Amarena: Reasoning about Revoking and Suspending Memory Access on Capability Machines

YOUIE LY, KU Leuven, Belgium
DOMINIQUE DEVRIESE, KU Leuven, Belgium

Verification of robust safety on capability machines has seen steady progress toward more realistic machine architecture, more advanced security mechanisms, and better proof automation. However, none of this recent progress supports reasoning about both revoking and suspending delegated memory access. All earlier work that does support it resorts to a Kripke logical relation indexed by worlds with public and private transitions, an approach that has drawn recurring criticism for its complexity and poor usability. We argue that a simpler, reusable foundation for reasoning about revocation and suspension is needed for such verification to scale, for example, to larger machine models and more complex, interacting security mechanisms.

We take a first step toward such a foundation by contributing an approach for expressing invariants that hold only for a limited lifetime (e.g., until delegated access is revoked). We apply this approach to develop **Amarena**, a program logic and logical relation built on Cerise and mechanized in Rocq. Amarena supports reasoning about a stack-sharing calling convention that revokes and suspends memory access by limiting where capabilities can be retained in memory.

Our approach follows the lead of the broader literature on logical relations, moving away from explicit world reasoning toward reusable rules that take full advantage of Iris's ghost resources and invariants. Compared with the Kripke approach, our approach's specialized machinery for reasoning about revocation and suspension is conceptually simpler and offers better compositionality. It is independent of any particular calling convention or program, and is non-invasive, invoked only where revocation or suspension is relevant.

CCS Concepts: • **Theory of computation** → **Separation logic**; **Program verification**.

Additional Key Words and Phrases: Cerise, capability machines, Iris, program logic, logical relations

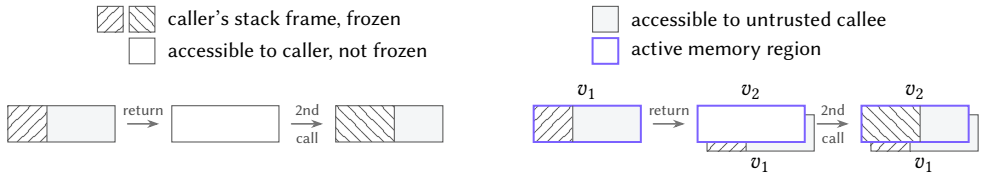
1 Introduction

Capability machines, based on ideas dating back to the 1960s [5, 7, 33, 40], have attracted surging research interest over the past two decades because of their hardware support for fine-grained access control and spatial memory safety [3, 6, 10–12, 15, 18, 21, 22, 35, 36, 42, 44, 51, 52, 54–57]. Much of this interest centers on Capability Hardware Enhanced RISC Instructions (CHERI) [32, 53–55], which extends pointers with metadata such as memory bounds and permissions to restrict their use. The hardware prevents components from forging capabilities from integers, widening capability bounds, or increasing permissions. CHERI has also drawn industry attention (e.g., Arm's Morello processor [19] and Microsoft's CHERIoT platform for embedded devices [4]).

On capability machines, carefully *choosing which capabilities to delegate* is vital for guarding against malicious and unintended behavior: a component's memory access is restricted by the capabilities it initially holds and those it later receives. Supporting reliable capability delegation is therefore an active area of research, with new proposals for architectures, language extensions, and security mechanisms emerging rapidly [3, 6, 11, 12, 15, 18, 21, 22, 35, 42, 44, 56, 57].

A growing line of work has proved remarkably strong security guarantees for some of these proposals [14, 15, 18, 23, 25, 30, 35, 39, 42, 43], mostly mechanized in Rocq [46] (formerly Coq [47]). These guarantees establish *robust safety*: verified security properties hold even when trusted components are linked with untrusted code. Much recent work in this line [23, 25, 30, 39] draws on Cerise [14], which pairs a program logic built on Iris [28] with a logical relation [1]. At the core of the logical relation are *safety assertions*, capturing whether a capability is safe to share with an

Authors' Contact Information: Youie Ly, KU Leuven, Leuven, Belgium, youie.li@kuleuven.be; Dominique Devriese, KU Leuven, Leuven, Belgium, dominique.devriese@kuleuven.be.



(a) Temporary invariant challenge (b) Logical memory approach

Fig. 1. The temporary invariant challenge and the logical memory approach. (a) The caller's stack frame must stay frozen in an invariant while an untrusted callee runs. When the callee returns, the caller pops the frame and, for its next call, pushes a different one at the same addresses, thereby violating the invariant. (b) Logical memory solves the problem by allocating a fresh version v_2 when the callee returns, pushing the caller's next stack frame there, protecting it with a new invariant, and making v_2 active. Invariants over v_1 remain valid.

adversary. If every capability the adversary can access is safe to share, then the adversary preserves the invariants that trusted code establishes about the machine state throughout the execution.

While verification of robust safety on capability systems has seen steady progress toward more realistic machine architecture [23], more advanced security mechanisms [30, 39], and better proof automation [25], none of this recent progress supports reasoning about both *revoking and suspending*[1] *delegated memory access*. Yet revocation and suspension (*temporary delegation* for short) are just as crucial to reliable capability delegation as choosing which capabilities to share.

Unfortunately, reasoning about temporary delegation has turned out to be extremely complicated. All earlier work that supports it resorts to a *Kripke logical relation indexed by worlds with public and private transitions* [8] (the *Kripke logical relation*, or KLR, for short). Unlike its predecessors [15, 18, 43] (hereafter *Cerise with KLR*), Cerise [14, 16] (hereafter *base Cerise*) deliberately drops this support and adopts a much simpler logical relation to avoid the complexity. For example, in Cerise with KLR [15], verifying just six ML statements, whose assembly translation involves temporary delegation, takes ~2.8k proof lines, excluding proofs of reusable specifications.

Working directly with Kripke logical relations has drawn recurring criticism inside and outside capability machine verification, for their complexity and poor usability [9, 14, 48–50] and for being unintuitive and hard to understand [37, 42]. Proving examples was described as "painfully tedious, low level, and difficult to maintain" [50]. Reasoning about worlds also has poor support for framing, forcing proofs to account for the whole world even when only part of it changes, and is *invasive* [49, 50] in the sense that the Kripke machinery is invoked even where the features that demand its expressiveness are irrelevant (e.g., a program that never revokes access).

Research gap. Base Cerise has become a foundation for verification of robust safety on capability systems but does not support temporary delegation. The only known approach that does, the Kripke logical relation, is hard to work with. We argue that a simpler, reusable foundation for reasoning about temporary delegation is needed for such verification to scale, for example, to larger machine models and more complex, interacting security mechanisms. Ideally, this foundation extends base Cerise, so that existing work carries over and future work can add support for temporary delegation at little cost. The broader literature on logical relations also advocated and demonstrated the benefits of moving away from explicit world reasoning toward reusable rules, working directly with Iris's ghost resources and invariants, and introducing non-invasive machinery [13, 27, 29, 49, 50].

We follow this lead and take a first step toward such a foundation by contributing an approach to the central challenge in reasoning about temporary delegation: expressing *temporary* invariants

[1]Suspension means that a component's access is revoked temporarily and later restored.

over *revocable regions*. Revocable regions are regions where a component's access can be revoked or suspended. Temporary means these invariants' validity is guaranteed only for a limited lifetime (e.g., until delegated access is revoked). Consider stack sharing in Fig. 1a. A caller pushes its stack frame (hatched) and shares the rest of the stack with an untrusted callee. While the callee runs, the frame must remain inaccessible to the callee and any code it calls. With a Cerise-style logical relation, the natural way to prove this is to establish an invariant that freezes the frame's contents and show that it holds while the callee runs. But an invariant, once established, holds permanently. After the callee returns, the caller regains access to the revoked stack region and reuses the same addresses to push a new stack frame (hatched in the other direction) when it makes a second call, thereby violating the invariant. Worse, it also breaks safety assertions because they depend on such invariants. The stack capability given to the callee in Fig. 1a was safe to share during the call, but becomes unsafe when the caller pushes a larger stack frame that overlaps the capability's range.

To support ordinary invariants and safety assertions over revocable regions, we draw inspiration from a technique called *logical memory* [24]. In logical memory, a physical address can have several *logical versions*, each with its own contents. Only the active version of an address describes the address's contents in physical memory. Writes to one version do not break invariants or safety assertions over other versions. For the example in Fig. 1a, before the caller pushes its second frame, we *logically reclaim* the stack region by allocating and activating a fresh version v_2. The caller can then push the new frame to v_2, while invariants over v_1 remain valid. The key insight is that, for a revocable region, we establish a separate invariant for each revocation stage, without considering its contents at other stages. The Kripke logical relation instead combines multiple stages in a single invariant indexed by a world, forcing the user to reason about them at once.

Challenges. While this idea seems promising, adapting logical memory raises three challenges.

Component-local unreachability: Logical memory was originally developed for reasoning about garbage collection, where reclaiming a region is justified by its global unreachability from GC roots. In our setting, when temporarily delegated access is revoked, the revoked region remains reachable from the component that delegated the access. Such *component-local* unreachability violates the global unreachability premise underlying the soundness of using logical memory.

Revocation may be reversed: Restoring suspended accesses to a region can require reactivating an earlier version of that region (v_1 in Fig. 1b). The original logical memory approach does not support reactivation because reclamation is permanent in garbage collection.

Well-bracketed control flow: Reliable suspension and restoration of access often depend on enforcing well-bracketed control flow, in which every callee either returns to its immediate caller, makes a nested call, or diverges. Without such enforcement, an untrusted callee could return past its caller and skip the restoration of suspended accesses. Our approach therefore aims to support reasoning about well-bracketedness, which base Cerise leaves out along with the Kripke logical relation [14]. However, logical memory alone does not track how calls nest and cannot rule out returns that break well-bracketedness.

To solve the first two challenges, we introduce *version points-to* assertions $a \dashrightarrow v$, which state that the active version of address a is v and behave much like memory points-to assertions. Intuitively, they act as an indirection that determines which version's memory points-to assertion reflects the current contents of a. Logical reclamation and restoration then become ghost operations that redirect this indirection. For example, reclamation satisfies the following simplified rule:

$$a_v \mapsto w * a \dashrightarrow v \;\vdash\; a_v \mapsto w * \exists v_f.\, a_{v_f} \mapsto w * a \dashrightarrow v_f$$

It allocates a fresh active version v_f for address a with the same stored value w, while preserving the memory resource for a_v, address a at the previously active version v.

Reasoning about well-bracketed control flow is known to be hard in high-level languages and on capability machines alike [8, 35, 49]. To address the third challenge, we adapt *ghost stacks* [49], a recently developed technique for reasoning about well-bracketedness in high-level languages.

Contributions. Our methodological contribution is a novel approach, inspired by logical memory, for expressing invariants that hold only for a limited lifetime, for example, until a callee returns. Our adaptation takes logical memory beyond garbage collection and, to the best of our knowledge, is the first to support both component-local unreachability and the restoration of suspended accesses. Both violate assumptions underlying the original approach's soundness.

We apply our approach to a stack-based calling convention using local capabilities [43] (§ 2.4) and make the technical contribution **Amarena**:

- A program logic extended with logical memory and version points-to assertions (§§ 4.1–4.3) and a logical relation additionally extended with ghost stacks (§ 4.4), which together support proofs of robust safety guarantees for this calling convention.
- Proofs of examples from base Cerise [14] and Cerise with KLR [15, 43], including a simpler proof of the *Very Awkward Example* [8] than in Georges et al. [15], in a smaller development (§ 7.3).
- A mechanization of Amarena in Rocq, building on top of Iris and Cerise.

The main strengths of our approach are:

- Preservation of base Cerise (§ 7.1): Our logical relation follows base Cerise's logical relation, using ordinary invariants instead of worlds and their state transition systems.
- Non-invasiveness and orthogonality (§ 7.2): Proofs invoke the specialized machinery—logical reclamation and ghost stack reasoning—independently and only where relevant.
- Better compositionality (§ 8): Reclamation and restoration require reasoning only about the affected memory regions and their invariants, leaving unrelated resources framed.
- Generality (§ 8): The logical memory layer and ghost stacks are independent of any particular calling convention or program.

2 Background

This section follows the program that motivates our setting, the very awkward example, from its high-level form down to its capability-machine translation. Along the way, we review the Cerise program logic and logical relation [14] that Amarena builds on, and the calling convention [43] that upholds, in that translation, the assumptions behind the high-level program's correctness.

2.1 The Very Awkward Example: From Well-Bracketedness to Revocation

The *very awkward example* (VAE) [8] is a canonical program whose correctness relies on well-bracketed control flow. The program is defined as follows[2]:

$$\mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\ \lambda f.\, x \leftarrow 0; f(); x \leftarrow 1; f(); \mathsf{assert}(!x = 1) \tag{VAE}$$

It allocates a reference x and creates a closure that takes a function argument f. When invoked, the closure sets x to 0, calls f, then sets x to 1, calls f again, and asserts that x is still 1.

At first glance, the assert command seems to always succeed. The variable x is local to the closure, and the closure never intentionally shares access to x with f. Function f cannot modify x directly. The only visible modifications of x are the assignments performed by the closure itself. However, this reasoning assumes that f behaves in a well-bracketed manner, which can be expressed by the Hoare triple

$$\{\mathsf{True}\}\ f()\ \{y.\, y = ()\}$$

[2]The original VAE program returns the value of x directly. Here, we use the assert command to make it easier to compare with our Cerise translation in § 2.5.

We take it to mean that if f returns, f returns the unit value without causing an assertion to fail.

When VAE is translated to our assembly-level setting, this high-level assumption becomes a proof obligation about the security of the underlying calling convention. Program VAEFail shows a concrete example where the assertion fails when f makes a nested call to the closure:

$$x \leftarrow 0;\ f();\ x \leftarrow 1;\ f() \qquad ;❹\ \mathsf{assert}(!x = 1)$$
$$\underbrace{x \leftarrow 0;\ g();❸\ x \leftarrow 1;\ g();\ \mathsf{assert}(!x = 1)}_{\text{nested call to the closure}} \qquad \text{(VAEFail)}$$

During the second call to f, the function receives a stack pointer and a return pointer to the assert command, marked by ❹. It can store this return pointer (for example, higher on the stack) and make a nested call to the closure, passing an accomplice function g. The nested call resets x to 0 and calls g with a second return pointer, marked by ❸. Instead of returning to ❸, g retrieves the earlier return pointer to ❹ and jumps to it. Execution resumes at the assert command at ❹, even though the nested call is not finished. Since the nested call just reset x to 0, the assertion that $x = 1$ fails.

