.

The first rule Jmp-not-IE specifies the jump to a word that is not an indirect sentry capability. It states that, if the pc contains a valid PC capability pointing to the encoding of `jmp r`, and that the register r points to any word w that is not a sentry capability, then after the execution, the pc register contains the updated version of w, *i.e.,* if $w = (\text{E}, b, e, a)$, then $\text{updatePcPerm}(w) = (\text{RX}, b, e, a)$, otherwise $\text{updatePcPerm}(w) = w$.

Similarly, the second rule Jmp-IE specifies the (successful) jump to an indirect sentry capability. The precondition of the rule states that the pc register contains a valid PC capability pointing to the encoding of `jmp r`, the register r points to a valid indirect sentry capability (IE, b, e, a) (valid meaning, the addresses a and $(a+1)$ are in the bounds of the capability), and the addresses a and $(a+1)$ respectively points to the words w_1 and w_2. After the execution of the instruction, the pc now contains the w_1 and the idc register contains w_2.

We highlight that the specification for Jmp-IE requires more resources than Jmp-not-IE. Because of the level of indirection, the specification requires the address points to predicates for a and $(a+1)$, and because the instruction also has the idc register as destination register, the specification also requires the register points to predicate for idc. In practice,

The rule Jmp-IE presented in Figure 6 is actually an instance of a more general rule, which also specifies the cases where some addresses or registers are equals. For the precondition Jmp-IE to hold, it requires the addresses a, $(a+1)$ and a_{pc} to be different from each others. Similarly, it means that the registers pc, idc and r are all different. However, the operational semantics does not prevent those cases, and the more general rule also specify them.

3.1 Adequacy

An *adequacy theorem* is a theorem that links certain specifications in terms of the program logic to specifications purely in terms of the operational semantics. Using an adequacy theorem, we can show that the user-defined memory invariants that hold throughout the complete, safe execution of the machine also hold at the level of the operational semantics. The adequacy theorems of our program logic are the same as the adequacy theorems for the Cerise program logic [7, §4.3], which themselves build on top of the more general Iris adequacy theorem.

4 Logical Relation

4.1 Context: (unityped) logical relations for capability machines

The challenge we are concerned with is that we want to reason about *robust safety*, showing specifications that involve state encapsulation. For example: this program fragment has a location that is 'private' to it, in the sense the location will not be modified by the rest of the program, even if the rest of the program includes unknown, arbitrary, and potentially adversarial code that the code fragment hands control over to (for example by a jump).
Capability monotonicity [4] makes it possible to reason up to a domain change. More concretely, one can recursively compute the transitive authority granted by a word w (made available to the adversary) by instructions other than jumping: if w is a capability, then it has authority over its bounds, plus the authority granted by the words in bounds, recursively. If an address is not covered by this authority (or merely readable, but not writeable), then it can indeed not be modified until a domain change.
If we want to reason about unknown code that includes jumps back into the program fragment that has private state, then we need a more expressive property than capability monotonicity. Figure 7 gives a representation of what our property intends to capture. In this setting, the authority can change non-monotonically during execution, and can thus cover the private state. This is challenging to reason about, because we need to reason about the fact that the program gets authority over this private state, yet the adversary is unable to use this authority to change the private state as it wishes, but merely as the program fragment explicitly allowed it via the code it explicitly exposes.
Concretely, we want our property to allow us to reason about the following type of example. Consider a code fragment which has private location a that wants to preserve the invariant that a always contains a non-negative integer. Consider that the code fragment exposes an entry point that increments the number and returns the new value. As long as only this entry point is exposed, calling the adversary should respect the invariant, even though the adversary can jump to the exposed increment entry point. If we merely computed the authority as a set, then this private location a would be included in that set, and so we could not conclude that its contents is always non-negative. (This is similar to closures in higher-level languages. However, in high-level languages, private state encapsulation is a language-level guarantee. Here, it has to be implemented using capabilities as building blocks, and so does not hold trivially.) Figure 8 gives a representation of such an example.

We define this relation $\mathcal{V}$ in Iris, and rely on Iris' built-in notion of resources. Specifically, we can write that a contains a non-negative integer formally as $R \triangleq \exists n.\, \lceil n \geq 0 \rceil * a \mapsto n$, and make it into an Iris invariant as $\boxed{R}$. Then, to show that jumping to the adversary's code while giving it access to a word w is safe in the sense that it will not break the invariant $\boxed{R}$ (or, put another way, to show that w is safe-to-share), it suffices to show that $\boxed{R}$ is *compatible* with $\mathcal{V}(w)$, as captured by $\boxed{R} * \mathcal{V}(w)$. In particular, because $a \mapsto _$ expresses exclusive ownership of a, and is thus not duplicable, this means that R and $\mathcal{V}(w)$ cannot both contain $a \mapsto _$, and so $\mathcal{V}(w)$ cannot contain it. Defining "w is safe" (our desired $\mathcal{V}(w)$) in a way that makes it obvious that it captures safety is technically challenging, and so we define it somewhat indirectly: we split the definition in two: $\mathcal{V}(w)$, "w is safe-to-share", and $\mathcal{E}(w)$, "w is safe-to-execute". We then prove that any word w that is safe-to-share is also safe-to-execute, which means that w is safe. This is the so-called "fundamental theorem of the logical relation", abbreviated FTLR, which we return to in §4.3.

We extend the Vanilla Cerise's logical relation [7, §5.1] (which gives an extensive explanation of the logical relation), to support indirect sentries.

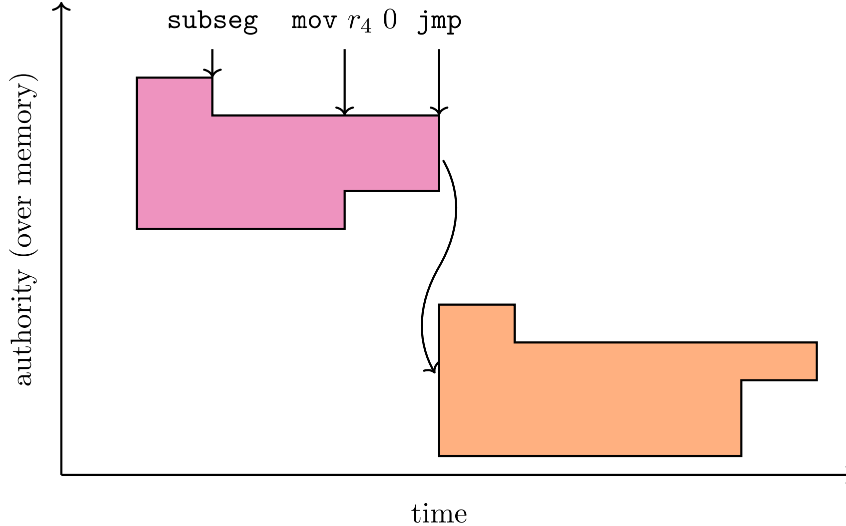


Figure 7: Evolution of authority over time: as per capability monotonicity, it decreases (via `subseg`s and loss of access) until a domain change (induced by a `jmp`)

4.2 Our logical relation for indirect sentries

We extend the Cerise logical relation to deal with capability safety in the presence of indirect sentries. The main new complexity comes from (1) the fact that jumping to an indirect sentry involves not only the `pc` register, but also the `idc`, which needs to be captured by the safe-to-execute predicate, and (2) the design of safe-to-share for an indirect sentry.

Generalised safe-to-execute The safe-to-execute relation needs to capture the more expressive domain transition of the indirect sentries. When the machine jumps to an indirect sentry, it updates both the `pc` and the `idc` registers. The purpose of indirect sentries is to start the execution of the closure with a specific context. As such, the safe-to-execute relation is really about two words. Technically, the safe-to-execute predicate should be parametrised by the *code value* and by the *context value*, which might not be safe-to-share. In order to be able to choose the content of the data register, we generalise and relax the definition of $\mathcal{E}$, such that $\mathcal{E}_{\mathcal{G}}(w_1, w_2)$ captures what it means for w_1 to be safe-to-execute under the context w_2. We then define the expression relation $\mathcal{E}$ in terms of the generalised one

$$\mathcal{E}(w) \triangleq \forall w'.\, \mathcal{V}(w') \twoheadrightarrow \mathcal{E}_{\mathcal{G}}(w, w').$$

Safe-to-share indirect sentries The intuition behind $\mathcal{V}(\textsc{ie}, b, e, a)$ is to capture the fact that *"the state after jumping to an indirect sentry is safe to execute"*. We explain the technical definition using a green labels to refer to specific parts in Figure 9. If the indirect sentry capability is not in bounds (1), it does not grant any authority and is thus useless, and is therefore trivially safe-to-share. Because indirect sentries have a level of indirection, the safe-to-share relation needs to record the points-to predicate of the indirection (3a)-(3b), which capture the ownership of the addresses a and $(a+1)$, and the knowledge of the code word w_1 and the data word w_2. As those resources are meant to be preserved, even when shared with an adversary, they are encapsulated inside an invariant. In addition, because indirect sentries are immutable,