A secure calling convention prevents this attack by temporarily revoking the return pointer ❹ while $g()$ runs, and restoring it before the nested call returns. In § 2.5, we explain how an existing convention [43] achieves this by enforcing *local state encapsulation*—a callee cannot reach stack frames other than its own—and well-bracketed control flow. The assembly translation of VAE is therefore an ideal stress test for both properties and for the proof methods used to verify them.

2.2 Capability Machine Model and the Cerise Program Logic

Capability machine model. The capability machine we use has the syntax and operational semantics of the one in Georges et al. [14], extended with the local-capability constructs of Skorstengaard et al. [43]. Fig. 2 summarizes part of the syntax, with local-capability-related constructs and instructions highlighted in yellow. The machine state φ is a pair of a register file and memory. Memory *mem* is a contiguous array of cells addressed from 0 to AddrMax. Register file *reg* contains general-purpose registers r_0 through r_{31} and a program counter pc. Both are finite maps from addresses or register names to machine words. A machine word is either an integer or a capability. The machine distinguishes them strictly, and neither can be converted into the other. Instructions are themselves integers, decoded at runtime.

A capability is a tuple (p, l, b, e, a). It points at address a for memory access and grants permission p, which can combine read R, write W, write-local WL, and execute X permissions. Permissions and locality form a lattice, ordered by $\preceq$, with O (opaque, no access) at the bottom and RWLX at the top, so RWX $\preceq$ RWLX. Local capabilities sit lower in the lattice than global capabilities Local $\preceq$ Global. A capability can be weakened to any lower permission or locality. The write-local permission allows writing a local capability into its range, and is useful mostly for accessing the stack. For our purposes, local capabilities can be stored only in registers or stack memory, so any copies an adversary saves in memory are confined to the stack. The current address can be moved within or outside the range $[b, e)$, but only accesses within $[b, e)$ are allowed.

A capability with permission E is a *sentry*, which encapsulates a compartment that typically holds executable code and private state. A sentry does not give read or write access to its region, and its bounds and permissions cannot be modified. The only allowed operation is a jump, which turns the sentry into an RX capability and starts executing the code at its current address. Sentries are therefore the primary way to switch between mutually distrusting compartments.

Program logic. The Cerise program logic is built on Iris [28], a higher-order separation logic [38], and reasons about known programs on this machine. Fig. 3 shows part of its syntax. Throughout this section, purple highlighting marks the Amarena extensions introduced in § 5. We review only what we use and refer to Georges et al. [14] for the full account. Cerise has the usual quantifiers

$$
\begin{array}{lll}
a \in \mathrm{Addr} & \triangleq [0, \mathrm{AddrMax}] & r \in \mathrm{RegName} \quad ::= \mathtt{pc} \mid r_0 \mid r_1 \mid \cdots \mid r_{31} \\
p \in \mathrm{Perm} & ::= \mathrm{O} \mid \mathrm{E} \mid \mathrm{RO} \mid \mathrm{RX} \mid \mathrm{RW} \mid \mathrm{RWX} \mid \mathrm{RWL} \mid \mathrm{RWLX} & \rho \in \mathbb{Z} + \mathrm{RegName} \\
l \in \mathrm{Locality} & ::= \mathrm{Local} \mid \mathrm{Global} & \mathit{reg} \in \mathrm{Reg} \quad \triangleq \mathrm{RegName} \rightarrow_{\mathrm{fin}} \mathrm{Word} \\
c \in \mathrm{Cap} & \triangleq \{(p, l, b, e, a) \mid b, e, a \in \mathrm{Addr}\} & \mathit{mem} \in \mathrm{Mem} \quad \triangleq \mathrm{Addr} \rightarrow_{\mathrm{fin}} \mathrm{Word} \\
w \in \mathrm{Word} & \triangleq \mathbb{Z} + \mathrm{Cap} & \varphi \in \mathrm{ExecConf} \quad \triangleq \mathrm{Reg} \times \mathrm{Mem} \\
i \in \mathrm{Instr} & ::= \mathtt{fail} \mid \mathtt{halt} \mid \mathtt{jmp}\ r \mid \mathtt{jnz}\ r\ r \mid \mathtt{mov}\ r\ \rho \mid \mathtt{load}\ r\ r \mid \mathtt{store}\ r\ \rho \mid \mathtt{add}\ r\ \rho\ \rho \mid \mathtt{sub}\ r\ \rho\ \rho \mid \mathtt{lt}\ r\ \rho\ \rho \\
& \mid \mathtt{lea}\ r\ \rho \mid \mathtt{restrict}\ r\ \rho \mid \mathtt{subseg}\ r\ \rho\ \rho \mid \mathtt{getp}\ r\ r \mid \mathtt{getl}\ r\ r \mid \mathtt{gete}\ r\ r \mid \mathtt{geta}\ r\ r \mid \mathtt{getb}\ r\ r
\end{array}
$$

Fig. 2. A selection of the machine syntax.

$$
\begin{array}{rll}
P, Q \in \mathrm{iProp} ::= & \mathrm{True} \mid \mathrm{False} \mid \forall x.\, P \mid \exists x.\, P \mid P \vee Q \mid \cdots & \text{higher-order logic} \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ulcorner \phi \urcorner \mid \Box P \mid \triangleright P \mid \Rrightarrow P \mid P \Rrightarrow Q \mid \boxed{P} \mid \cdots & \text{separation logic} \\
& \mid r \Mapsto l\, w \mid a_v \mapsto l\, w \mid [b, e)_v \mapsto l\, ws & \text{machine resources} \\
& \mid a \dashrightarrow v \mid a \dashrightarrow\!\Box\, v \mid [b, e) \dashrightarrow v \mid [b, e) \dashrightarrow\!\Box\, v & \text{version resources} \\
& \mid \{P\} \rightarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \bullet & \text{program logic}
\end{array}
$$

Fig. 3. Syntax of the Cerise [14] and the Amarena program logic. Purple highlights mark (part of) the constructs by which Amarena extends Cerise.

$$
\mathrm{decode} : \mathbb{Z} \rightarrow \mathrm{Instr} \qquad \mathrm{Local}(l\, w) \triangleq \begin{cases} \mathrm{True} & \text{if } l\, w = (-, \mathrm{Local}, -, -, -, -) \\ \mathrm{False} & \text{otherwise} \end{cases}
$$

$$
\frac{\mathrm{decode}(n) = \mathtt{store}\ r_{\mathtt{dst}}\ r_{\mathtt{src}} \qquad b \le a < e \qquad \text{if } \mathrm{Local}(l\, w) \text{ then } \mathrm{RWL} \preceq p \text{ else } \mathrm{RW} \preceq p \qquad \cdots}{\left\{ \begin{array}{l} \mathtt{pc} \Mapsto (p_{\mathtt{pc}}, l_{\mathtt{pc}}, b_{\mathtt{pc}}, e_{\mathtt{pc}}, a_{\mathtt{pc}}, v_{\mathtt{pc}}) * \\ a_{\mathtt{pc}, v_{\mathtt{pc}}} \mapsto n * a_{\mathtt{pc}} \dashrightarrow_{q_1} v_{\mathtt{pc}} * \\ r_{\mathtt{dst}} \Mapsto (p, l, b, e, a, v) * r_{\mathtt{src}} \Mapsto l\, w * \\ a_v \mapsto - * a \dashrightarrow_{q_2} v \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \mathtt{pc} \Mapsto (p_{\mathtt{pc}}, l_{\mathtt{pc}}, b_{\mathtt{pc}}, e_{\mathtt{pc}}, a_{\mathtt{pc}} + 1, v_{\mathtt{pc}}) * \\ a_{\mathtt{pc}, v_{\mathtt{pc}}} \mapsto n * a_{\mathtt{pc}} \dashrightarrow_{q_1} v_{\mathtt{pc}} * \\ r_{\mathtt{dst}} \Mapsto (p, l, b, e, a, v) * r_{\mathtt{src}} \Mapsto l\, w * \\ a_v \mapsto l\, w * a \dashrightarrow_{q_2} v \end{array} \right\}}
$$

$$
\frac{\mathrm{decode}(n) = \mathtt{jmp}\ \mathtt{r} \qquad \text{if } p = \mathrm{E} \text{ then } p_{new} = \mathrm{RX} \text{ else } p_{new} = p \qquad \cdots}{\left\{ \begin{array}{l} \mathtt{pc} \Mapsto (p_{\mathtt{pc}}, l_{\mathtt{pc}}, b_{\mathtt{pc}}, e_{\mathtt{pc}}, a_{\mathtt{pc}}, v_{\mathtt{pc}}) * \\ a_{\mathtt{pc}, v_{\mathtt{pc}}} \mapsto n * a_{\mathtt{pc}} \dashrightarrow_{q_1} v_{\mathtt{pc}} * r \Mapsto (p, l, b, e, a, v) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \mathtt{pc} \Mapsto (p_{new}, l, b, e, a, v) * \\ a_{\mathtt{pc}, v_{\mathtt{pc}}} \mapsto n * a_{\mathtt{pc}} \dashrightarrow_{q_1} v_{\mathtt{pc}} * r \Mapsto (p, l, b, e, a, v) \end{array} \right\}}
$$

Fig. 4. Specifications for the instructions `store` and `jmp`. Dots stand for the conditions we leave out here.

$$
\mathcal{E}(w) \triangleq \forall\, \mathit{reg}.\, \left\{ \mathtt{pc} \Mapsto w * \left(\mathop{\Large *}_{(r, w_r) \in \mathit{reg},\, r \neq \mathtt{pc}} r \Mapsto w_r * \mathcal{V}(w_r) \right) \right\} \rightsquigarrow \bullet
$$

$$
\mathcal{V}(w) \triangleq \begin{cases}
\mathcal{V}(z) & \triangleq \mathrm{True} \quad \text{for } z \in \mathbb{Z} \\
\mathcal{V}(\mathrm{O}, -, -, -) & \triangleq \mathrm{True} \\
\mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\mathrm{RX}, b, e, a) \\
\mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e)} \exists P.\, \boxed{\exists\, w.\, a \mapsto w * P(w)} * \text{read-safe } (a, P) \\
\mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e)} \exists P.\, \boxed{\exists\, w.\, a \mapsto w * P(w)} * \text{read-safe } (a, P) * \text{write-safe } (a, P)
\end{cases}
$$

$$
\text{read-safe } (a, P) \triangleq \triangleright \Box\, (\forall\, w.\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)) \qquad \text{write-safe } (a, P) \triangleq \triangleright \Box\, (\forall\, w.\, \mathcal{V}(w) \mathrel{-\!\!*} P(w))
$$

Fig. 5. The Cerise [14] logical relation, defining "safe-to-share $\mathcal{V}$" and "safe-to-execute $\mathcal{E}$" words.

and connectives ($\forall$, $\exists$, $\vee$), the separating conjunction $P * Q$ over disjoint resources, the magic wand $P \mathrel{-\!\!*} Q$ for implication, and the pure assertion $\ulcorner \phi \urcorner$ for meta-level facts ϕ such as $x = 0$. A proposition is either *ephemeral*—possibly invalidated later and not duplicable—or *persistent*—permanently true once established and freely duplicable. The persistent modality $\Box P$ asserts the duplicable part of P. The later modality $\triangleright P$ means, roughly, that P holds one execution step later. An invariant $\boxed{P}$ keeps

P true across every execution step, even when P is ephemeral. The invariant can be opened to obtain P for one atomic step, after which P must be re-established. The update modality $\Rrightarrow$ marks a fancy update, and $P \Rrightarrow\!\!* Q$ abbreviates $P \mathrel{-\!\!*} \Rrightarrow Q$. We omit later modalities and the invariant masks that prevent reopening an invariant.

Points-to assertions connect the logic to the machine state: $r \Mapsto w$ says register r stores w, and $a \mapsto w$ gives exclusive read–write access to address a, which currently holds w. The assertion $[b, e) \mapsto ws$ gives the same access to a contiguous range of memory holding the words ws. Fig. 3 also lists three triples, all built on Iris weakest preconditions, for a single instruction $\{P\} \to \{s.\, Q\}$, a code fragment $\{P\} \rightsquigarrow \{s.\, Q\}$, and a complete execution $\{P\} \rightsquigarrow \bullet$, where s is the resulting execution state. The last triple asserts safe execution: it holds when the machine, started from P, halts, faults, or diverges while preserving all invariants.

We refer to Georges et al. [15] for the full operational semantics. To illustrate the program logic [14, 15], we highlight the rules for `store` and `jmp` in Fig. 4. The `store` rule says that the instruction writes the word w from the source register to address a, provided the destination register holds a capability with write permission pointing to a within its range $[b, e)$. If w is a local capability, the destination capability also needs write-local permission. The `jmp` rule says that executing `jmp` r copies the capability (p, l, b, e, a) from register r to the program counter. If the capability is a sentry ($p = \mathsf{E}$), the copy's permission becomes RX. These rules have no program expression between their pre- and post-conditions because the machine is stateful and the instruction to execute is already encoded in memory at the address held in the program counter.

2.3 Cerise Logical Relation: Reasoning about Untrusted Code

Using the program logic in § 2.2, we can step through known programs and reason about how machine states evolve. When the program uses a `jmp` instruction to call an adversary whose code is unknown, it passes along all capabilities in the register file. The essence of verifying robust safety on capability machines is to prove the adversary cannot use these capabilities to break desirable properties of the machine state, expressed as Iris invariants established in the program logic.

Logical relations are a natural fit for this setting because, although the adversary's code and capabilities are unknown, the new capabilities it receives are syntactically known. A Cerise-style logical relation [14] defines what it means for a capability to be safe to share $\mathcal{V}$ with an adversary, judging from the capability's syntax, and what it means for a capability to be safe to execute $\mathcal{E}$ by an adversary, provided that the rest of the register file is safe. The *fundamental theorem of logical relations* (FTLR) says that if a capability is safe to share $\mathcal{V}(w)$, then it is safe to execute $\mathcal{E}(w)$. A direct corollary of the FTLR is a *universal contract* [25] that applies to arbitrary programs:

$$\vdash \left\{ \mathop{\Large *}_{r \in \mathsf{RegName}} \exists w.\, r \Mapsto w * \mathcal{V}(w) \right\} \rightsquigarrow \bullet \qquad \text{(CeriseUniversalContract)}$$

The contract says the program executes safely if the register file contains only safe-to-share words. Fig. 5 summarizes the Cerise logical relation[3]. The value relation captures the intuitive idea that capabilities that are safe to share give transitive access to integers, other capabilities that are also safe to share, and/or code that is safe to execute. The expression relation says a word is safe to execute if, when the machine executes from that word, it preserves the Iris invariants established in the program logic, as long as the rest of the register file contains only safe-to-share words. Preservation of invariants is implicitly required by $\{-\} \rightsquigarrow \bullet$.

[3]We use a slightly different definition for the RW/RWX cases from the one in Georges et al. [14, Fig. 9] to compare the Amarena logical relation in Fig. 11 with the Cerise logical relation more clearly. Lemma `write_allowed_inv` proves that these two definitions imply each other [17].

Safe-to-share is defined by case splitting. Integers and opaque (O) capabilities are always safe to share, since the adversary can use neither to gain privileges. Read-only (RO) and read-execute (RX) capabilities are safe if every word loaded from their regions is safe to share. Read-write (RW) and read-write-execute (RWX) capabilities further require that only safe-to-share words be written to their regions. Safety of a memory capability such as $(\mathsf{RWX}, \mathsf{Global}, b, e, -)$ is expressed using one Iris invariant per address in its range $[b, e)$. Each invariant takes the form $\boxed{\exists\, w.\, a \mapsto w * P(w)}$, where P describes what the word stored at a must satisfy. For read access, $P(w)$ implies that w is safe to share; for write access, safe-to-share words must satisfy P, so the invariant can be re-established after a write. This is what makes the Cerise relation simple: the safety of an entire region reduces to ordinary points-to assertions and invariants, the everyday machinery of separation logic.

The primary and most interesting use of the logical relation is proving that a sentry (E) capability is safe to share, meaning that it is safe to invoke as long as the register file contains safe-to-share words. An adversary can invoke the sentry at any time, so safety of the register file alone must provide every resource and guarantee needed to step through the sentry's code.

For our purposes, the later modality $\triangleright$ makes this seemingly recursive definition of $\mathcal{V}$ non-circular, and $\square$ makes capability safety duplicable and, crucially, permanent. Permanence is fitting because access in the Cerise model is never revoked. But the calling convention we turn to next is built on local capabilities, which Fig. 5 does not cover. Once a stack region is lent to a callee and later reclaimed, its contents change after the invariants over it have been established.

2.4 A Stack-Based Calling Convention Using Local Capabilities

Prior work [15, 43] verifies a stack-based calling convention scall using local capabilities, first on paper and later in Iris. The key idea of scall is to pass only two capabilities, both local, when a caller transfers control to a callee:

- a return capability $(\mathsf{E}, \mathsf{Local}, b, a, \cdots)$, where the range $[b, a)$ contains the caller's stack frame
- a stack capability $(\mathsf{RWLX}, \mathsf{Local}, a, e, a)$, where $[a, e)$ is the unused part of the stack above the caller's frame and is cleared before the callee gains control

As a sentry, the return capability hides the caller's frame from the callee. A jump to it produces a local read-execute capability $(\mathsf{RX}, \mathsf{Local}, b, a, -)$ and resumes execution in the caller's activation record, which restores the caller's private state to registers.

The stack capability gives the callee write-local access to $[a, e)$, letting it push a frame containing stack and return capabilities. This allows the callee to make further calls and callbacks (e.g., f in VAE can call back into the closure). Capabilities can be copied freely, so the calling convention must account for every copy an untrusted component may keep. Since local capabilities can be stored only in stack memory through a stack capability, any copies an adversary hides are confined to the stack region temporarily accessible to it. The convention clears this region before passing control to the callee, so the callee cannot retrieve these capabilities.

For scall to revoke access reliably, the initial machine state must satisfy the assumption that the untrusted callee does not already have transitive access to the stack region scall provides [43, § 3]. In § 4.2, we return to the proof burden imposed by this initial state assumption and explain how our logical relation discharges it.

2.5 The Very Awkward Example on Capability Machines

Fig. 6 gives a faithful Cerise translation of VAE and maps each program block to the command it implements. It closely follows the prior work's scall translation of the same program [15]. We walk through the blocks of Fig. 6 and highlight the capabilities shared among the closure, the adversary,

let x = ref(0) in

```
1 malloc off_malloc 1
2 store  r1 0
3 mov    r2 r1
4 mov    r1 pc
5 lea    r1 off_line9
6 crtcls off_malloc
7 rclear RegName \ {pc, r0, r1}
8 jmp    r0
```

$\lambda f.\, x \leftarrow 0;\ f();$

```
9  reqloc    r7 Global
10 prepstk
11 reqsizege r_stk 10
12 store     r_env 0
13 mov       r8 r0
14 mov       r9 r7
15 scall     [r8; r9; r_env] []
```

$x \leftarrow 1;\ f();$

```
16 sretepl [r0; r7; r_env]
17 store   r_env 1
18 mov     r8 r0
19 scall   [r8; r_env] []
```

assert($!x = 1$)

```
20 sretepl [r0; r_env]
21 load    r4 r_env
22 mov     r5 1
23 assert  off_assert
24 mclear  r_stk
25 rclear  RegName \ {pc, r0}
26 jmp     r0
```

Fig. 6. Code layout for the Cerise translation of VAE. The highlighted instructions mark places where the later proof has to account for changes to private heap state and revoked stack regions.

and the function argument f. Fig. 7 marks these capabilities with the bullets ❓. They appear black ❓ in our explanation to indicate that the discussion of their safety is deferred to § 3.

Lines 1–8: The machine is initialized, and the closure's local variable x is allocated on the heap, reachable only through a global capability the closure keeps. The closure is then created with the closure-creating macro `crtcls`, stored in a heap region, and exposed as a global sentry capability.

Lines 9–11: The adversary invokes the closure with argument f, a stack capability ⓐ, and a return capability ⓿ to which the closure returns after finishing execution. Macro `reqloc` checks if f is a global capability; macro `prepstk` checks that ⓐ is local and has permission RWLX and then makes ⓐ point to the start of its range; macro `reqsizege` checks that ⓐ provides enough space to hold the closure's own stack frame.

Lines 12–15: After (re)setting x to 0, the closure invokes scall. The convention pushes the closure's stack frame at the bottom of the stack region of ⓐ and then transfers control to f by jumping to the register containing f. It leaves f with two capabilities in the register file: a stack capability ⓑ for the stack region above the closure's stack frame, and a return capability ❶ that resumes execution at the code that sets x to 1.

Lines 16–19: When f returns to ❶, the macro `sretepl` reinstates the closure's stack frame, after which the closure sets x to 1 and invokes scall again. The convention pushes a new stack frame for the closure, again at the bottom of the region of ⓐ. It then clears the stack region that f previously had access to, namely the region of ⓑ, so that the capabilities ⓑ and ❶ are no longer accessible to f. Finally, f receives two new capabilities: a stack capability ⓒ and a return capability ❷.

Lines 20–26: When f returns to ❷, the closure reads x and asserts that it is 1. Before returning to the adversary, the closure clears the entire stack region of ⓐ with `mclear` and almost all registers with `rclear`, preventing the capability to x from being leaked.

The calling convention scall provides f with only the intended stack and return capabilities. Since the return capabilities ❶ and ❷ are sentries and the stack capabilities ⓑ and ⓒ cover only the regions above the closure's private frames, f cannot read these frames or reach x (local state encapsulation). By the clearing argument in § 2.4, f also cannot pass these capabilities to the accomplice g in VAEFail. We sketch the proof of these guarantees using Amarena in § 4.5.

3 Motivation: Reasoning about Suspension

We use this VAE translation to illustrate two expressiveness needs that drove existing work [15, 18, 43] to a Kripke logical relation, which base Cerise [14] leaves out for simplicity. The first is to express invariants and safety assertions whose truth changes as the adversary's access is suspended and later restored. The second is to state an invariant that lets the local variable x change back and forth between 0 and 1 while ruling out $x = 0$ at the assert.

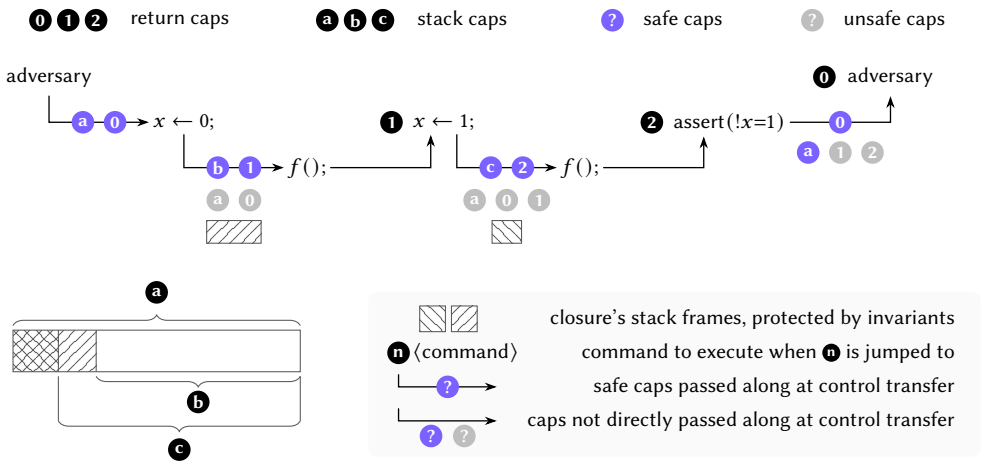


Fig. 7. How the safety of the stack and return capabilities evolves across the control transfers in the Cerise translation of VAE. In the stack layout (bottom left), the adversary's a spans the whole region, while the closure's two overlapping frames occupy its lower part. b and c govern the free regions above them.

Temporary invariants and capability safety. Proving the Cerise VAE requires tracking the capabilities shared among the adversary, the closure, and the argument f. Their safety changes as control is transferred between the three (Fig. 7). When the adversary invokes the closure, the stack a and return 0 capabilities are safe to share, but both become unsafe once f gains control. Sharing a would give f access to the closure's stack frame, and sharing 0 would let f skip the rest of the closure and return to the adversary, breaking well-bracketedness. The pattern repeats inside the closure. The return capability 1 is safe only for the first call to f. Once the second call provides f with 2, any retained copy of 1 would allow f to jump back to a stale return point, as g does with a retained 4 in VAEFail. The revocation of a and 0 is temporary, however. After the closure clears the stack and registers, the capability to x can no longer leak. By then, every call made during the closure's execution has returned to its immediate caller. Both a and 0 therefore become safe again.

This is where the Cerise logical relation breaks down. Once a capability is safe, its safety assertion is persistent. But the safety of VAE's return and stack capabilities keeps changing. Moreover, safety of a memory capability such as a depends on per-address invariants $\boxed{\exists w.\, a \mapsto w * \mathcal{V}(w)}$ asserting that its range stores only safe words. When the calling convention scall pushes the closure's frames into a's region, these invariants must be replaced for two reasons. The closure's stack frames store the capability to x, which is unsafe to share with f, so the "only safe words" guarantee no longer holds. At the same time, the proof must record the frame's exact layout in a new invariant $\boxed{a \mapsto w_1 * a + 1 \mapsto w_2 * \cdots}$, so we can later verify the code that restores the closure's local state. The relation cannot replace one invariant with another over the same memory region.

Well-bracketed control flow. The Cerise logical relation concerns only addresses shared between trusted and untrusted components, but the local variable x lives at a private heap address. Reasoning about x therefore falls to the program logic. The usual device is to put the memory resource $x \mapsto -$ in an invariant that constrains how x evolves across calls to unknown code. The invariant acts as a proxy for well-bracketedness. Since f cannot access x directly, the invariant should become impossible to re-establish exactly when control reaches the assert command while

x is still 0. Fixing x to a specific value $\boxed{x \mapsto n}$ is too strong, since the value of x changes between 0 and 1. In a high-level language, Timany et al. [49, § 2] take a more advanced option and prove that f preserves an invariant that exploits the Iris one-shot ghost state:

$$\boxed{\mathsf{pending}\ \gamma * x \mapsto 0 \lor \mathsf{shot}\ \gamma * x \mapsto 1} \qquad \text{(AwkwardInv)}$$

The ghost resources in AwkwardInv allow the invariant to be re-established after x changes from 0 to 1, while ruling out the case where $x = 0$ when the closure returns. However, AwkwardInv is still too restrictive. It enforces an irreversible transition from 0 to 1. A later invocation that resets x to 0 cannot satisfy either branch of the invariant because shot γ cannot revert to pending γ.

4 Key Ideas of Amarena

To meet the two expressiveness needs in § 3, we adapt logical memory [24] to express temporary invariants and capability safety assertions, and ghost stacks [49] to reason about well-bracketed control flow.

4.1 Logical Memory

Hur et al. [24] first proposed the *garbage-collected separation logic* (GCSL) for reasoning about assertions over regions that are reclaimed through garbage-collection. The key idea of GCSL is its use of logical memory, where a physical address can have several logical versions, each storing its own word, as in Fig. 8a. For example, k_v stores the integer 4 and k_{v+1} a capability. A write to one version cannot break invariants or safety assertions over another. That closes the first gap of § 3.