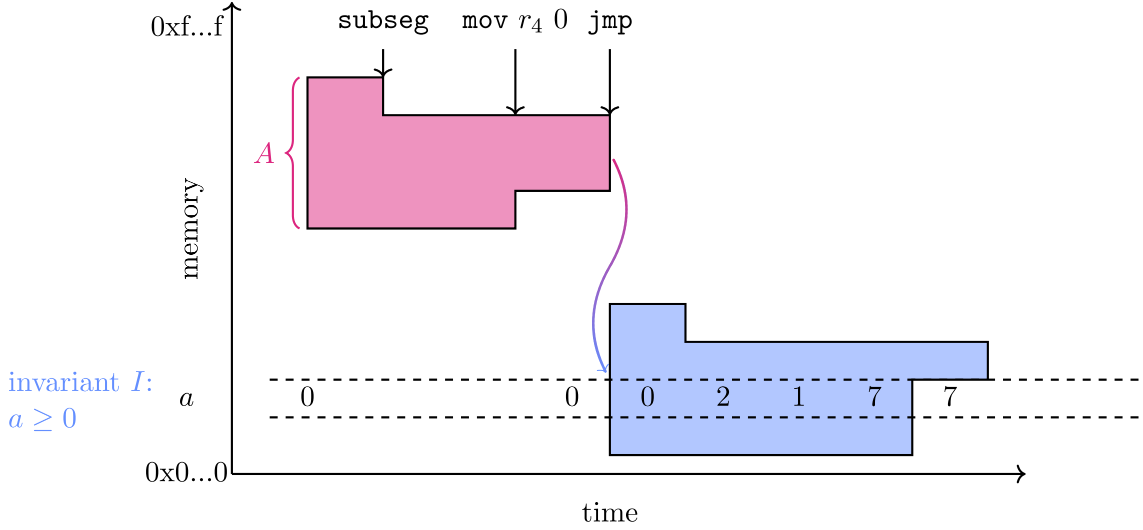


Figure 8: Evolution of authority (coloured regions, drawn as contiguous regions for simplicity; we also draw authority as binary: coloured or not, when it can be more fine-grained: read, write, read-write, execute, ...) over time.
(1) As per capability monotonicity, authority decreases, via `subseg`s and overwriting (line of instructions at the top), until a domain transition (gradient curved arrow) induced by a `jmp`. Therefore, address a, which is outside of the initial authority of the adversary (in purple, top left), does not change before a domain transition.
(2) With capability safety, we can say more: if the initial authority A of the adversary is compatible with the invariant I stating that the word at address a is non-negative, then even when the authority of the known code (in blue, bottom right) covers address a after the domain transition, the word at a remains non-negative.
More technically: to show that it is safe to call the adversary, if we write $w_0, \ldots, w_{31}$ for the roots made available to the adversary in registers $r_0, \ldots, r_{31}$, then the proof obligation is $\boxed{I} \vdash \mathcal{V}(w_0) * \cdots * \mathcal{V}(w_{31})$, which requires that A is compatible with the invariant I of the known code.

we know that some persistent properties hold about the code word and the data word, call them P_1 and P_2 respectively (2). Usually, the exact shape of the code and data words are constrained by P_1 and P_2. A typical instantiation would be $P_1(w_1) \triangleq \lceil w_1 = (p_{code}, b_{code}, e_{code}, a_{code}) \rceil$ and $P_2(w_2) \triangleq \lceil w_2 = (p_{data}, b_{data}, e_{data}, a_{data}) \rceil$. Finally, the *continuation predicate* (4) expresses that it is safe to execute the machine, if an adversary jumps to the indirect sentry. Although it does not correspond to the intended use of indirect sentries, the data word w_2 can be an integer, which is likely to lead to a crash later if the following code expects a capability. The actual reason for the disjunction $P_2(w_2) \vee \mathit{is_int}(w_2)$ is quite technical, and we delay the explanation to the proof of the fundamental theorem in paragraph 4.3. Our study shows that this definition is complete enough to prove interesting use cases, and we discuss how we could lift this disjunction in Section 7.2.

4.3 Fundamental Theorem

The Fundamental Theorem of the Logical Relation 1 (FTLR) states that any safe-to-share (in $\mathcal{V}$) word is also safe-to-execute (in $\mathcal{E}$). Informally, the FTLR means that sharing a safe-to-share word cannot give extra authority or break memory invariant, even by executing it. The FTLR is

$$\mathcal{E}(w) \triangleq \forall w'.\, \mathcal{V}(w') \twoheadrightarrow \mathcal{E}_{\mathcal{G}}(w, w')$$

$$\mathcal{E}_{\mathcal{G}}(w_1, w_2) \triangleq \left\{ w_1 \;;\; \begin{array}{c} \mathsf{idc} \Mapsto w_2 \,* \\ \mathop{\Huge *}_{\{(r,w) \,\mid\, (r,w) \in \mathit{regs}\}} r \Mapsto w * \mathcal{V}(w) \end{array} \right\} \rightsquigarrow \bullet$$

$$\mathcal{V}(w) \quad \begin{cases} \mathcal{V}(z) & \triangleq \mathsf{True} \text{ for } z \in \mathbb{Z} \\ \mathcal{V}(\mathrm{O}, -, -, -) & \triangleq \mathsf{True} \\ \mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\mathrm{RX}, b, e, a) \\ \mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e)} \boxed{\exists w.\, a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e)} \exists P.\, \boxed{\exists w.\, a \mapsto w * P(w)} * \triangleright \Box\, (\forall w.\, P(w) \twoheadrightarrow \mathcal{V}(w)) \\ & \quad * \; \mathit{persistent_cond}\ P \\ \mathcal{V}(\mathrm{IE}, b, e, a) & \triangleq \text{(1)}\ \lceil b \leq a < a+1 < e \rceil \twoheadrightarrow \\ & \quad \left(\begin{array}{l} \text{(2)}\ \exists P_1, P_2.\, \mathit{persistent_cond}\ P_1 * \mathit{persistent_cond}\ P_2 \,* \\ \text{(3a)}\ \boxed{\exists w_1.\, a \mapsto w_1 * P_1(w_1)}^{a} * \\ \text{(3b)}\ \boxed{\exists w_2.\, (a+1) \mapsto w_2 * P_2(w_2)}^{(a+1)} * \\ \text{(4)}\ \forall w_1, w_2.\, \triangleright \Box\, (P_1(w_1) * (P_2(w_2) \vee \mathit{is_int}(w_2)) \twoheadrightarrow \mathcal{E}_{\mathcal{G}}(w_1, w_2)) \end{array} \right) \end{cases}$$

where

$$\begin{array}{lcl} \mathit{regs} & \triangleq & \{(r, w) \mid (r, w) \in \mathit{reg},\ r \neq \mathsf{pc}, r \neq \mathsf{idc}\} \\ \mathit{persistent_cond}\ P & \triangleq & \forall w.\, \mathit{Persistent}\ P(w) \end{array}$$

Figure 9: Logical relation defining “safe to share” ($\mathcal{V}$) and “safe to execute” ($\mathcal{E}$).

sometimes referred as a *universal contract*, in the sense that every instruction has to respect this property (see [9]), because a safe-to-share word can contain arbitrary code. [7, §5.2] explains why the fundamental theorem expresses that the machine "works well" and enables to capture capability safety.

Theorem 1 (Fundamental Theorem) $\forall w.\, \mathcal{V}(w) \twoheadrightarrow \mathcal{E}(w)$

Usually, the FTLR is not directly used as stated in Theorem 1, but rather by its consequence stated in Corollary 1. Corollary 1 states that, it is safe to jump to any safe-to-share word, if all the registers contain safe-to-share values. In other words, to prove that jumping to an unknown word is safe, it suffices to show that the registers contains safe-to-share values.

Corollary 1 (Jump to a safe word[1]) *For any register r_i other than* pc*,*

$$\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \vdash \triangleright \forall regs. \left(\left\{ (p_{pc}, b_{pc}, e_{pc}, a_{pc});\ \begin{matrix} a_{pc} \mapsto \texttt{jmp}\ r_i\ * \\ \left(\mathop{\Large *}\limits_{(r,w) \in regs, r \neq \mathsf{pc}} r \Mapsto w * \mathcal{V}(w) \right) \end{matrix} \right\} \rightsquigarrow \bullet \right)$$

We prove that the FTLR still holds with the extension of Cerise with indirect sentries and our new definition of the logical relation. In the remainder of this section, we sketch the proof of the FTLR, and highlight the new interesting cases related to the indirect sentries. The proof of the FTLR explores every possible case of the operational semantics, and checks that it does not gain authority or break the (user-defined) internal memory invariants of the different closures. More specifically, if the registers contains only safe values, then after executing any instruction, the machine reaches a state that did not break any invariant. Similarly to the rules of the program logic, indirect sentries do not change the proof for most instructions. We focus on the interesting cases, namely the `jmp` and `restrict` instructions.

Proof of the FTLR Let w be a machine word. We show the case $w = (p_{pc}, b_{pc}, e_{pc}, a_{pc})$, as the other cases are trivial.
We proceed by Löb induction: by assuming the FTLR to hold later (after one execution step), we show that the property holds.

Let

$$\begin{aligned} \text{IH} \triangleq\ & \forall p_{pc}, b_{pc}, e_{pc}, a_{pc}.\, \mathcal{V}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \twoheadrightarrow \\ & \forall reg. \left\{ (p_{pc}, b_{pc}, e_{pc}, a_{pc}); \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet \end{aligned}$$

and assume $\triangleright$IH. Let further assume $\mathcal{V}(w)$. We want to show $\mathcal{E}(w)$. By unfolding the definition of safe to execute, we assume a word w_{idc} such that $\mathcal{V}(w_{idc})$, and we need to show that, for any register maps *regs* such that $\mathsf{pc}, \mathsf{idc} \notin regs$,

$$\left\{ (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ \begin{matrix} \mathsf{idc} \Mapsto w_{idc}\ * \\ \mathop{\Large *}\limits_{\{(r,w)\ \mid\ (r,w) \in regs\}} r \Mapsto w * \mathcal{V}(w) \end{matrix} \right\} \rightsquigarrow \bullet$$

For the purpose of the presentation, we further assume the content of pc is a valid PC capability, *i.e.,* $\mathsf{pc} \Mapsto (p_{pc}, b_{pc}, e_{pc}, a_{pc})$ with $p_{pc} \in \{\text{RX}, \text{RWX}\}$ and $b_{pc} \leq a_{pc} < e_{pc}$.

Because we know that this capability is safe to share, from the assumption of the FTLR, we get $\boxed{\exists w_{pc}.\, a_{pc} \mapsto w_{pc} * \mathcal{V}(w_{pc})}^{a_{pc}}$. By opening the invariant a_{pc}, we get a word w_{pc}, the

[1]The corollary is slightly different from the original one in Cerise. Further discussion in Section 7.2.

points-to resource, and that w_{pc} is safe to share. We recall that we ought to close the invariant after one execution step.

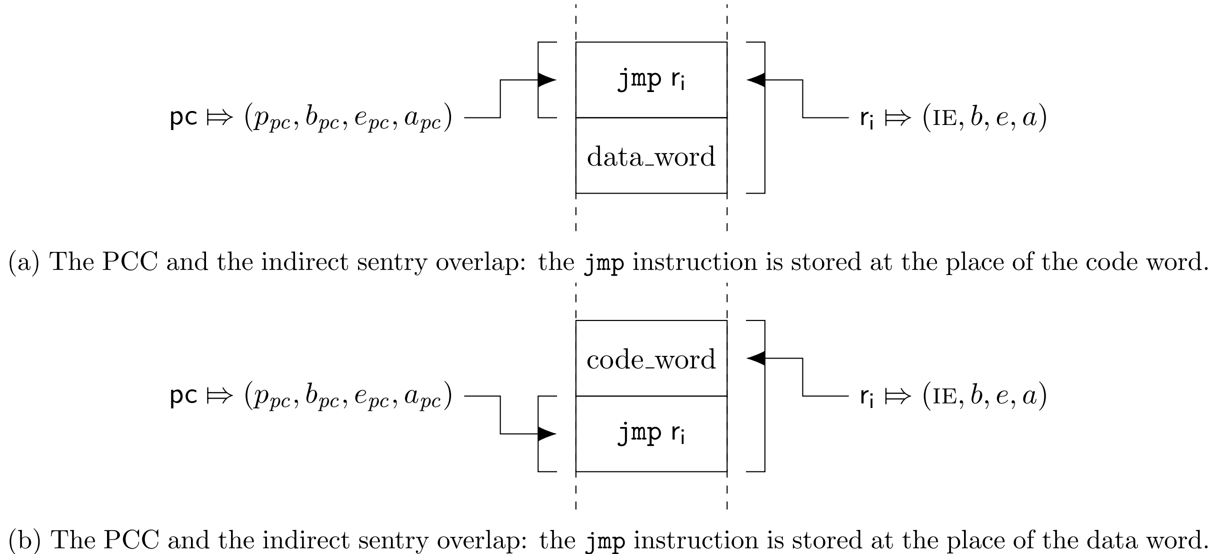
We assume that w_{pc} is an integer — if it were a capability, it would be trivially safe — and we proceed by case analysis over $decode(w_{pc})$.

Case Jmp to indirect sentry We take a look at the case $decode(w_{pc}) = \texttt{jmp}\ r$, when r is not pc. From the assumption over the register maps *regs*, we know that it exists a word w_r such that $r \Mapsto w_r$ and $\mathcal{V}(w_r)$. We proceed by case analysis over w_r, and focus on the case $w_r = (\text{IE}, b, e, a)$ with $b \leq a < (a{+}1) < e$, *i.e.,* the case where w_r is an in-bounds indirect sentry. By definition of $\mathcal{V}(\text{IE}, b, e, a)$ and because it is in-bounds, we get $\boxed{\exists w_1.\, a \mapsto w_1 * \ P_1(w_1)}^{a}$ and $\boxed{\exists w_2.\, (a+1) \mapsto w_2 * \ P_2(w_2)}^{(a+1)}$ for some persistent predicates P_1 and P_2, and the continuation predicate.

The next step of the proof should be to apply the rule Jmp-IE, but the points-to resources necessary to apply the rule are in the invariants named a and $a+1$. There are corner cases that need to be considered in order to open the invariants: a_{pc} can be the address of the code word of the indirect sentry being jumped to, or the address of the data word. In other words, we need to consider the cases where the PCC overlaps with the indirect sentry. From a strictly technical point of view, because the invariant named a_{pc} is already opened, and because it is not possible to open two invariant with the same name twice, we need to consider the case with $a = a_{pc}$ and the case with $a + 1 = a_{pc}$. Each case follows a similar approach:

1. Open the invariants a and $a + 1$ that are not opened yet (*i.e.,* that are not a_{pc}).
2. Use the rule Jmp-IE to update the resources of the machine.
3. Close all the invariants previously opened.
4. Terminate the proof by showing that the machine executes safely and completely.

We consider the expected case first, and then the two corner cases (where a or $a + 1$ clash with a_{pc}).



(a) The PCC and the indirect sentry overlap: the jmp instruction is stored at the place of the code word.

(b) The PCC and the indirect sentry overlap: the jmp instruction is stored at the place of the data word.

Figure 10: The corner cases of the jmp instruction

- In the case with no overlap, addresses a and $(a + 1)$ contain words w_1 and w_2 such that $P_1(w_1)$ and $P_2(w_2)$ hold for some P_1 and P_2. After the jump, pc and idc respectively

contain w_1 and w_2. Because we persistently know $P_1(w_1)$ and $P_2(w_2)$, the continuation predicate states $\mathcal{E}_{\mathcal{G}}(w_1, w_2)$, which means that the machine executes completely and safely from this machine state.

- In the case where a_{pc} is the address a, then this address contains the encoding of a `jmp` instruction, which is an integer z. Moreover, $(a+1)$ contains a word w_2 such that $P_2(w_2)$ holds, for some P_2. Figure 10a illustrates this case. After the jump, pc contains the aforementioned integer z and idc contains w_2. As the pc register contains an integer, the machine safely and completely execute trivially from this machine state, because the next cycle will fail.
- In the case where a_{pc} is the address $(a+1)$, then this address contains the encoding of a `jmp` instruction, which is an integer. Moreover, a contains a word w_1 such that $P_1(w_1)$ holds, for a certain P_1. Figure 10b illustrates this case. After the jump, pc contains w_1, and idc contains the aforementioned integer. Because we persistently know $P_1(w_1)$ and $is_int(w_2)$, the continuation predicate states $\mathcal{E}_{\mathcal{G}}(w_1, w_2)$. It means that the machine executes completely and safely from this machine state.

The disjunction for the data word w_2 in the continuation, $P_2(w_2) \vee is_int(w_2)$, enables the last corner case 10b to continue after the jump. We discuss in Section 7.2 possible solutions to make the last corner case to work without the disjunction with $is_int(w_2)$.

Case `restrict` The case of the `restrict` instruction relies on the safe-to-share-monotonicity lemma 1 below, which roughly states that if a capability (p, b, e, a) is safe-to-share, then restricting its permission p to a less permissive $p' \preccurlyeq p$ preserves the safe-to-share property.

Lemma 1 (Safe-to-share monotonicity) *Let p, p' be capability permissions such that $p \neq \text{E}$, $p \neq \text{IE}$, and $p' \preccurlyeq p$. Let b, e, and a be addresses, and let w_{idc} be a safe-to-share word $\mathcal{V}(w_{\mathsf{idc}})$. Further assume $\triangleright IH$,* i.e., *that the FTLR holds later (in the sense of $\triangleright$). If $\mathcal{V}(p, b, e, a)$, then $\mathcal{V}(p', b, e, a)$.*

We sketch the proof for the case $p = \text{RO}$ and $p' = \text{IE}$. We assume $\mathcal{V}(\text{RO}, b, e, a)$ and need to show $\mathcal{V}(\text{IE}, b, e, a)$. By definition of safe-to-share for an indirect sentry, we assume that $b \leq a < (a+1) < e$, and we need to show that there exists two persistent propositions P_1 and P_2, such that $\boxed{\exists w_1.\, a \mapsto w_1 * P_1(w_1)}$ and $\boxed{\exists w_2.\, (a+1) \mapsto w_2 * P_2(w_2)}$, and the continuation

$$\forall w_1, w_2. \triangleright \square \left(P_1(w_1) * (P_2(w_2) \vee is_int(w_2)) \mathrel{-\!\!*} \mathcal{E}_{\mathcal{G}}(w_1, w_2)\right)$$

By definition of $\mathcal{V}(\text{RO}, b, e, a)$, we get P_1 and P_2, as well as the invariants. It only remains to show the continuation. Let w_1 and w_2 be machine words such that $P_1(w_1)$ and $(P_2(w_2) \vee is_int(w_2))$. The continuation is under a persistency modality $\square$, which means that it can only rely on persistent resources. The continuation is also under a later modality $\triangleright$, which mean that we can strip the later from the induction hypothesis IH. Moreover, we also know $\mathcal{V}(w_1)$ and $\mathcal{V}(w_2)$, because, $is_int(w_2) \mathrel{-\!\!*} \mathcal{V}(w_2)$ and, $\mathcal{V}(\text{RO}, -, -, -)$ ensures that $P_1(w_1) \mathrel{-\!\!*} \mathcal{V}(w_1)$ and $P_2(w_2) \mathrel{-\!\!*} \mathcal{V}(w_2)$.

Because we have $\mathcal{V}(w_1)$ and $\mathcal{V}(w_2)$, the induction hypothesis IH gives us $\mathcal{E}_{\mathcal{G}}(w_1, w_2)$, which is what we wanted to prove.

5 Case Studies

In this section, we illustrate that indirect sentries can be used to create closures that can be shared with unknown code. We illustrate that the logical relation enforces local state encapsulation. We present two cases: a module that increments a private counter at each call; and a secure heap-based calling convention that do not require any trampoline code.

5.1 Counter closure

We illustrate the use of indirect sentries to implement an encapsulated closure with a private counter, with a single entry point to code that increments the value of the counter. This is a variation of the counter module of Cerise [7, §6.2], where we use the indirect sentry to separate the code and the data (whereas their implementation mixed them, which we argued is undesirable, see 1). With this example, we illustrate how the indirect sentry can be used to create a closure that encapsulates some private state, yet can be safely shared with unknown, adversarial code.

Program Figure 11 shows the code of the counter closure. The initial memory of the machine is split in two parts: the code part in the region [`init`, `end`) and the data part in the region [`data`, `data_end`). At the initial state of the machine, the program initializes (*lines 4 – 23*) the closure by storing the code capability at the address `data`, storing the data capability at the address `data`+1, and creates the indirect sentry before jumping to an adversary.

At the invocation of the indirect sentry, the program (*lines 29–34*) fetches the value of the counter, increments its value by one, and jumps back to the caller. The initial value of the counter is 0.

In order to show that the state of the closure is indeed private, it suffices to show that the machine can completely and safely execute, under the invariant that the value of the counter is always non-negative. The fact that the counter always remains non-negative witnesses the fact that an adversary cannot change the value of the counter arbitrarily.

Specification The crucial part of the specification is to prove that the indirect sentry of the counter closure is safe-to-share under the invariant that the counter value is always non-negative (1). This invariant captures Local State Encapsulation, because an adversary with the write-access to the counter could change its value arbitrarily (see [7, §6.2]).

$$\boxed{\exists n.\, \mathsf{counter} \mapsto n * \lceil 0 \leq n \rceil}\,(1) * \boxed{[\mathsf{code}, \mathsf{end}) \mapsto \mathtt{counter_instrs}}\,(2)$$
$$* \boxed{\mathsf{data} \mapsto (\textsc{rx}, \mathsf{init}, \mathsf{end}, \mathsf{code})}\,(3) * \boxed{\mathsf{data}+1 \mapsto (\textsc{rw}, \mathsf{data}, \mathsf{data_end}, \mathsf{counter})}\,(4)$$
$$\vdash \mathcal{V}(\textsc{ie}, \mathsf{data}, \mathsf{data_end}, \mathsf{data})$$

The other invariants ((2), (3), (4)) are allocated along the proof of the specification of the full program: the region of memory $[\mathsf{code}, \mathsf{end})$ contains the instructions of the increment program (2), the address data contains the code capability of the indirect sentry (3), and the address $\mathsf{data}+1$ contains the data capability of the indirect sentry (4).