Capabilities in the logic carry versions, restricting memory access to a specified version of an address. Memory assertions describe logical addresses such as a_v instead of the physical address a directly. For instance, we reason with versioned memory points-to assertions $a_v \mapsto 1$ instead of $a \mapsto 1$. The running code, however, remains version-insensitive. Even memory instructions such as `store` $r_{\mathtt{dst}}$ $r_{\mathtt{src}}$ refer only to registers.

Sharing now has a more precise meaning. Two components share an address only when their capabilities refer to the same version of the address. Revocation is modeled by logically reclaiming the shared address through a fancy update that picks a fresh version and allocates a new memory points-to assertion over that version to the trusted code. This is a ghost operation that does not affect the running machine.

Logical reclamation gives us a way to state invariants over revocable stack regions. We enforce a correspondence invariant[4] across execution steps: each address has at most one active version (framed in purple in Fig. 8a), and the logical and physical machines agree on registers and on memory at those versions. For example, the capability stored at k_{v+1} points to version $v - 1$ of the region $[b, e)$. After erasing its version, it is exactly the capability stored at k. Memory accesses in our logic must also go through active versions. Because correspondence covers only active memory regions, the previous version accessible to the adversary and any invariant over it remain permanently valid. In a setting without logical memory, reusing the region could invalidate those invariants. The inactive version simply no longer reflects the physical machine state.

When an address is logically reclaimed, the fresh version becomes active, and reasoning about trusted code proceeds using resources for that version. This keeps logical and physical execution aligned. Compared to GCSL, Amarena uses active versions not only for this correspondence, but also to define when stack and return capabilities are safe. The point is to let a change of active version make a previously safe stack or return capability unsafe. Our setting also needs this change to be reversible. Revocation through the calling convention scall is temporary, and a well-bracketed

[4] An invariant in the ordinary sense, not an Iris invariant

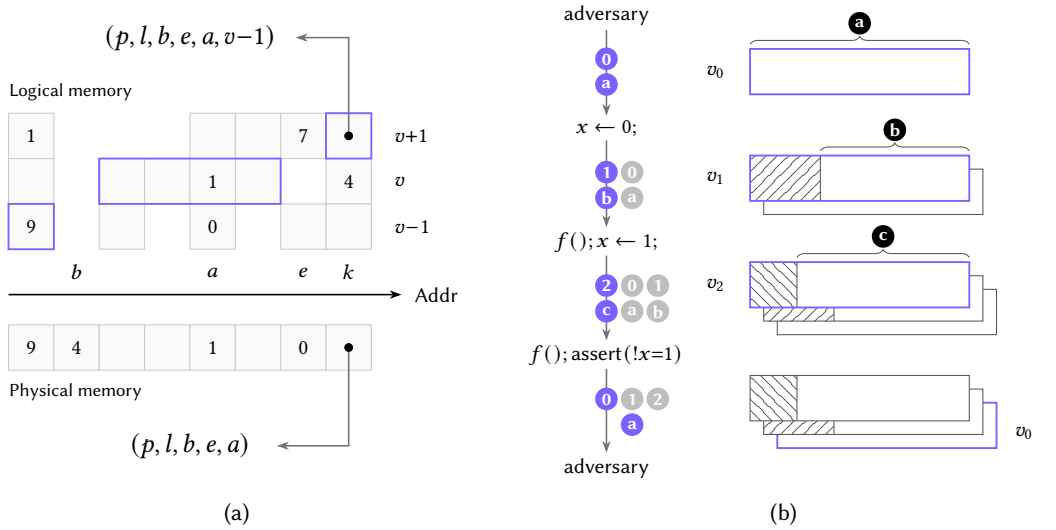


Fig. 8. Sketches of the central idea, logical memory. (a) An example logical memory (above the axis), whose active regions correspond to the physical memory regions (below the axis). (b) An example of logical reclamation and restoration in our proof of VAE. The active version of the stack region governed by ⓐ changes from v_0 to v_1 to v_2, then back to v_0. Safety $\mathcal{V}$ of capabilities depends on active versions, so the capabilities in the figure become safe or unsafe as the active version changes. See Fig. 7 for the capability legend.

return reinstates the caller's access. As Fig. 7 shows, capabilities such as ⓐ and ⓪ follow this pattern. They become unsafe during the call and safe again when the closure returns. Logical reclamation can only turn a safe capability unsafe, not the reverse, so our setting needs more than GCSL provides. We add a second ghost operation, *logical memory restoration*, which switches the active version of a region back to a previously active one.

To make these ghost operations concrete, Fig. 8b shows how they are used in VAE. The adversary provides a safe return capability ⓪ and a stack capability to version v_0 when invoking the closure. Before calling f, the closure reclaims the stack, records its frame in an invariant over the fresh version v_1, and gives f access to v_1. This makes capabilities ⓪ and ⓐ unsafe. The second call to f repeats the pattern with another fresh version, v_2, making ⓑ and ❶ unsafe as well. When the closure is ready to return to the adversary, we restore the stack's active version to v_0, making the return capability ⓪ safe to jump to again.

4.2 Challenges: From Garbage Collection to Reversible Revocation

Logical reclamation and restoration give the right high-level story, but they leave soundness issues. The correspondence invariant and our requirement that accesses go through only active versions do not make reads and writes sound by themselves. In Fig. 8a, the active address k_{v+1} stores a capability to the inactive address a_{v-1}. If a program loads the capability and writes through it, the physical cell a changes while the active logical address a_v does not, breaking correspondence.

GCSL rules this out using a global reachability invariant that requires every capability reachable from registers to point to an active region. Fig. 8a violates this requirement, since k_{v+1} holds a capability into the inactive version a_{v-1}. The invariant is natural and necessary for garbage collection, where reclamation is justified precisely by unreachability from all roots.

IrvkDupl
$a \dashrightarrow\!\square\ v \vdash a \dashrightarrow\!\square\ v * a \dashrightarrow\!\square\ v$

IrvkSame
$a \dashrightarrow\!\square\ v_1 * a \dashrightarrow\!\square\ v_2 \vdash \ulcorner v_1 = v_2 \urcorner$

Restore
$a_{v_1} \mapsto lw_1 * a_{v_2} \mapsto lw_2 * a \dashrightarrow v_2$
$\Rrightarrow\!\!*_I\ a_{v_1} \mapsto lw_2 * a_{v_2} \mapsto lw_2 * a \dashrightarrow v_1$

RestoreInv
$\boxed{\exists\, lw.\, a_{v_1} \mapsto lw * P(lw)} *$
$a_{v_2} \mapsto lw * P(lw) * a \dashrightarrow v_2$
$\Rrightarrow\!\!*_I\ a_{v_2} \mapsto lw * a \dashrightarrow v_1$

Revoke
$a_v \mapsto lw * a \dashrightarrow v$
$\Rrightarrow\!\!*_I\ a_v \mapsto lw * \exists\, v_f.\, a_{v_f} \mapsto lw * a \dashrightarrow v_f$

RvkIrvkFalse
$a \dashrightarrow v_1 * a \dashrightarrow\!\square\ v_2 \vdash \ulcorner \mathsf{False} \urcorner$

Fig. 9. Lemmas for version points-to assertions[5]. Some resources are omitted for readability.

Amarena instead needs a different kind of unreachability, one that is reversible and component-local: a revoked capability may remain reachable by trusted code but not by the component whose authority has been revoked. In Fig. 7, the stack capability ⓐ is stored in the closure's frame below ⓑ, out of reach of f. Since f is supplied by the adversary, ⓐ can be treated as unreachable by the adversary while the closure runs, and reachable again when the closure returns.

The two settings also differ in what enforces unreachability. In GCSL, the garbage collector, a runtime mechanism, maintains it globally. In Amarena, scall establishes component-local unreachability only partially. Its check of f's locality works only under the assumptions about the initial machine state explained in § 2.4. Those assumptions are necessary because nothing at runtime inspects what f can already reach or forces f to give up access overlapping ⓐ's region. We therefore need a clean, modular, and footprint-local way to turn "f *is a global capability*" into a semantic guarantee "f has no access to stack capabilities or to ⓐ's region before it is called, and hence after scall, the revoked stack and return capabilities are truly inaccessible to f".

4.3 Version Points-to Assertions

Our solution is to introduce version points-to assertions $a \dashrightarrow_q v$ and, compared to GCSL, a very different proof scheme. We make active-version bookkeeping partially explicit with these new assertions. A version points-to assertion $a \dashrightarrow_q v$ is analogous to a fractional memory points-to assertion. It states that the active version of address a is v. We use only two forms. The exclusively owned $a \dashrightarrow v$ gives write access, so we can change the active version of a. The persistent $a \dashrightarrow\!\square\ v$ fixes the active version of a to v permanently. Intuitively, the former covers revocable stack regions and the latter covers irrevocable heap regions.

Version points-to assertions play three roles: they justify component-local unreachability in our logical relation, make memory reads and writes sound, and allow the user to logically reclaim and restore stack regions.

Justifying component-local unreachability. The logical relation is the natural place to turn the syntactic fact that f is global into semantic reachability facts. We extend safe-to-share $\mathcal{V}$ in a way that allows us to infer three things: 1) a safe stack capability must be local; 2) a safe global capability gives transitive access only to integers or other safe global capabilities; 3) every address transitively accessible to a safe global capability carries a persistent resource $a \dashrightarrow\!\square\ v$, fixing its version, whereas every address in a safe stack capability's range carries an exclusive resource $a \dashrightarrow v$, making it reclaimable.

[5]Here $P \Rrightarrow\!\!*_I\ Q \triangleq \forall\ reg, mem.\ P * I(reg, mem) \Rrightarrow\!\!* Q * I(reg, mem)$. This connective, along with the soundness of Revoke, Restore, and RestoreInv, is discussed in more detail in Appendix A.

These extra requirements rule out the problematic scenarios about f's access before it is called. A global stack capability is never safe to share, since the adversary could stash it in its heap and keep a private stack that scall can never clear. Requiring safe global capabilities to give access only to integers or other global capabilities is therefore exactly what implies that they do not have a private stack. By RvkIrvkFalse in Fig. 9, which says that the active version of an address cannot be both fixed or changeable, no safe global capability's access range can overlap that of a safe stack capability. The extended safe-to-share $\mathcal{V}$ discussed in § 6 then accounts for the component-local unreachability that the machine does not check dynamically.

Soundness of memory reads and writes. A direct adaptation of GCSL to our setting would use the following proof scheme:

$$\underbrace{\text{correspondence} + \text{reachability}}_{\text{internal to the logic}} + \underbrace{\text{logical memory reclamation}}_{\text{ghost operation by the user}} \Rightarrow \text{GC}$$

In this adaptation, the user reasons about memory accesses using register and memory points-to assertions and performs logical reclamation for memory that has been physically garbage-collected, while active-version bookkeeping is hidden inside the logic. Since registers are the garbage-collection roots, the reachability invariant of § 4.2 keeps both memory accesses and logical reclamation sound implicitly. Amarena uses a different division of labor:

$$\underbrace{\text{correspondence}}_{\text{internal to the logic}} + \underbrace{\text{version points-to} + \text{logical memory reclamation}}_{\text{ghost operation by the user}} \Rightarrow \text{calling convention}$$
$$\underbrace{\text{correspondence}}_{\text{internal to the logic}} + \underbrace{\text{version points-to} + \text{logical memory restoration}}_{\text{ghost operation by the user}} \Rightarrow \text{well-bracketed return}$$

Informally, the shift is from eager to lazy enforcement of active-version checks. Amarena delays each check until it matters for a concrete memory access. The global bookkeeping for active versions remains internal to the logic, much like the authoritative state of memory and registers in separation logic. The user does not manipulate this global state directly. Instead, they manage a small, footprint-local bundle of version points-to assertions. Program-logic rules for reads and writes now require not only the usual register and memory resources, such as $r \Mapsto (\mathrm{RW}, l, b, e, a, v)$ and $a_v \mapsto lw$, but also the version resource $a \dashrightarrow_q v$, which witnesses that the accessed logical address is indeed active.

Concretely, this laziness plays out in three ways. A register may contain a capability to an inactive version. For example, assertions $r \Mapsto (\mathrm{RW}, l, b, e, a, 0)$ and $a \dashrightarrow 1$ can hold at the same time. Active-version checks are needed only for instruction fetch and memory instructions, not for register-only instructions. When a memory access does occur, the check is footprint-local. Accessing a_v through $(\mathrm{RW}, l, b, e, a, v)$ requires the assertion $a \dashrightarrow_q v$, but not version resources for the rest of $[b, e)$. This suffices because the instruction accesses only the address a.

Logical reclamation and restoration. Version points-to assertions and the two ghost operations provide the user-facing interface for the active-version changes needed in our setting, namely reclamation at scall, the point where revocation happens, and restoration at well-bracketed returns. The main rules for these operations are shown in Fig. 9.

Revoke logically reclaims a memory address. If we own the memory resource $a_v \mapsto lw$ and the exclusive version resource $a \dashrightarrow v$, then we can switch the active version of a to some fresh version v_f and obtain a fresh memory resource $a_{v_f} \mapsto lw$ storing the same word lw. The resource $a_v \mapsto lw$ survives, so an invariant over the version v can remain valid while trusted code continues with the new resources $a_{v_f} \mapsto lw$ and $a \dashrightarrow v_f$.

Restore performs logical memory restoration. Suppose version v_2 is active and we want to switch to a previous version v_1. The rule requires ownership of the memory resources for both a_{v_1}

and a_{v_2}, together with the exclusive version resource $a \dashrightarrow v_2$. It makes v_1 active and overwrites a_{v_1} with the word from a_{v_2}. The reactivated version again stores the same word as the physical address.

Allowing a switch back to a previously active version raises a question: Can we switch versions at will, making arbitrary capabilities safe or unsafe? The answer is no. Overwriting a_{v_1} with the word lw_2 held by the active version a_{v_2} requires write access to a_{v_1}. When the memory is shared and protected by an invariant, restoration succeeds only if lw_2 satisfies that invariant (RestoreInv, explained in Appendix A). If the invariant is a safety invariant, the switch to the previous version v_1 succeeds only after lw_2 is proved safe.