Proof By definition of $\mathcal{V}(\textsc{ie}, \mathsf{data}, \mathsf{data_end}, \mathsf{data})$, it suffices to prove that there exists two persistent predicates P_1 and P_2 such that

1. $\boxed{\exists w_1.\, \mathsf{data} \mapsto w_1 * P_1(w_1)}$
2. $\boxed{\exists w_2.\, \mathsf{data}+1 \mapsto w_2 * P_2(w_2)}$

```
1  ; initially, PC = (RX, init, end, init)
2  ;             IDC = (RW, data, data_end, data)
3  ;             r31 = (unknown) capability to the context
4  init:
5    ; 1. store the code capability
6    mov r1 PC                ; r1 = (RX, init, end, init)
7    lea r1 [code-init]       ; r1 = (RX, init, end, code)
8    store IDC r1             ; mem[data] <- (RX, init, end, code)
9
10   ; 2. store the data capability
11   mov r1 IDC               ; r1 = (RW, data, data_end, data)
12   lea r1 [data-counter]     ; r1 = (RW, data, data_end, counter)
13   lea IDC 1                ; IDC = (RW, data, data_end, data+1)
14   store IDC r1             ; mem[data+1] <- (RW, data, data_end, counter)
15
16   ; 3. prepare the IE
17   lea IDC (-1)             ; IDC = (RW, data, data_end, data)
18   restrict IDC IE          ; IDC = (IE, data, data_end, data)
19
20   ; 4. jump to unknown code
21   mov r1 0                 ; r1 = 0
22   jmp r31                  ; jump to unknown code: we only give it access
23                            ; to an indirect sentry capability pointing to 'code'
24
25 ; when 'code' gets executed from the IE capability,
26 ;   PC = (RX, init, end, code)
27 ;   IDC = (RW, data, data_end, counter)
28 ;   r31 = (unknown) return capability to the continuation
29 code:
30   load r1 IDC              ; r1 = <counter value>
31   add r1 r1 1              ; r1 = <counter value> + 1
32   store IDC r1             ; mem[counter] <- <counter value> + 1
33   mov IDC 0                ; IDC = 0
34   jmp r31                  ; return to unknown code
35 end:
36
37 data:
38   0xFFFF, ; will be overwritten with (RX, init, end, code), i.e.
39           ; a read-execute capability to the code
40   0xFFFF, ; will be overwritten with (RW, data, data_end, counter), i.e.
41           ; a read-write capability to the counter value
42 counter:
43   0       ; our private data: the current value of the counter
44 data_end:
```

Figure 11: Program implementing a secure counter with indirect sentries

3. $\forall w_1, w_2. \triangleright \square (P_1(w_1) * (P_2(w_2) \vee is_int(w_2)) \twoheadrightarrow \mathcal{E}_{\mathcal{G}}(w_1, w_2))$

We choose $P_1(w) \triangleq \lceil w = (\textsc{rx}, \mathsf{init}, \mathsf{end}, \mathsf{code}) \rceil$ and $P_2(w) \triangleq \lceil w = (\textsc{rw}, \mathsf{data}, \mathsf{data_end}, \mathsf{counter}) \rceil$. Using (3) and (4), we can prove the two first invariants, and it remains to prove the continuation. To do so, we show that the increment program is safe to share, (1) when the data word w_2 is the expected capability $(\textsc{rw}, \mathsf{data}, \mathsf{data_end}, \mathsf{counter})$, and (2) when the data word w_2 is an integer. Because the increment program assumes that the data word is a capability, the case (2) where the data word is an integer trivially fails at the first `load` instruction, and is thus safe to execute. For the case (1), where the data word contains the expected capability $(\textsc{rw}, \mathsf{data}, \mathsf{data_end}, \mathsf{counter})$, it suffices to show that the program does not break the invariant $\boxed{\exists n,\ \mathsf{counter} \mapsto n * \lceil 0 \leq n \rceil}$,

which holds because the program only increments the counter value.

Conclusion This counter closure example illustrates how to use indirect sentry to create an encapsutaled closure (*i.e.,* with a private state), and to safely share an entry point to this closure. In the next example, we show how to use the indirect sentries to create a secure calling convention.

5.2 Heap-based calling convention

We present a secure calling convention using indirect sentries that enforces Local State Encapsulation, *i.e.,* a callee cannot access the local variables of the closure. This calling convention is inspired by function calls in high-level language, such as the calling convention used to extend Standard ML with call/cc [2]. Our calling convention is an adaptation of Vanilla Cerise's calling convention [7, §7.3]. We show that this calling convention guarantees the encapsulation of local state, even when the callee (or caller) is unknown, untrusted code. In particular, this guarantee does not rely on the callee using the same calling convention. Whereas Cerise's calling convention creates a return pointer with an entry capability intertwining code and data, our calling convention uses an indirect sentry. As a consequence, our calling convention does not need "trampoline code" to restore the local state.

Figure 12 shows the code of the call routine. We assume that we have a trusted `malloc` macro. The macro is *trusted*, in the sense that we have a specification, that proves the macro to only allocate an available region of memory. Its code and specification are described in Cerise [7, §7.1]. Before jumping to the target, the `call` sub-routine stores the local state and prepares the return pointer as follows:

1. Dynamically allocate some heap memory region $[l, l_{end})$ with `malloc`, and store the local state. *(lines 9 – 11)*
2. Dynamically allocate some heap memory region $[b_{ie}, e_{ie})$ with `malloc`, and store the code and the data capabilities. The code capability is a capability that points to the instruction following the call. The data capability is the allocated capability pointing to the local state $(\text{RWX}, l, l_{end}, l)$. *(lines 12 – 13)*
3. Create an indirect sentry pointing to the code and data capabilities $(\text{IE}, b_{ie}, e_{ie}, b_{ie})$. This indirect sentry is the return pointer to the caller, passed to the callee. *(lines 15 – 21)*
4. Clear all registers except those in *params*. *(line 22)*
5. Jump to *target*.*(line 23)*

After the jump, the callee has access to the parameters and the return pointer, but does not have access to the local state. When the callee jumps to the return pointer, the jump restores the code capability in the `pc` register, and the data capability pointing to the local state in the `idc` register. Finally, after the jump, the caller can restore the local state.

6. Restore the local state (`RESTORE_LOCALS` `idc` *locals*).

The specification relies on the specification of the malloc. The mallocInv invariant roughly describes that $[b_m, e_m)$ contains the code of the malloc routine, and the memory pool from which the new memory is allocated, making sure it only allocates free memory. The precondition of the call describes the resources required before the execution of the routine. The `pc` points to

```
1  ; initially, PC = (RWX, code, end, a)
2  ;             target = register containing the address to jump to
3  ;             locals, params = lists of register names
4  code:
5    ...
6  a:
7    MALLOC (len locals)          ; 1. macro: allocate and store local state
8    STORE_LOCALS r1 locals
9    mov r6 r1
10   MALLOC 2                     ; 2. macro: allocate region for code/data of indirect sentry
11   mov r31 r1
12 x:
13   mov r1 pc                    ;    prepare and store the continuation
14   lea r1 [cont - x]
15   store r31 r1
16   lea r31 1                    ;    store the capability to locals
17   store r31 r6
18   lea r31 -1                   ; 3. create the return capability
19   restrict r31 IE
20   RCLEAR RegName\({PC,r31,target} ∪ params) ; 4. clear all registers except parameters
21   jmp target                   ; 5. jump to target
22 cont:
23   RESTORE_LOCALS idc locals    ; 6. reinstate local state
24   ...
25 data:
26   (RO, table, end, table)      ; environment table
27 table: ; linking table
28   (E, bm, em, bm)              ; entry point to the malloc subroutine
29   ...                          ; possibly other routines
30 end:
```

Figure 12: Our heap-based calling convention using indirect sentries.
We use Coq as an assembler, so macros (in all caps) take arguments which are Coq terms. The macro parameters *locals* and *params* are instantiated with lists of registers. The *linking table* contains a list of trusted routines, including the one called by the MALLOC macro.

the start of the instructions of the call. Via the PC capability, the program has access to a linking table (RO, table, end, table) which includes the entry capability to the malloc routine.

The postcondition of the specification stops at the last instruction, before the jump (step 5, line 23). After the call, we know that there exists some dynamically allocated addresses b_{ie} and e_{ie} for the indirection of the indirect sentry, and l and l_{end} for storing the locals. The locals *lws* are indeed stored in the memory in $[l, l_{end})$. The register r_{31} contains the return capability, as an indirect sentry. Every register, except the parameters *params*, the *target*, and r_{31}, has been

cleared.

$$\boxed{\text{mallocInv}(b_m, e_m)} \vdash \left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{a}); \begin{array}{l} [\mathsf{a}, \mathsf{cont}) \mapsto \mathtt{call_instrs} * \\ \mathsf{data} \mapsto (\text{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) * \mathsf{table} \mapsto (\text{E}, b_m, e_m, b_m) * \\ \mathit{params} \Mapsto \mathit{pws} * \mathit{locals} \Mapsto \mathit{lws} * \mathit{target} \Mapsto w_{adv} * \\ \bigast_{\substack{(r,v) \in reg \\ r \notin \{\mathsf{pc}, \mathit{target}\} \\ r \notin \mathit{params} \cup \mathit{locals}}} r \Mapsto v \end{array} \right\} \rightsquigarrow$$

$$\left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{cont} - 1); \begin{array}{l} \exists b_{ie}, e_{ie}, l, l_{end}, reg'. \\ \mathsf{r_{31}} \Mapsto (\text{IE}, b_{ie}, e_{ie}, b_{ie}) * \\ \mathsf{data} \mapsto (\text{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) * \mathsf{table} \mapsto (\text{E}, b_m, e_m, b_m) * \\ \mathit{params} \Mapsto \mathit{pws} * \mathit{target} \Mapsto w_{adv} * [l, l_{end}) \mapsto \mathit{lws} * \\ [b_{ie}, e_{ie}) \mapsto [(p, \mathsf{code}, \mathsf{end}, \mathsf{cont}); (\text{RWX}, l, l_{end}, l_{end})] * \\ \bigast_{\substack{(r,v) \in reg' \\ r \notin \{\mathsf{pc}, \mathit{target}, \mathsf{r_{31}}\} \\ r \notin \mathit{params}}} r \Mapsto v \end{array} \right\}$$

Because w_{adv} could be anything, the specification after the jump is left to the user. Usually, w_{adv} is safe-to-share, and it suffices to use the Corollary 1 to continue. As such, it requires little additional proof effort in addition to our calling convention.

We highlight that our calling convention does not need a trampoline code that intertwine code and data, as explained in §1. Because of the return pointer is an indirect sentry, the machine restores both the code and the local state during the jump back to the caller. In the case of a heap-based calling convention, the gain is negligible. However, for stack-based calling convention, the trampoline code has to be stored in the stack, which forces the stack to be executable. Using a calling convention with indirect sentries allows to remove this trampoline code. We hope that secure stack-based calling conventions ensuring Well-Bracketed Control Flow [14, 15, 8, 6] can also be extended with indirect sentries, avoiding the trampoline code in the stack. We leave this extension as future work (see §7).

5.3 Sharing a sub-buffer

We show that our calling convention safely encapsulates local state in a concrete example. Moreover, we show that the logical relation is complete enough to reason about the interaction with an adversary. The program stores a *private* value in a buffer, shares the remaining, public part of the buffer with an adversary using the calling convention, and checks whether the private value has been modified after the call. We are only interested in the integrity of the private value, not in the confidentiality. More precisely, the full program in Figure 13 does the following:

1. Dynamically allocate a buffer $(\text{RWX}, b_{mem}, e_{mem}, b_{mem})$ of size N using `malloc` *(lines 1 to 5)*
2. Store a private data in the buffer, at an offset $0 \leq \mathit{offset} < N$ *(lines 7 – 8)*
3. Derive two capabilities, with $a_{\mathit{off}} \triangleq b_{mem} + \mathit{offset} + 1$:
   - a capability pointing to the public part of the buffer, $C_{\text{pub}} = (\text{RWX}, a_{\mathit{off}}, e_{mem}, a_{\mathit{off}})$ *(lines 10 – 13)*
   - a capability pointing to the private part of the buffer, $C_{\text{priv}} = (\text{RWX}, b_{mem}, a_{\mathit{off}}, b_{mem})$ *(lines 16 – 19)*

```
1  code:
2    MALLOC size                   ; 1. allocate the buffer
3    mov r7 r1
4    mov r1 0
5    mov r7 r8
6
7    lea r7 secret_off
8    store r7 secret_val           ; 2. store the secret data
9
10   getb r2 r7
11   gete r3 r7
12   add r2 (secret_off + 1)
13   subseg r7 r2 r3               ; 3.1. derive public capability C_p
14
15   ; secret
16   getb r2 r8
17   getb r3 r8
18   add r3 (secret_off + 1)
19   subseg r8 r2 r3               ; 3.2. derive secret capability C_s
20
21   CALL r30 [r8] [r7]            ; 4. call the adversary
22   RESTORE_LOCALS idc [r8]       ; 5.1. restore the local state
23
24   lea r8 secret_off
25   load r4 r8
26   mov r5 secret_val
27   ASSERT r4 r5                  ; 5.2. assert unchanged secret value
28
29   ; halts
30   halt
31 a:
```

Figure 13: Program sharing a sub-buffer using the calling convention

4. Call the adversary using the calling convention. The call encapsulates the private buffer C_{priv}, and pass the public buffer C_{pub} as a parameter to the adversary. *(line 21)*
5. After the call, restore the local state, i.e. the private buffer, is restored, and dynamically assert the integrity of the private value. *(lines 22 – 27)*

Because the buffer is dynamically allocated, the address of the private data is not statically known, and so we cannot directly state that it does not change. We solve this with a little bit of indirection, following the same methodology as Cerise, by using an `assert` macro. We show that the assert never fails throughout the whole execution, indicating that the adversary cannot modify the private value, and that the calling convention enforces LSE. Internally, the dynamic `assert` macro changes a flag if the assertion fails. This flag is located at a statically known address. By using an invariant stating that the value of the flag never changes, we make sure that the assertion does not fail.

A variant of this example was already mechanised [12], which itself is a variant of the subbuffer case study in Cerise [7, §6.1] with the use of the dynamic allocation [7, §7.1], the dynamic assertion [7, §7.2] and the secure calling convention [7, §7.3] with a trampoline code. With our example, we aim to show that the same security properties hold using our calling convention, and that we can specify similar programs. In the rest of the section, we highlight

the key parts where our proof differs from that of [12].

The specification of the program states that, if the pc points at the first instruction of the program, and if the linking table contains the entry points for malloc and assert, the invariants bellow for the malloc and the assert (which talks about the assertion flag) will hold throughout the entire execution:

$$\boxed{\text{mallocInv}(b_m, e_m)} * \boxed{\text{assertInv}(b_a, e_a)}$$
$$\vdash \left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{code}); \begin{array}{l} [\mathsf{code}, \mathsf{data}) \mapsto \texttt{subbuffer_instrs} * \\ \mathsf{data} \mapsto (\text{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) * \\ [\mathsf{table}, \mathsf{end}) \mapsto [(\text{E}, b_m, e_m, b_m); (\text{E}, b_a, e_a, b_a)] * \\ \mathop{\ast}_{(r,w) \in reg, r \neq \mathsf{pc}} r \Mapsto w * \mathcal{V}(w) \end{array} \right\} \rightsquigarrow \bullet$$

The specification of the example can be split in 3 main parts: (1) the specification of the code *before* the `call` macro (*lines 1–21*), (2) the specification of the `call` macro, before the final jump (implicit in *line 21*), and (3) the specification following the jump (*lines 21–31*). The first part (1) is only known code, and requires basic usage of the program logic. The second part (2) uses the specification of the `call` macro, which stops at the pc pointing to the jump instruction. The third part (3) is the key part of the proof, as it requires to use the FTLR to reason with the unknown code.

In the remainder of this section, we focus on the second and third part of the proof. The following is a specification starting after the first part. We use the indicator (X) to describe the resources in the text. The blue color in the postcondition highlights the changes between the precondition and the postcondition.

After the execution of the first part of the program, the pc register (1) points at the beginning of the `call` macro (*line 21* in Figure 13). The dynamically allocated buffer $[b_{mem}, e_{mem})$ can be split in two parts: the private buffer $[b_{mem}, a_{off})$ (4), with $a_{off} - 1$ containing the private $private_{val}$, and the public buffer $[a_{off}, e_{mem})$ (5), which contains only zeroes. The register r_7 (6) contains a capability pointing to the public part of the buffer. The register r_8 (7) contains a capability pointing to the private part of the buffer. The register r_{30} (8) contains the adversary pointer, that is safe to share (9).

$$
\boxed{\text{mallocInv}(b_m, e_m)} * \boxed{\text{assertInv}(b_a, e_a)}
$$

$$
\vdash \left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{a})(1); \begin{array}{l} \exists b_{mem}, e_{mem}, a_{off}. \\ [\mathsf{a}, \mathsf{cont}) \mapsto \mathtt{call_instrs}(2) * \cdots (3) * \\ [b_{mem}, a_{off}) \mapsto [0; \cdots; private_{val}](4) * \\ [a_{off}, e_{mem}) \mapsto [0; \cdots; 0](5) * \\ \mathsf{r}_7 \Mapsto (\text{RWX}, a_{off}, e_{mem}, a_{off})(6) * \\ \mathsf{r}_8 \Mapsto (\text{RWX}, b_{mem}, a_{off}, b_{mem})(7) * \\ \mathsf{r}_{30} \Mapsto w_{adv}(8) * \mathcal{V}(w_{adv})(9) \\ * \cdots (10) \end{array} \right\} \rightsquigarrow
$$

$$
\left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{cont} - 1)(11); \begin{array}{l} \exists b_{mem}, e_{mem}, a_{off}, b_{ie}, e_{ie}, l, l_{end}, reg'. \\ [\mathsf{a}, \mathsf{cont}) \mapsto \mathtt{call_instrs}(2) * \cdots (3) * \\ [b_{mem}, a_{off}) \mapsto [0; \cdots; private_{val}](4) * \\ [a_{off}, e_{mem}) \mapsto [0; \cdots; 0](5) * \\ [l, l_{end}) \mapsto [(\text{RWX}, b_{mem}, a_{off}, b_{mem})](12) * \\ [b_{ie}, e_{ie}) \mapsto [(p, \mathsf{code}, \mathsf{end}, \mathsf{cont}); (\text{RWX}, l, l_{end}, l_{end})](13) * \\ \mathsf{r}_7 \Mapsto (\text{RWX}, a_{off}, e_{mem}, a_{off})(6) * \\ \mathsf{r}_8 \Mapsto 0(14) * \\ \mathsf{r}_{30} \Mapsto w_{adv}(8) * \mathcal{V}(w_{adv})(9) * \\ \mathsf{r}_{31} \Mapsto (\text{IE}, b_{ie}, e_{ie}, b_{ie})(15) * \\ \cdots (10) \end{array} \right\}
$$

The postcondition of the call macro stops when the pc register (11) points at the final jump of the macros. For the sake of readability, the ellipsis in the pre and postcondition hide the non relevant resources. They corresponds to the memory points-to predicates of the first and third parts of the code (3), the memory points-to predicates of the linking table (3), and all the other register points-to predicates containing safe-to-share words (10). The reader can find the full specification in Appendix B

To end the proof of the full specification, we can use the Corollary 1, as the next executed instruction is a `jmp` r. To do so, we need to show that the content of every register is safe-to-share. In particular, we need to show that the content of r_7 and r_{31} is safe to share.

For r_7 (6), proving that the public capability $C_{\text{pub}} \triangleq (\text{RWX}, b_{mem}, a_{off}, b_{mem})$ is safe to share reduces to showing that its content is safe to share. Here, the buffer $[a_{off}, e_{mem})$ (5) contains only zeroes, which are trivially safe to share.

For r_{31} (15), we need to show that $(\text{IE}, b_{ie}, e_{ie}, b_{ie})$ is safe to share. By definition of $\mathcal{V}(\text{IE}, -, -, -)$, and because $b_{ie} \leq b_{ie} < b_{ie} + 1 < e_{ie}$, we get to choose the predicates P_1 and P_2 such that $l \mapsto w_1 * P_1(w_1)$ and $(l+1) \mapsto w_2 * P_2(w_2)$. We choose $P_1 \triangleq \lambda w. \lceil w = (p, \mathsf{code}, \mathsf{end}, \mathsf{cont}) \rceil$ and $P_2 \triangleq \lambda w. \lceil w = (\text{RWX}, l, l_{end}, l_{end}) \rceil$. Then, we allocate the invariant for the addresses b_{ie} and $(b_{ie}+1)$, and it remains to show the *continuation predicate*.