4.4 Lighter Variant of Ghost Stacks

A plain Iris invariant cannot directly express the reversible evolution of the closure's local variable x in VAE. To capture this evolution and reason about well-bracketedness in high-level languages, Timany et al. [49] propose *ghost stacks* as a simpler alternative to the Kripke logical relation. We use a lighter variant of this machinery, adapted to our capability machine model and its use of return capabilities; § 8 compares it with their full ghost stacks.

The idea is to track a ghost stack, which is a list of ghost names γs, in the definition of safe-to-execute $\mathcal{E}$ through $\bullet\, \gamma s$ and inside the invariant VAEInv through $\circ\, \gamma s$. These assertions are formally defined in § 6. They enjoy the following two rules:

$$\bullet\, \gamma s * \circ\, \gamma s' \vdash \ulcorner \gamma s = \gamma s' \urcorner \quad \text{(GSAgree)} \qquad \bullet\, \gamma s * \circ\, \gamma s \Rrightarrow * \bullet\, \gamma s' * \circ\, \gamma s' \quad \text{(GSUpd)}$$

In VAE, we reason about the local variable x using the invariant

$$\boxed{\exists\, \gamma, \gamma s'.\ \circ\, \gamma :: \gamma s' * (\mathsf{pending}\ \gamma * x \mapsto 0 \vee \mathsf{shot}\ \gamma * x \mapsto 1)} \qquad \text{(VAEInv)}$$

It ties the topmost ghost name of the ghost stack to the variable x and connects the ghost resource associated with that topmost name to the value of x. New assertions of the form pending γ' can always be allocated for a fresh ghost name γ'. We allocate a fresh pending γ', push γ' to the ghost stack using GSUpd, and let VAEInv use γ' for the new $x = 0$ to $x = 1$ transition. Pushing γ' as the topmost name releases the ghost resource pending γ from VAEInv. Advancing pending γ to shot γ and holding on to it lets us pop γ' and then re-establish VAEInv for γ with $x = 1$ at the closure's well-bracketed return.

4.5 Putting the Ideas Together on VAE

We sketch the proof of the Cerise VAE, presenting the main steps as Hoare triples written in the style of the source-level VAE. Roughly, the specification is

$$\{\, [b_{\mathsf{malloc}}, e_{\mathsf{malloc}}) \dashrightarrow\!\square\ v_{\mathsf{malloc}} * \cdots \,\}\ \mathsf{VAE}\ \{\, g.\ \forall\, f.\ f \text{ is safe-to-share} \Rightarrow g\ f \text{ executes safely} \,\}$$

The pre-condition requires the version resources needed to step through the preamble of VAE. For example, the heap $[b_{\mathsf{malloc}}, e_{\mathsf{malloc}})$ storing the malloc subroutine is pinned to version v_{malloc}. The post-condition states that the returned function, which is a global sentry capability for the closure body, executes safely for any function argument f, itself a safe global capability in our case. Unfolding "executes safely" gives us the triple

$$\begin{aligned} &\{\, \mathsf{VAEInv} * \bullet\, \gamma :: \gamma s * [b, e) \dashrightarrow v_0 * \cdots \,\} \\ &\quad x \leftarrow 0;\ f();\ x \leftarrow 1;\ f();\ \mathsf{assert}(!x = 1) \\ &\{\, _.\ \bullet\, \gamma :: \gamma s * [b, e) \dashrightarrow v_0 * \cdots \,\} \end{aligned}$$

Suppose the stack capability ⓐ that the closure receives when invoked gives access to the memory range $[b, e)$. The triple states that the closure executes in a well-bracketed manner, reflected by VAEInv and $\bullet\, \gamma :: \gamma s$ in the pre- and post-conditions. Its pre-condition provides $[b, e) \dashrightarrow v_0$, which

$$
\begin{array}{lll}
v \in \mathsf{Version} & \triangleq \mathbb{N} & \\
la \in \mathsf{LAddr} & \triangleq \mathsf{Addr} \times \mathsf{Version} & \\
lc \in \mathsf{LCap} & \triangleq \{(p, l, b, e, a, v) \mid b, e, a \in \mathsf{Addr}\} & \\
lw \in \mathsf{LWord} & \triangleq \mathbb{Z} + \mathsf{LCap} & \\
vm \in \mathsf{VMap} & \triangleq \mathsf{Addr} \rightharpoonup_{\mathrm{fin}} \mathsf{Version} & \\
lreg \in \mathsf{LReg} & \triangleq \mathsf{RegName} \rightharpoonup_{\mathrm{fin}} \mathsf{LWord} & \\
lmem \in \mathsf{LMem} & \triangleq \mathsf{LAddr} \rightharpoonup_{\mathrm{fin}} \mathsf{LWord} & \\
lws \in \mathsf{list}\ \mathsf{LWord} &
\end{array}
$$

Fig. 10. Logical machine syntax. Each L-prefixed construct is the versioned counterpart of the like-named physical construct in Fig. 2; purple marks everything Amarena adds.

lets us logically reclaim $[b, e)$ using Revoke and restore it when the closure returns (see Fig. 8b). After the closure sets x to 0, Amarena supports intermediate triples such as

$$
\begin{array}{l}
\{\, \mathsf{VAEInv} * \bullet \gamma' :: \gamma :: \gamma s * \mathsf{shot}\ \gamma * [b, e) \dashrightarrow v_1 * [b, e)_{v_1} \mapsto - * \cdots \,\} \\
\quad f(); \ x \leftarrow 1; \ f(); \ \mathsf{assert}(!x = 1) \\
\{\, _. \bullet \gamma :: \gamma s * [b, e) \dashrightarrow v_0 * \cdots \,\}
\end{array}
$$

Implicit in this pre-condition is that the stack ⓐ and return ⓪ capabilities are now unsafe, since $[b, e)$ has been logically reclaimed. The final step of proving VAE is

$$
\{\, \mathsf{VAEInv} * \bullet \gamma :: \gamma s * \mathsf{shot}\ \gamma * [b, e) \dashrightarrow v_2 * \cdots \,\}\ \mathsf{assert}(!x = 1)\ \{\, _. \bullet \gamma :: \gamma s * [b, e) \dashrightarrow v_0 * \cdots \,\}
$$

where we apply Restore to $[b, e) \dashrightarrow v_2$ after the assert succeeds, making ⓐ and ⓪ safe again. Logical reclamation, restoration, and ghost-stack tracking complete the proof of the Cerise VAE.

5 The Amarena Program Logic

Amarena extends the Cerise program logic (§ 2.2) with a logical machine. The programs being verified still run on the version-insensitive physical machine. The previous section gave the intuition behind logical memory, version points-to assertions, and the two ghost operations. We now give their formal definitions and provide the soundness argument in Appendix A.

Fig. 10 defines the logical machine by adding versions to the constructs of Fig. 2. A logical memory is a finite map from logical addresses, which are physical addresses indexed by versions, to logical words. Versions are natural numbers used only for bookkeeping. Their order carries no meaning. A fresh version is always available because logical memory is finite. A logical word is either an integer or a logical capability (p, l, b, e, a, v), which grants permission p and locality l over version v of $[b, e)$. Logical registers carry no versions. Only the capabilities they hold point to logical addresses.

Fig. 3 shows Amarena's extensions to the Cerise program logic syntax in purple. Memory points-to assertions are versioned. The assertion $a_v \mapsto lw$ owns logical address a_v storing lw, and $[b, e)_v \mapsto lws$ owns version v of $[b, e)$. This ownership gives read-write access. Register points-to assertions are not versioned: $r \Mapsto lw$ says register r stores logical word lw. The version resource $a \dashrightarrow_q v$ satisfies the rules in Fig. 9. IrvkDupl duplicates the witness that an address is irrevocable. IrvkSame states that an irrevocable address is fixed to exactly one version. RvkIrvkFalse rules out an address being both revocable and irrevocable.

Amarena's instruction rules add version points-to assertions to Cerise's pre- and post-conditions, as shown by `store` and `jmp` in Fig. 4. A rule carries version resources only for the addresses its instruction accesses. Since instructions are fetched through the program counter, every rule includes $a_{\mathrm{pc}} \dashrightarrow_{q_1} v_{\mathrm{pc}}$ for the address being executed. A memory instruction additionally requires a version witness $a \dashrightarrow_{q_2} v$ for the logical address a_v it accesses. Register-only instructions, such as `jmp`, do not require further version resources. These version resources are either exclusive or persistent, indicated by the fractions q_1, q_2. Each post-condition returns the same version resources required by its pre-condition.

$$\mathcal{E}_{vm,ys}(lw) \triangleq \forall\, lreg.\ \left\{\mathrm{pc} \mapsto lw * \left(\mathop{\Large *}_{(r,lw_r)\in lreg,\, r\neq \mathrm{pc}} r \mapsto lw_r * \mathcal{V}_{vm,ys}(lw_r)\right) * \left(\mathop{\Large *}_{(a,v)\in vm} a \dashrightarrow v\right) * \bullet ys\right\} \rightsquigarrow \bullet$$

$$\mathcal{V}_{vm,ys}(lw) \triangleq \begin{cases}
\mathcal{V}_{vm,ys}(z) & \triangleq \mathrm{True} \quad \text{for } z \in \mathbb{Z} \\
\mathcal{V}_{vm,ys}(\mathrm{O},-,-,-,-,-) & \triangleq \mathrm{True} \\
\mathcal{V}_{vm,ys}(\mathrm{E},\mathrm{Global},b,e,a,v) & \triangleq \triangleright\,\square\,(\forall\, vm', ys'.\ \mathcal{E}_{vm',ys'}(\mathrm{RX},\mathrm{Global},b,e,a,v)) \\
 & \quad * \left(\mathop{\Large *}_{a\in[b,e)} \text{irrevocable}\ (a,v)\right) \\
\mathcal{V}_{vm,ys}(\mathrm{E},\mathrm{Local},b,e,a,v) & \triangleq \triangleright\,\square\,\mathcal{E}_{vm,ys}(\mathrm{RX},\mathrm{Local},b,e,a,v) * \left(\mathop{\Large *}_{a\in[b,e)} \text{active}\ (a,v,vm)\right) \\
\mathcal{V}_{vm,ys}(\mathrm{RO/RX},l,b,e,-,v) & \triangleq \mathop{\Large *}_{a\in[b,e)} \exists P.\ \boxed{\exists\, lw.\ a_v \mapsto lw * P(lw)} * \text{read-safe}\ (a,P,l) \\
 & \quad * \begin{cases} \text{irrevocable}\ (a,v) & l = \mathrm{Global} \\ \text{active}\ (a,v,vm) & l = \mathrm{Local} \end{cases} \\
\mathcal{V}_{vm,ys}(\mathrm{RW/RWX},l,b,e,-,v) & \triangleq \mathop{\Large *}_{a\in[b,e)} \exists P.\ \boxed{\exists\, lw.\ a_v \mapsto lw * P(lw)} * \text{read-safe}\ (a,P,l) \\
 & \quad * \text{write-safe}\ (a,P,\mathrm{RW/RWX}) \\
 & \quad * \begin{cases} \text{irrevocable}\ (a,v) & l = \mathrm{Global} \\ \text{active}\ (a,v,vm) & l = \mathrm{Local} \end{cases} \\
\mathcal{V}_{vm,ys}(\mathrm{RWL/RWLX},l,b,e,-,v) & \triangleq \mathop{\Large *}_{a\in[b,e)} \exists P.\ \boxed{\exists\, lw.\ a_v \mapsto lw * P(lw)} * \text{read-safe}\ (a,P,\mathrm{Local}) \\
 & \quad * \text{write-safe}\ (a,P,\mathrm{RWL/RWLX}) \\
 & \quad * \ulcorner l = \mathrm{Local} \urcorner * \text{revocable}\ (a,v,vm)
\end{cases}$$

where

$$ys \in \mathrm{GhostStack} \triangleq \text{list gname} \qquad \bullet ys \triangleq \boxed{\bullet ys}^{ygs} \qquad \circ ys \triangleq \boxed{\circ ys}^{ygs}$$

$$vm \in \mathrm{VMap} \qquad \text{revocable}\ (a,v,vm) \triangleq \ulcorner (a,v) \in vm \urcorner \qquad \text{irrevocable}\ (a,v) \triangleq a \dashrightarrow\!\square\ v$$

$$\text{active}\ (a,v,vm) \triangleq \text{revocable}\ (a,v,vm) \vee \text{irrevocable}\ (a,v)$$

$$\text{read-safe}\ (a,P,l) \triangleq \begin{cases} \triangleright\,\square\,(\forall\, lw.\ P(lw) \mathrel{-\!\!*} \mathcal{V}_{vm,ys}(lw) * \ulcorner \neg\, \mathrm{Local}(lw) \urcorner) & l = \mathrm{Global} \\ \triangleright\,\square\,(\forall\, lw.\ P(lw) \mathrel{-\!\!*} \mathcal{V}_{vm,ys}(lw)) & l = \mathrm{Local} \end{cases}$$

$$\text{write-safe}\ (a,P,p) \triangleq \begin{cases} \triangleright\,\square\,(\forall\, lw.\ \mathcal{V}_{vm,ys}(lw) * \ulcorner \neg\, \mathrm{Local}(lw) \urcorner \mathrel{-\!\!*} P(lw)) & p = \mathrm{RW/RWX} \\ \triangleright\,\square\,(\forall\, lw.\ \mathcal{V}_{vm,ys}(lw) \mathrel{-\!\!*} P(lw)) & p = \mathrm{RWL/RWLX} \end{cases}$$

Fig. 11. Our logical relation. Highlights mark the additions to the Cerise relation (Fig. 5) related to the revocation mechanism, logical memory, and ghost stacks respectively.

6 The Amarena Logical Relation

Fig. 11 gives the complete Amarena logical relation and highlights its extensions to the Cerise logical relation (Fig. 5). Table 1 summarizes the facts each extension provides. The rest of this section shows how those extensions formalize the ideas of § 4 for reasoning about revocation of untrusted code's access.