To apply the continuation, we first need to introduce the $\Box$-modality, which means that the continuation can only rely on persistent properties. Intuitively, the reason of the $\Box$-modality exhibits the fact that, the indirect sentry can be called at any time, in any content, and then its specification after the jump can only rely on predicate that always holds (*i.e.,* that are persistent). As such, we allocate an invariant containing the last part of the code. Then, the continuation starts with $\mathsf{pc} \Mapsto (p, \mathsf{code}, \mathsf{end}, \mathsf{cont})$, *i.e.,* pointing at the first instruction of `restore_local`. The register $\mathsf{idc} \Mapsto w_2$ for a certain w_2 such that, either $w_2 = (\text{RWX}, l, l_{end}, l_{end})$ or $w_2 \in \mathbb{Z}$. All the other registers contain safe to share values. The case $w_2 = (\text{RWX}, l, l_{end}, l_{end})$ means that the local state was correctly restored in the idc register. It is what the `restore_local` expects, and we the skip the rest of the specification. The case $w_2 \in \mathbb{Z}$ is an artefact of the

logical relation. After a few instructions, the macro `restore_local` attempts to execute a `Load` r `idc`, which fails because `idc` contains an integer. Because failing is a safe behaviour, this concludes this case of the proof.

6 Related Work

6.1 Capability monotonicity

Capability monotonicity has been proven for CHERI [11], stating that the authority of the machine can only decrease, until it reaches a controlled domain transition. Morello has indirect sentries, and Bauereiss et al. [4] take them into account in their monotonicity theorem as follows:

> "This guarantees that software cannot escalate its privileges by forging capabilities that are not reachable from the starting state. Non-monotonic changes in the set of reachable capabilities are limited to the specific mechanisms defined above for transferring control to another security domain, i.e. ISA exceptions or sealed capability invocations, installing capabilities belonging to the new domain in the PCC (and possibly IDC) register. The monotonicity guarantee stops before such a domain transition happens. Sealed capability invocations within a security domain are monotonic, however; the theorem does cover capability invocation instructions, e.g. branch instructions taking sentry capabilities, if the unsealed invoked capability is reachable in the current security domain."

Whereas capability monotonicity stops at domain transition, our logical relation 4 intends to capture capability safety, and allows us to reason even after such a domain transition. However, Cerise is a simplified CHERI-like machine. We do not have to deal with the concrete ISA, which makes the proofs more tractable.

6.2 Points-to-PCC indirect sentry capabilities

The CHERI ISA-V9 [17] proposes two different implementations for the indirect sentries: *points-to-pair* and *points-to-PCC*. In this work, we explored formally how to reason about programs with the *points-to-pair* implementation.

A *points-to-PCC* indirect sentry points to the code capability, and ranges over the data. Upon invocation, the machine unseals the indirect sentry with the permission RW, installs the code capability in the `pc` register, and moves the current address of the unsealed indirect sentry to the next address. These three actions are done during one instruction. Figure 14 shows a representation of an indirect sentry with the *points-to-PCC* flavour.

It would be interesting to explore formally if the security properties would also hold for this flavor of indirect sentries. In particular, in this setup, the data word is always a capability, as it corresponds to the unsealed indirect sentry. Therefore, it could avoid the disjunction for the continuation of the logical relation for IE-cap. We leave such formalization as future work.

6.3 Implementing encapsulated closures with sentry capabilities

With sentry capabilities (E permission), a closure needs to embed both the code and the data contiguously. Moreover, it requires some trampoline code at the beginning of the closure to fetch the data and store it in the data register. As discussed in Section 1, trampoline code is undesirable and breaks the principle of least privilege. Figure 15 shows the result of the software implementation for creating closures using sentry capabilities.

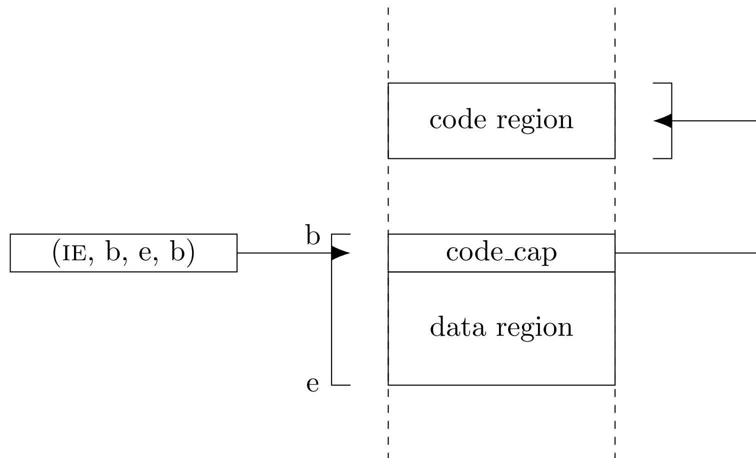


Figure 14: Indirect Sentry capability — Points-to-PCC

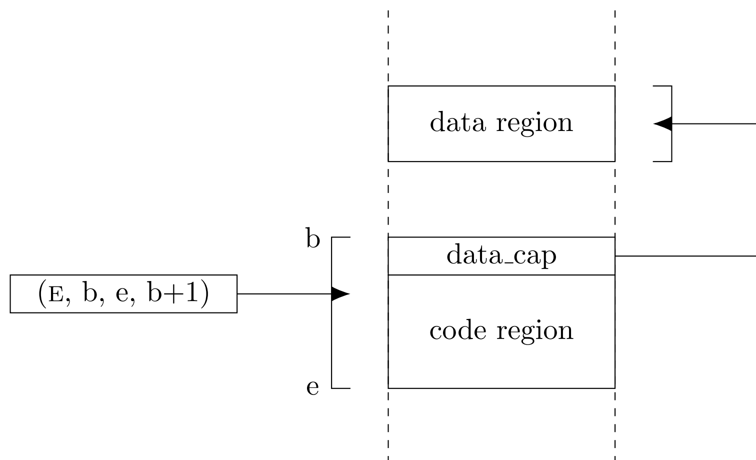


Figure 15: Sentry capability — Closure

Usually, the code (executable) segment and the data (writable) segment are kept separate (for caching and protection). This implementation requires the code segment to be writable (or the data segment to be executable), which not only breaks the old conventions, but also forces the authority of the code segment (respectively the data segment) to be more important than it could otherwise be. Another downside of this implementation is that, storing code requires a large number of instructions, which makes creation of closure less efficient than setting up indirect sentries.

6.4 Implementing encapsulated closures with sealed pairs

Seal capabilities offers are another way to create closures in capability machines. Sealing allows to mark capability as non-mutable and non-dereferencable. Sealed capability are sealed using an *otype*, which is controlled by sealing capabilities. Seals offer a form of data encapsulation. In CHERI, their main purpose is to offer a way to link a pair of code and data capabilities, by creating *sealed pairs*. Each closure requires a unique otype. It also comes with an additional instruction `Invoke`, which takes two sealed capabilities as arguments: the sealed code capability and the sealed data capability. Both capability needs to be sealed with the same otype for being invoked.

From a formal point of view, invocation of sealed pairs is challenging. CHERI has a permission CINVOKE for limiting the sealed capabilities that can be invoked, which, ease the formal reasoning. But it still requires complicated bookkeeping for all otypes involved, and requires strong engineering requirement [15]. Moreover, another downside of this approach is the limited number of otypes: 4 bits in 64-bits capabilities, and 18 bits in 128-bits capabilities. Because the typical use of otypes is to define the boundaries of domains (e.g. different libraries linked together), this limitation can be annoying. For instance, CHERIoT [1] has a very limited number of otypes. On the other hand, it is possible to create as many Indirect Sentry capabilities as we wish.

As already drawn out previously, Indirect Sentry capabilities are less expressive than `Invoke` of sealed pairs: the closure made with Indirect Sentry capability are tied to their unique context. Indirect sentries are less expressive, but they are easier to formally reason about. As we saw in Section 4, the logical relation for indirect sentries is very similar to the logical relation for regular entries.

6.5 Cerise's flavours

In the effort of formal studying security guarantees of CHERI-like machines, several work have been done.

Stack safety [14, 15, 8, 6] focus on efficient and secure calling convention, and enforce stack safety properties such as Local State Encapsulation and Well Bracketed Control Flow in the presence of an explicit stack. [8, 6] are fully mechanised in the Coq proof assistant. [14] uses local capabilities and implements closures with entry capabilities. [15] study the experimental linear capabilities and implements closures with sealed pair. [8] uses local capabilities, study experimental uninitialised capabilities and implements closures with entry capabilities. [6] uses local capabilities and the experimental uninitialised capabilities, study experimental directed capabilities, and implements closures with entry capabilities. Our work focuses on Local State Encapsulation only, and does not support local capabilities. However, we have good hope we can extend the aforementioned work with indirect sentries similarly.

Data abstraction 'Vanilla' Cerise [7] is a fully mechanised mini-CHERI machine that uses entry capabilities to implement closures. [16] extends Cerise with sealed capabilities, which enables the study of data abstraction. They use entry capabilities to implement closures, as they do not support invocation of sealed pairs.

7 Future Work

7.1 Secure Stack-based Calling Convention

In this section, we propose an extension of this work for an efficient and secure stack-based calling convention. In particular, we propose to revisit the work of Georges et al. [6], to create the return pointer with an indirect sentries instead of the regular sentry, which allows to remove the trampoline code from the stack. The formalization and verification of the calling convention is left as future work.

Monotone Cerise The Monotone Cerise machine in Georges et al. [6] extends Vanilla Cerise in a different direction from ours, focused on stack safety. It implements CHERI's *locality* bit used to separate stack-only capabilities from normal, so-called *global* capabilities, as well as two

experimental kinds of capabilities related to stack management: *uninitialized* capabilities [8] and *directed* capabilities [6].

In more detail, the locality bit is used as a temporary mechanism, and restrict the way a capability can be stored. If the locality is set to GLOBAL, the capability can be stored anywhere. If the locality is set to LOCAL, the capability can only be stored in the registers, or via a capability with the *write-local* (WL) permission. In the model, the stack capability is the only capability with WL permission. LOCAL capabilities prevent an adversary to store stack capabilities in its private memory.

A capability is an *uninitialized* capability if it has the U permission, where π is the regular permissions. The semantics of $\text{U}\pi$ is that the capability $(\text{U}\pi, g, b, e, a)$ has the π permission over the range $[b, a)$ and $\pi \setminus \{\text{R}, \text{X}\}$ over the $[a, e)$. The only way to move the bound a to higher addresses is to overwrite what was previously written at address a. Uninitialized capabilities prevent an adversary from storing stack capabilities in the stack in between two calls.

A capability is said *directed* if it has the DIRECTED locality. The semantics of DIRECTED is that the capability $(p, \text{DIRECTED}, b, e, a)$ cannot be stored in memory at higher addresses than the address it currently points-to, a. Directed capabilities means that the callee does not need to not clear its stack frame before returning to the caller.

Secure and Efficient Calling Convention Using the aforementioned features, Georges et al. [6] define a secure and efficient calling convention, ensuring local state encapsulation (LSE) and well-bracketed control flow (WBCF), and does not require any stack clearing. However, the calling convention uses entry capabilities as return pointers, which requires the trampoline code to be in the stack. As already discussed, trampoline code in the stack requires the stack to be executable, which breaks the principle of least privilege. As previously shown, trampoline code can be safely removed from the stack by using indirect sentries.

Figure 16 shows the state of the stack upon the jump to the next function in Monotone Cerise. The stack grows from low addresses (at the bottom) to high addresses (at the top). The key aspect is the *activation record* in the stack. For more explanation about the calling convention, see [6]. The activation record is a small trampoline code (4 instructions) that recovers the previous stack frame capability and jumps to the call site of the callee. Storing the trampoline code in the stack is unavoidable, because (1) the return pointer given to the caller needs to be a DIRECTED capability, in order to ensure that the caller does not store the pointer in its private memory; and (2) the caller needs to store its current stack frame capability in the closure of the return pointer, and the only memory region where the the stack capability can be stored is the stack itself.

The downside of this calling convention is that the stack needs to be executable, which does not really respect the principle of least privilege, as explained in §1.

Calling Convention with Indirect Sentries We propose a calling convention that uses indirect sentries instead of the regular entry capabilities, which thus makes it possible to implement callbacks without using trampoline code, and therefore means that the stack does not need to be made executable.

Figure 17 shows the state of the stack upon the jump to the next function, for our calling convention using indirect sentries. As opposed to calling conventions using regular entries, recovering the stack pointer of the caller at the callback does not require any trampoline code, as the semantic of the indirect sentries handles it.

In more details, the calling convention does the following:

1. Pushes the current environment, *i.e.,* the content of the register file, in the stack.

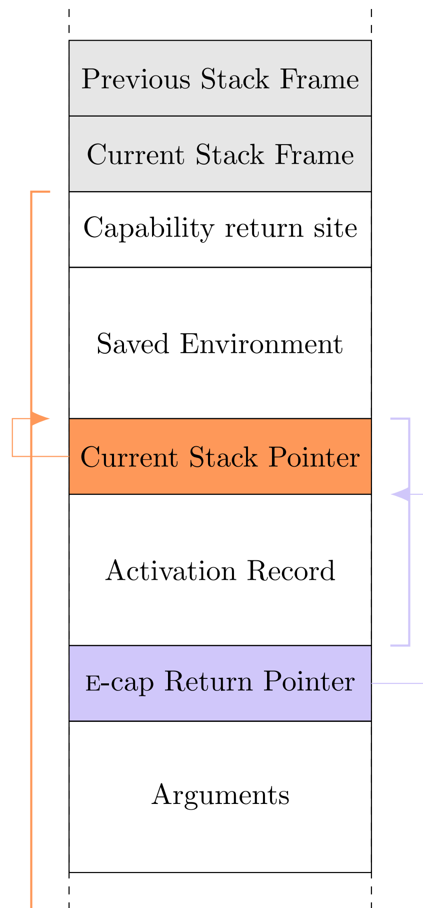


Figure 16: Stack-based Calling Convention with Entry capability and trampoline code. The stack grows bottom up.

2. Pushes the callback capability (PCC after the call macro) and the current stack capability on the stack. It corresponds to the pair that will be used for the indirect sentry.
3. Pushes the arguments of the call on the stack.
4. Creates and pushes an indirect sentry for the return capability.
5. Restricts the stack pointer to the new frame.
6. Clears the registers and jumps to the callee.

We expect the new calling convention to enforce the WBCF and the LSE security properties, but leave the proof of this for future work.

7.2 Technical Improvement

This section intends to point out some technical improvements that could be made on Cerise with Indirect Sentries, and informally propose possible directions to investigate in order to achieve those improvements. We leave such technical improvements as future work.

Jump to unknown word The Corollary 1 is slightly different from the original one in Cerise [7, §5.2]. Our specification start before executing the jump, whereas the original one starts after the execution of the jump, giving a form of continuation after the jump of a macro.

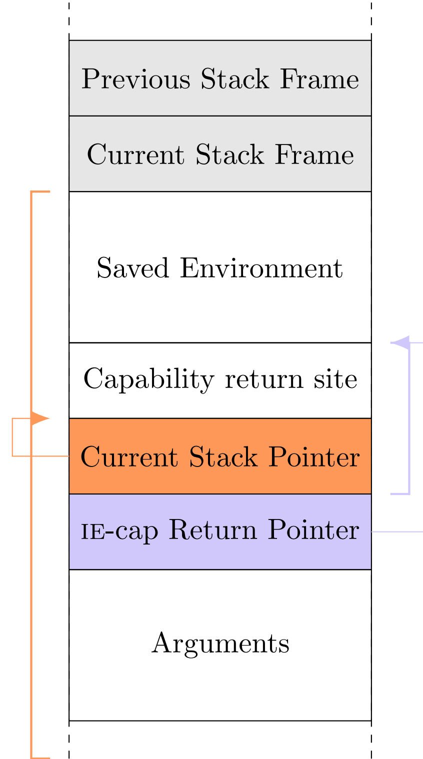


Figure 17: Stack-based calling convention using indirect sentry capabilities to avoid trampoline code. The stack grows from the bottom up.

Previously, the corollary *simulated* the jump. The main point of the corollary is to be used as a continuation point after jumping to a macro.

The reason of this difference is that, jumping to an indirect sentry requires the points-to resources of the indirection. In particular, to be able to jump to an arbitrary word w, the user needs to justifies that he owns the points-to resources. Moreover, the words in `pc` and `idc` depends on the content of indirection. However, the points-to predicates are only known after opening the invariant of $\mathcal{V}(w)$, in case it is an indirect sentry. In the case of specifying a macro with multiple instructions, the invariant cannot be opened before, and then cannot justify that it owns the points-to resources.

Our version of the corollary is an issue because it loses its modularity. First, Corollary 1 only states how to jump with the `jmp` instruction. We could easily define its counterpart for the `jnz` instruction, but it implies some duplication. Second, there is a lose of modularity when specifying a macro. Instead of specifying the postcondition of a macro to be “the state after the jump”, we need to specify the postcondition to be “the state before the jump”. Not only it usually require an heavy manipulation of the resources when using the specification, but it also clutters the specification of macros with different exit point. Finally, when specifying macros, we don’t know yet whether the jumping pointer is known by the user of the macro or not. Put another way, it is the responsibility of the user of the specification to justify the potentially necessary point-to resources, either if he directly owns the resources (in case the word is known), or if he only gets them from an invariant (*e.g.,* in case the word is unknown).

A direction to explore in order to recover some modularity is to define an HOCAP-style specification [5, §13].

Improvement Definition Logical Relation The logical relation for indirect sentries in Figure 9 defines the continuation predicate with a disjunction *is_int* over the data word. The proof of the corner case of the FTLR 4.3 highlights the technical reason of this *is_int trick*. Our study shows that this definition is complete enough to prove interesting use cases. But the consequences of this definition are that the user of the logical relation needs to prove two specifications of a closure: one with the expected $P_2(w_2)$, and one with the unexpected $is_int(w_2)$.A

We argue that the definition could be improved further, and remove the disjunction with *is_int* to only keep the expected predicate P_2. If the adversary has some control over the address of the data word, it suggests that the adversary created the indirect sentry. In particular, it means that the adversary also has some control over the address of the code word, which has to be safe to share. However, our current logical relation forgets the link between the code word and the data word. A first direction to explore would be to find a way to keep track of the link between the code word and the data word.

The technical reason of the *is_int* trick highlight that, the disjunction between the address of the pc capability and the indirect sentry we are jumping to, inherits from the program logic rules, which also do the distinction in the resources of the pre- and postcondition. Another direction to investigate would be to define the specifications in an alternative way, that does not require to distinguish the different cases to write the rules of the program logic

References

[1] Saar Amar et al. *CHERIoT: Rethinking security for low-cost embedded systems*. Tech. rep. MSR-TR-2023-6. Microsoft, Feb. 2023. URL: https://www.microsoft.com/en-us/research/uploads/prod/2023/02/cheriot-63e11a4f1e629.pdf.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[3] *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*. Tech. rep., p. 1294. (Visited on 01/19/2024).

[4] Thomas Bauereiss et al. "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*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 174–203. DOI: 10.1007/978-3-030-99336-8_7. URL: https://doi.org/10.1007/978-3-030-99336-8_7.

[5] Lars Birkedal and Aleš Bizjak. "Lecture Notes on Iris: Higher-Order Concurrent Separation Logic". In: (). URL: https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf (visited on 01/09/2024).

[6] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. "Le Temps Des Cerises: Efficient Temporal Stack Safety on Capability Machines Using Directed Capabilities". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). https://dl.acm.org/doi/10.1145/3527318, pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3527318. (Visited on 09/19/2022).

[7] Aïna Linn Georges et al. "Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code". In: 1.1 (), p. 55.

[8] Aïna Linn Georges et al. "Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities". In: *Proc. ACM Program. Lang.* 5.POPL (2021). DOI: 10.1145/3434287. URL: https://doi.org/10.1145/3434287.

[9] Sander Huyghebaert et al. "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*. Ed. by Weizhi Meng et al. ACM, 2023, pp. 2083–2097. DOI: 10.1145/3576915.3616602. URL: https://doi.org/10.1145/3576915.3616602.

[10] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151. URL: https://doi.org/10.1017/S0956796818000151.

[11] Kyndylan Nienhuis et al. "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1003–1020. DOI: 10.1109/SP40000.2020.00055. URL: https://doi.org/10.1109/SP40000.2020.00055.