Recall from § 2.3 that the primary use of the logical relation is proving sentry capabilities safe to share, meaning that they are safe to invoke whenever the register file contains only safe-to-share words. In Cerise, safety of the register file alone must supply every resource and guarantee needed to step through the sentry's code. In Amarena, these resources come from a safe register file together with assertions about the active versions of revocable stack regions and the ghost stack.

Indexed safety. The active versions of a program's footprint and the ghost stack both change as the program runs. Safe-to-share $\mathcal{V}$ and safe-to-execute $\mathcal{E}$ are therefore indexed by a version map vm and a ghost stack ys. Safe execution now owns the version resources tracked by vm and the ghost stack assertion $\bullet ys$ in its pre-condition, both defined in Fig. 11. For $\bullet ys$ it suffices to know it satisfies GSAgree and GSUpd. The FTLR is similar to that of Cerise, except that both relations are

Table 1. Extensions to the Cerise logical relation and the facts about safe capabilities derived from the relation.

(a) Derivable from $\mathcal{V}_{vm,\gamma s}(-)$ alone.

	Extension	Derived facts
1	$\ulcorner l = \text{Local} \urcorner$ in $\mathcal{V}(\text{RWL/RWLX}, l, -)$	• RWL/RWLX caps are local; hence global caps are not write-local.
2	$\ulcorner \neg\text{Local}(lw) \urcorner$ in read-safe$(-, l)$ only if l = Global	• Regions accessible through global caps do not contain local or RWL/RWLX caps.
3	$\ulcorner \neg\text{Local}(lw) \urcorner$ in write-safe$(-, p)$ only if p = RW/RWX	• RW/RWX caps can be used to write only non-local safe words. • RWL/RWLX caps can be used to write any safe word.

(b) Derivable when proving $\mathcal{E}_{vm,\gamma s}(-)$.

	Extension	Derived facts
4	irrevocable	• Global caps give access to active regions. • Their access ranges have fixed versions and cannot be reclaimed.
5	irrevocable + $\forall\ vm', \gamma s'$	• Global caps remain safe under any version map and ghost stack.
6	revocable + active + $\ast_{(a,v)\in vm}\ a \dashrightarrow v$	• Local caps give access to active regions. • RWL/RWLX caps provide exclusive version resources for reclamation and restoration. • A local cap is safe only while its region is active, including after restoration.
7	$\bullet\, \gamma s$	• The ghost stack may change during execution but must be restored before a well-bracketed return.

indexed by the same version map and ghost stack: $\vdash \mathcal{V}_{vm,\gamma s}(lw) \mathrel{-\!\!*} \mathcal{E}_{vm,\gamma s}(lw)$. We then obtain the universal contract:

$$\vdash \left\{ \left(\ast_{r\in\text{RegName}} \exists\, lw.\ r \Mapsto lw * \mathcal{V}_{vm,\gamma s}(lw) \right) * \left(\ast_{(a,v)\in vm} a \dashrightarrow v \right) * \bullet\, \gamma s \right\} \rightsquigarrow \bullet \qquad \text{(AmarenaUC)}$$

The safety of ❼ depends on vm and γs, and so does any universal contract that relies on this safety. Applying such a contract at a well-bracketed return to the adversary therefore requires first restoring the active versions and ghost stack that ❼ was safe under.

Reclamation. Safety of a stack capability allows us to recover the version resources needed to logically reclaim its region. This is supported by the revocable condition in the RWL/RWLX case of safe-to-share. The index vm records the active versions of a subset of all revocable stack regions. When the register file contains a stack capability (RWLX, Local, $b, e, -, v$), revocable says that each address in $[b, e)$ appears in vm at version v. It also says that the assertion $[b, e) \dashrightarrow v$ can be extracted from the version resource bundle indexed by vm in $\mathcal{E}$ (see Reclaim in Appendix B).

No stack authority through global capabilities. Component-local unreachability follows from the calling convention scall and assumptions about the initial machine state, as explained in § 4.3. The logical relation turns these assumptions into the derived fact that a safe global capability carries no stack authority. The definition of safe-to-share constrains the locality of every word transitively accessible from the capability. A word read through a safe global capability is non-local, by the $\ulcorner \neg\, \text{Local}(lw) \urcorner$ requirement in read-safe. Meanwhile, a safe stack capability must be local, by the $\ulcorner l = \text{Local} \urcorner$ requirement in the RWL/RWLX case. Any capability read from a safe global capability is again safe and global, so a safe global capability does not give access to safe stack capabilities. RvkIrvkFalse further rules out the case where the region of a safe non-opaque

global capability overlaps that of a safe stack capability. The former is irrevocable, whereas the latter is revocable, as its version resources in $\mathcal{E}$ are exclusively owned (see GlobalStackDisj in Appendix B).

Global capabilities under any index. Safety of global capabilities is independent of changes to active versions and the ghost stack:

$$\ulcorner \neg \mathsf{Local}(lw) \urcorner * \mathcal{V}_{vm,\gamma s}(lw) \mathrel{-\!\!*} \mathcal{V}_{vm',\gamma s'}(lw) \qquad \text{(GlobalSafeAll)}$$

The irrevocable condition, together with the fact that safe global capabilities give access only to safe non-local words, ensures that the regions reachable from a safe global capability cannot be reclaimed. Reclaiming stack regions elsewhere therefore does not affect the invariants that global memory capabilities depend on. Any sentry reachable from a safe global capability is global, so it is safe under any version map and ghost stack by definition.

This index independence is needed because local and global sentries are used differently. Local sentries, such as the return capabilities created by scall, are meant to be invoked only during a limited part of a call. A global sentry, in contrast, can be stored in the heap and invoked at any later time. The goal of VAE is precisely to show that, no matter how deep the calls to the closure are nested, scall prevents the malicious behavior of a global capability f as long as f calls only the safe capabilities it can reach. If global sentries became unsafe after stack reclamation or ghost stack changes, f could not call the closure in a nested call, ruling out the very behavior VAE exists to stress test.

Active access evidence. To step through a sentry's code, each capability in the register file that can be used to fetch from, read from, or write to memory must give access only to active regions. The revocable and irrevocable conditions establish this evidence for safe stack and global capabilities, respectively. The remaining case consists of local, non-stack capabilities. They satisfy the active condition, which associates each address with an exclusively owned or persistent version points-to assertion. Every safe capability then provides one of these two resources for each address it can access (see ActiveAccess in Appendix B), since revocable and irrevocable imply active.

Consistent reads and writes. Finally, the derived facts about reads and writes must agree. The $\ulcorner \neg \mathsf{Local}(lw) \urcorner$ requirement in read-safe guarantees that a word read through a safe global capability is non-local. But it would be unsound if the same region could first be used to store a local word through a global RW/RWX capability. Therefore write-safe is extended with a $\ulcorner \neg \mathsf{Local}(lw) \urcorner$ requirement as well. Only safe stack capabilities can be used to write local words.

7 Evaluation

We compare Amarena with Cerise with KLR [15, 41] and base Cerise [14] to provide evidence for three strengths claimed in § 1: preservation of base Cerise's reasoning (§ 7.1), non-invasiveness and orthogonality (§ 7.2), and a simpler VAE proof in a smaller reusable development (§ 7.3). We give further arguments for easier framing and generality in § 8.

Since our main contribution is a proof technique, we combine qualitative arguments with comparisons of specifications and proofs for the same examples in Amarena and prior work [14, 15, 41, 43]. As in the literature on robust safety for capability machines and logical relations [2, 15, 18, 30, 35, 43, 45, 49], most examples we prove are small but representative. The larger or more demanding examples include the dynamic sealing and interval libraries [14] and VAE. All specification and proof lines are counted with `coqwc`, the standard Rocq line counter.

7.1 Preserving Base Cerise's Reasoning

We reprove all eight libraries and case studies from base Cerise in Amarena, with minor adaptations to the programs and the guarantees established by their adequacy theorems (e.g., adding locality checks). These programs do not involve temporary delegation, and their proofs do not invoke logical reclamation, restoration, or ghost stack lemmas, providing concrete evidence that base Cerise's reasoning carries over.

Amarena's program logic and logical relation remain close to those of base Cerise. Instruction rules require additional version bookkeeping but do not themselves involve logical reclamation or restoration, allowing us to step through known programs much as we would in base Cerise when revoking and restoring access do not break existing invariants. Safety invariants in Amarena's logical relation have the same form as those in base Cerise. The core difference is that each invariant remains valid permanently but describes current physical memory only while the version of the address the invariant protects is active. Restoring a previously active version guarded by, say, a safety invariant is essentially a matter of proving that the contents of the currently active version are safe, a familiar obligation in base Cerise.

7.2 Non-invasiveness and Orthogonality

To provide further evidence for our claims of non-invasiveness and orthogonality, we prove programs that vary in the features their program behavior involves and the security properties their correctness relies on. This variation plays an important role in showing how the use of proof machinery grows as programs become more complex, since non-invasiveness concerns both where proofs invoke the machinery and where they can leave it unused.

Specifically, we prove the example programs previously verified either on paper or in Rocq from Cerise with KLR [15, 41] and the awkward example [2], list their program features and the security properties the correctness of each relies on, and report whether their proofs use the logical memory and/or ghost stack machinery in table 2. Neighboring rows differ in only one feature or property.

The result in table 2 has a staircase shape and shows that indeed the two layers are invoked only where the program features or security properties each layer addresses are relevant. By contrast, prior proofs in Cerise with KLR have always required world reasoning. The fact that the example programs proved from base Cerise (§ 7.1) did not invoke any of the machinery, just like `f1`, strengthens our argument.

Neither `f1` nor `f2` invokes logical reclamation or ghost stack reasoning. The remaining benchmark programs involve suspension and all invoke logical reclamation. This pattern suggests that logical reclamation is needed only when suspension is involved, and not necessarily when the program shares the stack.

Well-bracketed control flow alone does not call for ghost stacks. Both `f3` and VAE rely on well-bracketed control flow, where a well-bracketed return is captured by whether an assertion of the intended value of a private variable succeeds. A key distinction is that `f3`'s private variable lives on the stack while VAE's lives in the heap. The logical memory machinery alone can then handle changes to `f3`'s variable that break the invariant over it, but not changes to VAE's variable. This is because heap regions are irrevocable in our setting, thereby calling for ghost stacks.

The logical memory and ghost stack rules Revoke, Restore, GSAgree, and GSUpd are stated without reference to each other, providing positive signs that these two layers are orthogonal. We make a further observation that the awkward example's proof largely reuses VAE's proof components without the ghost stacks, further suggesting orthogonality and compositionality.

Table 2. Selective use of proof machinery across the benchmarks (● used, ○ not used, [ ] no proof to compare against, the bracketed mark being what we expect). The Amarena columns exclude merely extracting and returning version resources at the current version. LSE stands for local state encapsulation; WBCF stands for well-bracketed control flow.

				Amarena		Prior work [15, 43]
Benchmark program	**Property**	**Stack sharing**	**Reversible revocation**	Logical reclamation	Ghost stacks	World reasoning
`f1` [41]	LSE	○	○	○	○	●
`f2` [41]	LSE	●	○	○	○	●
Awkward example [2]	LSE	●	●	●	○	[●]
`f3` [41]	LSE + WBCF (stack)	●	●	●	○	●
VAE	LSE + WBCF (heap)	●	●	●	●	●

Table 3. Mechanized proof comparison with prior work: the VAE proofs against Cerise with KLR (top) and proof LoC by module of the whole developments (bottom). A proof sentence is a command within a proof that ends in a period. The files behind each module of all three developments are listed in table 7. KLR stands for Kripke logical relation.

				Logical memory & ghost stack / world reasoning lemma uses				
	Spec LoC	**Proof LoC**	**Proof sentences**	outside the closure	$x \leftarrow 0;$ $f()$	$x \leftarrow 1;$ $f()$	assert (return)	total
Cerise with KLR [15]	686	2,845	3,783	19	22	54	127	222
Amarena	493	968	983	2	8	8	7	25

	Machine	**Program logic**	**World/ Logical mem.**	**Logrel & FTLR**	**Macros & infra**	**Proofmode & utils**	**Total**
Cerise with KLR [15]	531	2,386	4,767	1,713	2,284	1,128	12,809
Base Cerise [14]	518	2,157	N/A	1,402	1,121	1,795	6,993
Amarena	502	2,052	922	1,763	1,961	1,283	8,483

7.3 A Smaller VAE Proof, a Lighter Development, and Low Specification Overhead

We compare our proof of VAE with the mechanized proof in Cerise with KLR [15], both under scall (table 3)[6]. The prior proof invokes its world reasoning lemmas 222 times, compared with 25 uses of Amarena's logical memory and ghost stack machinery. Across the closure's two calls and return, the prior proof's counts rise from 22 to 54 to 127, while Amarena's remain nearly constant at 8, 8, and 7. The prior proof's counts grow because, before the closure returns to the adversary, the proof has to show that the final world is a future of the world in which the return capability was safe. Every intervening world update adds to that obligation. Amarena instead restores the active versions and the ghost stack to the state under which the return capability was safe, so its return costs no more than its calls.

The prior VAE proof has 2.9× as many proof lines as ours, 2,845 against 968. Counting proof sentences, the commands that end in a period, reduces sensitivity to line breaking and widens the

[6]Georges et al. [15] also verify scallU, a variant that uses uninitialized capabilities to avoid stack clearing. We do not verify this variant.

gap to 3.8×, 3,783 against 983. Although proof style and automation affect both measures, and Amarena inherits stronger automation from base Cerise, both proofs apply the counted lemmas explicitly, making these counts less sensitive to differences in automation. Therefore, the lower lemma use counts, smaller proof sizes, and absence of explicit world relation reasoning support the claim that this proof is structurally simpler.

Amarena's reusable development consists of 8,483 proof lines against 12,809 in Cerise with KLR[7]. Most of the saving lies in the logical memory layer, at 922 lines against 4,767 for world machinery. Relative to base Cerise's 6,993 proof lines, Cerise with KLR is about 83% larger and Amarena about 21% larger. Amarena therefore supports temporary delegation while keeping its reusable development closer in size to base Cerise's.

A table comparing specification and proof sizes for the same eight examples in base Cerise and Amarena is provided in Appendix C. The table also reports sizes for the five benchmark programs in table 2. Across these thirteen programs, Amarena's specifications range from 67 to 698 lines. The share of specification lines containing explicit version points-to assertions ranges from 2.3 to 6.4%, indicating low specification overhead.

8 Related Work

We organize this section by reasoning techniques closely related to our approach or used to reason about different forms of revocation in capability systems.

The Kripke logical relation. A line of predecessors to Cerise [14] verifies revocation of local capabilities [43] and their variants, uninitialized [15] and directed [18] capabilities. All these works reason with a Kripke logical relation, either on paper [43] or mechanized in Iris as a program logic paired with the relation [15, 18]. StkTokens [42] also uses a Kripke logical relation, to verify a stack-based calling convention built on linear capabilities, which cannot be copied. It proves full abstraction of an overlay semantics, not robust safety as in our work.

The Kripke and Amarena logical relations both pair the physical machine state with a logical state. In the Kripke approach, this logical state is a world w, mapping memory regions to state transition systems and their current states. In Amarena, it is a pair $(vm, \gamma s)$ of a version map and a ghost stack. In both, properties of revocable regions can hold under one logical state but fail under another. But the two approaches use logical states in fundamentally different ways. For a given region, a Kripke invariant or assertion packages into its definition all cases the region can be in across revocation stages, and the world determines which case currently holds. For the same region, Amarena keeps separate invariants or assertions, and uses version maps and ghost stacks as an indirection that selects which one is in force. The schema below makes this high-level difference concrete by comparing the ingredients of each design. The indices of $P(w_1)$ and $P(vm, \gamma s)$ indicate that P does not necessarily hold once the indices have changed.

$$\text{Kripke: } \underbrace{\text{logical state}}_{w} + \underbrace{\text{update } w_1 \ + \ \text{restrict the update}}_{\text{world reasoning}} + \underbrace{\text{reuse } P(w_1) \text{ in } w_2}_{\text{prove } w_2 \text{ is a future world of } w_1}$$

$$\text{Amarena: } \underbrace{\text{logical state}}_{(vm, \gamma s)} + \underbrace{\text{update } (vm, \gamma s)}_{\substack{\text{reclaim part of } vm \\ + \text{ push to } \gamma s}} + \underbrace{\text{restrict the update } (= \ \text{reuse } P(vm, \gamma s))}_{\substack{\text{restore } (vm, \gamma s)\text{: re-establish invariants} \\ \text{over the reclaimed part of } vm + \text{ pop from } \gamma s}}$$

Worlds and their protocols carry meaning specific to the revocation mechanism and the program being verified. The Kripke relation therefore cannot decouple updates from restrictions, since a world changes only by following its transition protocol. Version maps and ghost stacks have no such protocols. They act as bookkeeping devices, updated by generic reversible ghost operations:

[7]We exclude proofs for uninitialized capabilities, an extension that Amarena does not support, as far as the file structure allows. Files devoted solely to this extension are left out, but support within shared files remains counted.

activating a fresh or previous version, and pushing or popping a ghost name. Because Amarena decouples updates from restrictions, restrictions apply only when restoring a previous logical state.

Restoring an earlier logical state $(vm, \gamma s)$ is how Amarena justifies reusing invariants and assertions indexed by that state. This amounts to re-establishing the invariants attached to the restored logical memory region or the original topmost ghost name and then returning to the earlier logical state by restoring active versions or popping the ghost stack. The VAE proof makes this obligation concrete in § 4.5 and Appendix A. To reuse $P(w_1)$, the Kripke relation keeps the current world w_2 instead and shows w_2 is a future world reachable from w_1 by the protocol.

Reasoning directly about invariants over revocable memory regions, instead of future-world relations, pays off in several ways. A world can cover not only the region being revoked but also heap memory shared between trusted and untrusted components and protected by safety invariants. Reusing an assertion in the Kripke relation requires proving that the entire current world w_2 is reachable from w_1, which imposes obligations on parts of the world irrelevant to revocation. In Amarena, reuse concerns only the revoked region, the previous topmost ghost name, or both. Everything else stays framed, so reasoning is confined to the part of the logical state that has changed. This narrower obligation and the orthogonality between versions and ghost names give Amarena a simpler proof interface, where each mechanism is invoked only when needed (§ 7.2).

Amarena remains compatible with the Cerise-style logical relation because restoration imposes familiar Cerise proof obligations. To re-establish an Iris invariant over the re-activated memory region, for example $\boxed{\exists lw_1.\, a_{v_1} \mapsto lw_1 * \mathcal{V}_{vm,\gamma s}(lw_1)}$, the proof shows that a word is safe to share with the adversary. In particular, RestoreInv writes the word lw_2 from the currently active version $a_{v_2} \mapsto lw_2$ back to $a_{v_1} \mapsto lw_2$. The remaining obligation is exactly $\mathcal{V}_{vm,\gamma s}(lw_2)$.

Logical memory and ghost stacks are also general, since their rules (Fig. 9 and § 4.4) are stated without reference to any revocation mechanism or program. The hard work is confined to defining a logical relation expressive enough for the mechanism being verified. Another mechanism, such as the convention scallU of Georges et al. [15] using uninitialized capabilities, would therefore call for a new logical relation, but the same logical memory and ghost stack machinery should carry over unchanged.

Logical memory. The main differences between Amarena's logical memory and GCSL have been discussed in § 4.3.

Cerisier [39] adapts logical memory [24] to a machine extended with primitives for enclaves and local attestation. When an enclave claims a region, a runtime *memory sweep* scans the whole machine and checks that no capability held outside the enclave overlaps that region. The sweep resembles stop-the-world garbage collection. Once it succeeds, the proof logically reclaims this memory region and reallocates the fresh version to the enclave. Revocation is global and irreversible, so Cerisier's adaptation stays close to GCSL. Much of the comparison with GCSL in § 4 therefore applies to Cerisier as well.

To the best of our knowledge, existing work that separates logical from physical memory works either in settings with garbage collection [20, 24, 31, 34], or in Cerisier's case, with a global memory sweep that its authors describe as GC-like [39]. In all of them, memory reclamation is permanent. Amarena is the first adaptation of logical memory to support both component-local unreachability and the restoration of suspended accesses.

Ghost stacks. The full ghost stack machinery is designed to be general. It allows several ghost stacks at once, each with its own name, and encodes arbitrary state transition systems with public and private transitions.

Amarena deliberately uses a lighter form of ghost stacks. A single ghost stack, appearing only in the logical relation, gives the relation just enough structure to express the well-bracketed control flow that scall enforces. However, we expect that Amarena's ghost stack can be extended toward the full machinery, leaving logical memory unchanged, since the two are orthogonal (§ 7.2).

Reckon [35] also uses a simpler form of ghost stacks to verify VAE. It tracks a ghost stack in its program logic, where triples for untrusted calls are parameterized by an arbitrary condition that the call must return unchanged. Reckon proves robust safety for a C-like language hardened by a capability-inserting compiler. Its semantics assumes a secure calling convention as a trusted machine primitive, so all programs, the adversary's included, are well-bracketed by construction. Reckon therefore leaves implementing and verifying that convention to future work.

Cancelable invariants. Iris-MSWasm [30] uses *cancelable invariants* to reason about a memory-safe WebAssembly that accesses memory through handles, which are its version of capabilities. Each cancelable invariant is tied to a ghost name, and canceling it swaps the cancellation token for that name with the resources the invariant protects. Iris-MSWasm's logical relation guards each memory segment with a cancelable invariant that is canceled when the segment is freed. Freeing a segment marks it freed in a runtime allocator, a trusted, centralized mechanism that mediates access, so a stale handle traps when it is used. Iris-MSWasm does not reason about suspension, and its logical relation stays close to base Cerise.

It is unclear whether cancelable invariants are also suitable for reasoning about stack revocation that is later reversed. A proof might be able to use this to establish a new invariant over the released resources, recording the trusted stack frame, and later cancel that invariant in turn and swap the resources back for the original token. But a memory capability such as ⓐ in VAE does not become unsafe when the invariants its safety depends on are canceled. Cancellation indeed releases the resources they guard but leaves the invariants in place, now guarding the cancellation token instead. Additional machinery would be needed to tie the swap to invalidating and reinstating the safety of memory capabilities.

Sentry capabilities pose a further challenge since their safety is not directly tied to invariants over memory regions. They also cannot always be proved safe directly as they might come from the adversary, whose code we cannot step through.

References

[1] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University. https://home.ttic.edu/~amal/ahmedthesis.pdf

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009.* ACM, 340–353. doi:10.1145/1480881.1480925

[3] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo I. Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2025. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025.* ACM, 67–84. doi:10.1145/3731569.3764844

[4] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert M. Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023.* ACM, 641–653. doi:10.1145/3613424.3614266

[5] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. In *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994.* ACM Press, 319–327. doi:10.1145/195473.195579

[6] Jianyi Cheng, A. Theodore Markettos, Alexandre Joannou, Paul Metzger, Matthew Naylor, Peter Rugg, and Timothy M. Jones. 2025. Adaptive CHERI Compartmentalization for Heterogeneous Accelerators. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 2002–2016. doi:10.1145/3695053.3731062

[7] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. doi:10.1145/365230.365252

[8] Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. ACM, 143–156. doi:10.1145/1863543.1863566

[9] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 185–198. doi:10.1145/1706299.1706323

[10] Hossam ElAtali, Merve Gülmez, Thomas Nyman, and N. Asokan. 2025. BLACKOUT: Data-Oblivious Computation with Blinded Capabilities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*. ACM, 2039–2053. doi:10.1145/3719027.3765169

[11] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 608–625. doi:10.1109/SP40000.2020.00098

[12] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert M. Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. ACM, 251–268. doi:10.1145/3620665.3640416

[13] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/LMCS-17(3:9)2021

[14] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. doi:10.1145/3623510

[15] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434287

[16] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2023. *Cerise: Coq Development*. https://github.com/logsem/cerise/tree/journal Branch `journal`, commit `9b3df10`; archived as `swh:1:rev:9b3df1040d4eac9906c35a56af3ea76676c46674`.

[17] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. `cap_machine.logrel`: Definition of the logical relation. https://logsem.github.io/cerise/journal/cap_machine.logrel.html#interp. Coq formalization, Cerise project. Accessed 2026-05-21.

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

[19] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform - Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[20] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 716–744. doi:10.1145/3622823

[21] Merve Gülmez, Håkan Englund, Jan Tobias Mühlberg, and Thomas Nyman. 2025. Mon CHERI: Mitigating Uninitialized Memory Access with Conditional Capabilities. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*. IEEE, 829–847. doi:10.1109/SP61157.2025.00133

[22] Merve Gülmez, Ruben Sturm, Hossam ElAtali, Håkan Englund, Jonathan Woodruff, N. Asokan, and Thomas Nyman. 2026. PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities. In *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity26/presentation/guelmez

[23] Angus Hammond, Ricardo Almeida, Thomas Bauereiss, Brian Campbell, Ian Stark, and Peter Sewell. 2025. Morello-Cerise: A Proof of Strong Encapsulation for the Arm Morello Capability Hardware Architecture. *Proc. ACM Program.*

Lang. 9, PLDI (2025), 1961–1983. doi:10.1145/3729329

[24] Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2011. Separation Logic in the Presence of Garbage Collection. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 247–256. doi:10.1109/LICS.2011.46

[25] Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese. 2023. Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. ACM, 2083–2097. doi:10.1145/3576915.3616602

[26] Iris Project. [n. d.]. Library iris.base_logic.lib.gen_heap. https://plv.mpi-sws.org/coqdoc/iris/iris.base_logic.lib.gen_heap.html. Accessed 2026-06-29.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152. doi:10.1145/3418295

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

[29] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 637–650. doi:10.1145/2676726.2676980

[30] Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 304–332. doi:10.1145/3689722

[31] Jean-Marie Madiot and François Pottier. 2022. A separation logic for heap space under garbage collection. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. doi:10.1145/3498672

[32] Noah Mauthe, Eric Ackermann, and Sven Bugiel. 2026. SoK: Capability Operating Systems: Is the Future Finally Here?. In *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, Baltimore, MD, 3715–3734. https://www.usenix.org/conference/usenixsecurity26/presentation/mauthe

[33] Mark Samuel Miller. 2006. *Robust composition: Towards a unified approach to access control and concurrency control*. Ph. D. Dissertation. Johns Hopkins University Baltimore, Maryland, USA.