[12] Bastien Rousseau, Aïna Linn Georges, and Lars Birkedal. *Cerise Exercises*. 2023. URL: https://github.com/logsem/cerise/blob/main/theories/exercises.

[13] Bastien Rousseau et al. *Proving capability safety in the presence of indirect sentries*. 2023. URL: https://github.com/logsem/cerise/tree/bastien/indirect-sentry.

[14] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management". In: *ACM Transactions on Programming Languages and Systems* 42.1 (Mar. 2020), pp. 1–53. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3363519. (Visited on 06/03/2022).

[15] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "StkTokens *: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities*". In: *Journal of Functional Programming* 31 (2021), e9. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S095679682100006X. (Visited on 06/03/2022).

[16] Thomas Van Strydonck and Dominique Devriese. *Cerise With Seals*. 2022. URL: https://github.com/logsem/cerise.

[17] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Tech. rep., 523 pages. DOI: 10.48456/TR-987. (Visited on 10/09/2023).

$$\textsc{ExecSingle}\quad (\mathsf{Running}, \sigma) \rightarrow \begin{cases} [\![decode(z)]\!](\sigma) & \text{if } \sigma.\mathrm{reg}(\mathsf{pc}) = (p, b, e, a) \land b \leq a < e \land \\ & \quad p \in \{\textsc{rx}, \textsc{rwx}\} \land \sigma.\mathrm{mem}(a) = z \\ (\mathsf{Failed}, \sigma) & \text{otherwise} \end{cases}$$

i	$[\![i]\!](\sigma)$	Conditions
`fail`	$(\mathsf{Failed}, \sigma)$	
`halt`	$(\mathsf{Halted}, \sigma)$	
`mov` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\sigma, \rho)$
`load` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto w])$	$\sigma.\mathrm{reg}(r_2) = (p, b, e, a)$ and $w = \sigma.\mathrm{mem}(a)$ and $b \leq a < e$ and $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\}$
`store` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{mem}.a \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a)$ and $b \leq a < e$ and $p \in \{\textsc{rw}, \textsc{rwx}\}$ and $w = \mathrm{getWord}(\sigma, \rho)$
`jmp` r	$\left(\mathsf{Executable}, \sigma\left[\begin{array}{l}\mathrm{reg}.\mathsf{pc} \mapsto newPc, \\ \mathrm{reg}.\mathsf{idc} \mapsto newIdc\end{array}\right]\right)$	if $\sigma.\mathrm{reg}(r) = (\textsc{ie}, b, e, a)$ then $b \leq a < a+1 < e$ $\land\ newPc = \sigma.\mathrm{mem}(a)$ $\land\ newIdc = \sigma.\mathrm{mem}(a+1)$ else $newPc = \mathrm{updatePcPerm}(\sigma.\mathrm{reg}(r))$ $\land\ newIdc = \sigma.\mathrm{reg}(\mathsf{idc})$
`jnz` $r_{\mathrm{dst}}\ r_{\mathrm{cond}}$	$\left(\begin{array}{l}\mathsf{Executable}, \\ \text{if } \sigma.\mathrm{reg}(r_{\mathrm{cond}}) \neq 0 \\ \text{then } \sigma\left[\begin{array}{l}\mathrm{reg}.\mathsf{pc} \mapsto newPc, \\ \mathrm{reg}.\mathsf{idc} \mapsto newIdc\end{array}\right] \\ \text{else } \mathrm{updPC}(\sigma)\end{array}\right)$	if $\sigma.\mathrm{reg}(r_{\mathrm{dst}}) = (\textsc{ie}, b, e, a)$ then $b \leq a < a+1 < e$ $\land\ newPc = \sigma.\mathrm{mem}(a)$ $\land\ newIdc = \sigma.\mathrm{mem}(a+1)$ else $newPc = \mathrm{updatePcPerm}(\sigma.\mathrm{reg}(r_{\mathrm{dst}}))$ $\land\ newIdc = \sigma.\mathrm{reg}(\mathsf{idc})$
`restrict` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a)$ and $p' = \mathrm{decodePerm}(\mathrm{getWord}(\sigma, \rho))$ and $p' \preccurlyeq p$ and $w = (p', b, e, a)$
`subseg` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a)$ and for $i \in \{1, 2\}$, $z_i = \mathrm{getWord}(\sigma, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1 < \mathrm{AddrMax}$ and $0 \leq z_2 \leq e$ and $p \neq \textsc{e}$ and $p \neq \textsc{ie}$ and $w = (p, z_1, z_2, a)$
`lea` $r\ \rho$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto w])$	$\sigma.\mathrm{reg}(r) = (p, b, e, a)$ and $z = \mathrm{getWord}(\sigma, \rho)$ and $p \neq \textsc{e}$ and $p \neq \textsc{ie}$ and $w = (p, b, e, a + z)$
`add` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \mathrm{getWord}(\sigma, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 + z_2$
`sub` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \mathrm{getWord}(\sigma, \rho_i)$ and $z_i \in \mathbb{Z}$ and $z = z_1 - z_2$
`lt` $r\ \rho_1\ \rho_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r \mapsto z])$	for $i \in \{1, 2\}$, $z_i = \mathrm{getWord}(\sigma, \rho_i)$ and $z_i \in \mathbb{Z}$ and if $z_1 < z_2$ then $z = 1$ else $z = 0$
`getp` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto z])$	$\sigma.\mathrm{reg}(r_2) = (p, _, _, _)$ and $z = \mathrm{encodePerm}(p)$
`getb` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto b])$	$\sigma.\mathrm{reg}(r_2) = (_, b, _, _)$
`gete` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto e])$	$\sigma.\mathrm{reg}(r_2) = (_, _, e, _)$
`geta` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto a])$	$\sigma.\mathrm{reg}(r_2) = (_, _, _, a)$
`isptr` $r_1\ r_2$	$\mathrm{updPC}(\sigma[\mathrm{reg}.r_1 \mapsto z])$	if $\sigma.\mathrm{reg}(r_2) = (_, _, _, _)$ then $z = 1$ else $z = 0$
$_$	$(\mathsf{Failed}, \sigma)$	otherwise

$$\mathrm{updPC}(\sigma) = \begin{cases} (\mathsf{Running}, \sigma[\mathrm{reg}.\mathsf{pc} \mapsto (p, b, e, a+1)]) & \text{if } \sigma.\mathrm{reg}(\mathsf{pc}) = (p, b, e, a) \\ (\mathsf{Failed}, \sigma) & \text{otherwise} \end{cases}$$

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

$$\mathrm{updatePcPerm}(w) = \begin{cases} (\textsc{rx}, b, e, a) & \text{if } w = (\textsc{e}, b, e, a) \\ w & \text{otherwise} \end{cases}$$

Figure 18: Operational semantics: execution of a single instruction.

A Full Operational Semantics

B Full spec example call

$$
\boxed{\text{mallocInv}(b_m, e_m)} * \boxed{\text{assertInv}(b_a, e_a)}
$$

$$
\vdash \left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{a});\ \begin{array}{l}
[\mathsf{code}, \mathsf{a}) \mapsto \texttt{pre_call_instrs} * \\
[\mathsf{a}, \mathsf{cont}) \mapsto \texttt{call_instrs} * \cdots * \\
[\mathsf{cont}, \mathsf{end}) \mapsto \texttt{post_call_instrs} * \\
\mathsf{data} \mapsto (\mathrm{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) * \\
\mathsf{table} \mapsto (\mathrm{E}, b_m, e_m, b_m) * \\
(\mathsf{table} + 1) \mapsto (\mathrm{E}, b_a, e_a, b_a) * \\
[b_{mem}, a_{\mathit{off}}) \mapsto [0; \cdots; private_{val}] * \\
[a_{\mathit{off}}, e_{mem}) \mapsto [0; \cdots; 0] * \\
\mathsf{r}_7 \Mapsto (\mathrm{RWX}, a_{\mathit{off}}, e_{mem}, a_{\mathit{off}}) * \\
\mathsf{r}_{30} \Mapsto w_{adv} * \mathcal{V}(w_{adv}) * \\
\mathsf{r}_8 \Mapsto (\mathrm{RWX}, b_{mem}, a_{\mathit{off}}, b_{mem}) * \\
\mathop{\huge *}_{\substack{(r,v) \in reg, \\ r \notin \{\mathsf{pc}, target\} \\ r \notin params\ \cup\ locals}} r \Mapsto v * \mathcal{V}(v)
\end{array} \right\} \rightsquigarrow
$$

$$
\left\{ (p, \mathsf{code}, \mathsf{end}, \mathsf{cont} - 1);\ \begin{array}{l}
\exists b_{ie}, e_{ie}, l, l_{end}, reg'. \\
[\mathsf{code}, \mathsf{a}) \mapsto \texttt{pre_call_instrs} * \\
[\mathsf{a}, \mathsf{cont}) \mapsto \texttt{call_instrs} * \cdots * \\
[\mathsf{cont}, \mathsf{end}) \mapsto \texttt{post_call_instrs} * \\
\mathsf{data} \mapsto (\mathrm{RO}, \mathsf{table}, \mathsf{end}, \mathsf{table}) * \\
\mathsf{table} \mapsto (\mathrm{E}, b_m, e_m, b_m) * \\
(\mathsf{table} + 1) \mapsto (\mathrm{E}, b_a, e_a, b_a) * \\
[b_{mem}, a_{\mathit{off}}) \mapsto [0; \cdots; private_{val}] * \\
[a_{\mathit{off}}, e_{mem}) \mapsto [0; \cdots; 0] * \\
[l, l_{end}) \mapsto [(\mathrm{RWX}, b_{mem}, a_{\mathit{off}}, b_{mem})] * \\
[b_{ie}, e_{ie}) \mapsto [(p, \mathsf{code}, \mathsf{end}, \mathsf{cont}); (\mathrm{RWX}, l, l_{end}, l_{end})] * \\
\mathsf{r}_7 \Mapsto (\mathrm{RWX}, a_{\mathit{off}}, e_{mem}, a_{\mathit{off}}) * \\
\mathsf{r}_{30} \Mapsto w_{adv} * \mathcal{V}(w_{adv}) * \\
\mathsf{r}_{31} \Mapsto (\mathrm{IE}, b_{ie}, e_{ie}, b_{ie}) * \\
\mathop{\huge *}_{\substack{(r,v) \in reg', \\ r \notin \{\mathsf{pc}, target, \mathsf{r}_{31}\} \\ r \notin params}} r \Mapsto v * \mathcal{V}(v)
\end{array} \right\}
$$