[34] Alexandre Moine, Arthur Charguéraud, and François Pottier. 2025. Will It Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection. *ACM Trans. Program. Lang. Syst.* 47, 1 (2025), 3:1–3:71. doi:10.1145/3716312

[35] Niklas Mück, Aïna Linn Georges, Derek Dreyer, Deepak Garg, and Michael Sammler. 2026. Endangered by the Language But Saved by the Compiler: Robust Safety via Semantic Back-Translation. *Proc. ACM Program. Lang.* 10, POPL (2026), 1153–1182. doi:10.1145/3776682

[36] Matthew Naylor, Alexandre Joannou, A. Theodore Markettos, Paul Metzger, Simon W. Moore, and Timothy M. Jones. 2026. CHERI-SIMT: Implementing Capability Memory Protection in GPUs. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2026, Pittsburgh, PA, USA, March 22-26, 2026*. ACM, 50–66. doi:10.1145/3760250.3762234

[37] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1003–1020. doi:10.1109/SP40000.2020.00055

[38] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

[39] June Rousseau, Denis Carnier, Thomas Van Strydonck, Steven Keuchel, Dominique Devriese, and Lars Birkedal. 2026. Cerisier: A Program Logic for Attestation in a Capability Machine. *Proc. ACM Program. Lang.* 10, PLDI (2026), 1004–1028. doi:10.1145/3808287

[40] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*. ACM, 170–185. doi:10.1145/319151.319163

[41] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. Reasoning About a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management - Technical Appendix Including Proofs and Details. *CoRR* abs/1902.05283 (2019). doi:10.48550/ARXIV.1902.05283 arXiv:1902.05283

[42] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* 3, POPL (2019), 19:1–19:28. doi:10.1145/3290332

[43] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2020. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 5:1–5:53. doi:10.1145/3363519

[44] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David F. Oswald, Frank Piessens, and Dominique Devriese. 2023. CHERI-TrEE: Flexible enclaves on capability machines. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 1143–1159. doi:10.1109/EUROSP57164.2023.00070

[45] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. doi:10.1145/3133913

[46] The Rocq Development Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.15149629

[47] The Coq Development Team. 2023. *The Coq Proof Assistant, Version 8.18.0*. Inria. https://coq.inria.fr

[48] Amin Timany. 2018. *Contributions in Programming Languages Theory: Logical Relations and Type Theory*. Ph. D. Dissertation. KU Leuven. https://lirias.kuleuven.be/1838165&lang=en

[49] Amin Timany, Armaël Guéneau, and Lars Birkedal. 2024. The Logical Essence of Well-Bracketed Control Flow. *Proc. ACM Program. Lang.* 8, POPL (2024), 575–603. doi:10.1145/3632862

[50] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[51] Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 243–255. doi:10.1109/CSF.2019.00024

[52] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 29–46. http://www.usenix.org/events/sec10/tech/full_papers/Watson.pdf

[53] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[54] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 20–37. doi:10.1109/SP.2015.9

[55] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 457–468. doi:10.1109/ISCA.2014.6853201

[56] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 545–557. doi:10.1145/3352460.3358288

[57] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 787–804. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

Appendix

A Soundness of the Amarena Program Logic

Logical memory keeps a separate word for each version of an address a, while the machine stores only one word at a. Owning a memory points-to assertion $a_v \mapsto lw$ no longer guarantees that the physical memory stores lw at a. Even when a has only a single version, lw can be a logical capability $(\mathsf{RX}, l, b, e, b, v)$ carrying a version that its physical counterpart $(\mathsf{RX}, l, b, e, b)$ does not have. Executing the program through them would drive the logical and physical machines apart, since the addresses they point to, b_v and b, can hold two different instructions. Whatever we establish about the logical machine is then not necessarily sound for the physical machine.

To solve this, we impose a *state interpretation invariant* $I(reg, mem)$ that holds between execution steps, as in a typical Iris-based program logic. It is a predicate on the physical machine state (reg, mem), asserting that the logical resources we own are consistent with that state. The invariant is hidden from the user, who reasons only about a program's footprint, such as $a_v \mapsto lw$, by applying the instruction rules of Fig. 4 and the ghost operations Revoke and Restore of Fig. 9. We prove once and for all that every rule and both ghost operations preserve the invariant.

Amarena's state interpretation invariant is more involved than Cerise's, which is straightforward: $I_{Cerise}(reg, mem) \triangleq \bullet\, reg * \bullet\, mem$. The resources $\bullet\, reg$ and $\bullet\, mem$ let us infer, from a register or memory points-to assertion alone, the content of r and a in the physical machine, regardless of the rest of the machine, using the lookup rules of the Iris authoritative heap [28, § 6.3.2] and the `gen_heap` library [26]. In Amarena, the invariant is defined as:

$$I(reg, mem) \triangleq \exists\, lreg, lmem, vm.\ \bullet\, lreg * \bullet\, lmem * \bullet\, vm * \ulcorner lreg \sim reg \urcorner * \ulcorner lmem \sim_{vm} mem \urcorner.$$

Intuitively, it says that there is always a logical machine and a version map such that the logical registers and active memory correspond to the physical ones, and they agree with the resources we own, such as $a_v \mapsto lw$ and $a \dashrightarrow v$. A logical word lw *corresponds* to a physical word w, written $lw \sim w$, when the two agree after erasing versions:

$$lw \sim w \quad \triangleq \quad lw = w = z \text{ for some integer } z \quad \text{or} \quad lw = (p, l, b, e, a, v) \text{ and } w = (p, l, b, e, a).$$

A *version map* is a finite map from physical addresses to their active versions. Correspondence lifts pointwise to logical register files and, under a specific version map, to the active regions of logical memories. We define these lifted relations as $lreg \sim reg$ and $lmem \sim_{vm} mem$:

$$\begin{aligned} lreg \sim reg \quad &\triangleq \quad \forall\, r \in \mathrm{dom}(reg).\ lreg(r) \sim reg(r), \\ lmem \sim_{vm} mem \quad &\triangleq \quad \forall\, a_v \in \mathrm{dom}(lmem).\ vm(a) = v \implies lmem(a_v) \sim mem(a). \end{aligned}$$

Registers correspond pointwise. For memory, the relation requires only that whenever a_v lies in logical memory and is active under vm, physical memory at a stores the corresponding word. The authoritative resources $\bullet\, lreg$, $\bullet\, lmem$, and $\bullet\, vm$ satisfy rules such as

$$\bullet\, lreg * r \Mapsto lw \Rrightarrow{*} \ulcorner lreg(r) = lw \urcorner \quad (1) \qquad \bullet\, lmem * a_v \mapsto lw \Rrightarrow{*} \ulcorner lmem(a_v) = lw \urcorner \quad (2)$$

$$\bullet\, vm * a \dashrightarrow_q v \Rrightarrow{*} \ulcorner vm(a) = v \urcorner \quad (3) \qquad \bullet\, vm * a \dashrightarrow v \Rrightarrow{*} \bullet\, vm[a \leftarrow v'] * a \dashrightarrow v' \quad (4)$$

$$\bullet\, lmem \Rrightarrow{*} \bullet\, lmem[a_v \leftarrow lw] * a_v \mapsto lw \quad \text{if } a_v \notin \mathrm{dom}(lmem) \quad (5)$$

Rules (1)–(3) are authoritative lookups. From $r \Mapsto lw$ and $a_v \mapsto lw'$ we infer $lreg(r) = lw$ and $lmem(a_v) = lw'$, and from $a \dashrightarrow_q v$ we infer $vm(a) = v$, where vm is the global version map the invariant uses to relate the logical and physical machines.

Rule (3) is what makes our program logic rules sound, memory instructions above all. Take the hard case, `store` in Fig. 4. The physical and logical machines fetch the same instruction: their program counters correspond by $lreg \sim reg$, so the physical program counter points to address a_{pc}, and the logical program counter to version v_{pc} of a_{pc}. Rule (3) with $a_{\mathrm{pc}} \dashrightarrow_{q_1} v_{\mathrm{pc}}$ gives $vm(a_{\mathrm{pc}}) = v_{\mathrm{pc}}$.

By $lmem \sim_{vm} mem$, the word n at the active version v_{pc} of a_{pc} corresponds to the physical word at a_{pc}. Integers correspond only when equal, so the word at a_{pc} is also n, decoded to the same instruction. A similar argument makes the write sound. Rule (3) with $a \dashrightarrow_{q_2} v$ gives $vm(a) = v$, so a_v is active. To re-establish $lmem \sim_{vm} mem$ we must show that after the write, a_v and a still store corresponding words. The instruction writes the contents of $r_{\mathtt{src}}$ to both a_v and a, and those contents already correspond by $lreg \sim reg$. Correspondence is therefore preserved.

So far the only role of the version resource is to tell us that we fetch from and write to active addresses. This amounts to reading from the version map vm. Fractional ownership is therefore sufficient and returned unchanged in the post-conditions of the instruction rules. Logical reclamation and restoration instead perform a ghost update on the active version resource and the version map vm. This amounts to writing to vm, so exclusive ownership is needed.

Unlike instruction rules, reclamation and restoration take no execution step. A step is the licensed occasion to break I and re-establish it by its end. Without such a step, Revoke and Restore instead preserve I explicitly through the connective $\Rrightarrow\!*_I$:

$$P \Rrightarrow\!*_I Q \quad \triangleq \quad \forall\, reg, mem.\, P * I(reg, mem) \Rrightarrow * Q * I(reg, mem).$$

The connective lets a ghost update use the authoritative resources and correspondence relations inside the invariant I, without taking a step and for any physical machine state (reg, mem). It turns $\{Q\} \rightsquigarrow \{s.\, R\}$ into $\{P\} \rightsquigarrow \{s.\, R\}$. Revoke and Restore therefore apply at any point in a proof, without exposing the user to I, and are not tied to any instruction, calling convention, or program. To prove Revoke, we open I and use the exclusively owned $a \dashrightarrow v$. Rule (4) changes the active version of a from v to a fresh v_f. Rule (5) allocates a fresh memory resource $a_{v_f} \mapsto lw$, copying the word lw from the active address a_v. The physical machine is unchanged, so re-establishing I amounts to showing the new active address a_{v_f} corresponds to a. Before the update, a_v was active and stored lw. By $lmem \sim_{vm} mem$, lw corresponds to the physical word at a. After the update, a_{v_f} is active and stores the same lw, so correspondence is preserved.

Restore is sound for the same reason. It makes an existing version v_1 active only after overwriting a_{v_1} with the word lw_2 in the currently active version a_{v_2}. By $lmem \sim_{vm} mem$, lw_2 already corresponds to the physical word at a. Since the physical machine is unchanged, the newly active address a_{v_1} corresponds to a as well.

Not every switch back to an existing version is allowed. Often the version being restored is shared and guarded by an Iris invariant of the form $\boxed{\exists\, lw.\, a_{v_1} \mapsto lw * P(lw)}$, such as a safety invariant in the Cerise logical relation. This case is covered by RestoreInv. Switching to v_1 overwrites a_{v_1}, so the invariant must be re-established for the word copied from the active version. The rule therefore requires that word to satisfy P before the switch. Once that premise is proved, soundness follows as in Restore. In our calling convention examples, VAE in particular, this premise is trivial when the closure returns, because the closure clears its entire supplied stack region before returning and the integer 0 is safe to share.

B Derived Lemmas of the Logical Relation

This appendix states the derived lemmas that § 6 explains in prose. Reclaim extracts, from the version resource bundle in $\mathcal{E}_{vm,\gamma s}(-)$, the exclusive version points-to assertions over a safe stack capability's range:

$$\mathcal{V}_{vm,\gamma s}(\mathsf{RWLX}, \mathsf{Local}, b, e, -, v) * \left(\mathop{\Large *}_{(a,v') \in vm} a \dashrightarrow v' \right) \mathrel{-\!\!*} [b, e) \dashrightarrow v * \mathop{\Large *}_{(a,v') \in vm,\, a \notin [b,e)} a \dashrightarrow v' \quad \text{(Reclaim)}$$

GlobalStackDisj states that a safe global capability does not overlap a safe stack capability:

$$\begin{aligned} &\ulcorner p \neq \mathrm{O} \urcorner * \mathcal{V}_{vm,\gamma s}(p, \mathsf{Global}, b_1, e_1, -, v_1) * \\ &\mathcal{V}_{vm,\gamma s}(\mathsf{RWL/RWLX}, \mathsf{Local}, b_2, e_2, -, v_2) * \left(\mathop{\Large *}_{(a,v)\in vm} a \dashrightarrow v \right) \\ -\!\!* \ &\ulcorner [b_1, e_1) \cap [b_2, e_2) = \emptyset \urcorner * \mathop{\Large *}_{(a,v)\in vm} a \dashrightarrow v \end{aligned} \tag{GlobalStackDisj}$$

ActiveAccess extracts, for each address within the range of a safe capability, an exclusively owned or persistent version points-to assertion:

$$\begin{aligned} &\ulcorner p \neq \mathrm{O} \wedge b \leq a < e \urcorner * \mathcal{V}_{vm,\gamma s}(p, l, b, e, -, v) * \left(\mathop{\Large *}_{(a',v')\in vm} a' \dashrightarrow v' \right) \\ -\!\!* \ &\left(a \dashrightarrow v * \mathop{\Large *}_{(a',v')\in vm,\ a'\neq a} a' \dashrightarrow v' \right) \vee \left(a \dashrightarrow\!\!\square\ v * \mathop{\Large *}_{(a',v')\in vm} a' \dashrightarrow v' \right) \end{aligned} \tag{ActiveAccess}$$

C Specification and Proof Sizes of the Examples

Table 4. Specification and proof lines for the base Cerise examples (top) and the benchmark programs of table 2 (bottom). V_s counts the Amarena specification lines that contain version points-to assertions, with their share of the example's Amarena specification lines in parentheses. Dashes mark programs that base Cerise does not prove.

	Base Cerise [14]		**Amarena**		
Example	Spec.	Proof	Spec.	V_s (%)	Proof
Buffer	64	86	67	2 (3.0%)	74
Minimal counter	131	195	133	3 (2.3%)	183
Adder	284	283	234	10 (4.3%)	274
Counter	430	921	401	20 (5.0%)	517
Read-only sharing	209	359	103	4 (3.9%)	237
Dynamic sealing	565	638	698	45 (6.4%)	844
Interval library	484	630	533	25 (4.7%)	724
Interval client	452	846	619	18 (2.9%)	895
`f1`	—	—	308	17 (5.5%)	600
`f2`	—	—	265	11 (4.2%)	520
Awkward example	—	—	435	20 (4.6%)	794
`f3`	—	—	457	22 (4.8%)	604
VAE	—	—	493	20 (4.1%)	968

Table 4 compares specification and proof sizes for the eight examples in base Cerise and reports the same sizes for Amarena's benchmark programs. Counts cover only each example's own files and leave out shared support, such as adequacy templates and call specifications, and imported libraries. The sealing and interval counts include the constructor and activation code inside those files. The ports adapt capability encodings and some macros and rule out some cases where capabilities are local, so the comparison does not assume identical instruction listings. The script `base_examples_compare.py` reproduces the table.

Table 5. Mechanized proof comparison with prior work: the VAE proofs against Cerise with KLR (top) and proof LoC by module of the whole developments (bottom). A proof sentence is a command within a proof that ends in a period. The files behind each module of all three developments are listed in table 7. KLR stands for Kripke logical relation.

				Logical memory & ghost stack / world reasoning lemma uses				
	Spec LoC	**Proof LoC**	**Proof sentences**	outside the closure	$x \leftarrow 0;$ $f()$	$x \leftarrow 1;$ $f()$	assert (return)	total
Cerise with KLR [15]	686	2,845	3,783	19	22	54	127	222
Amarena	493	968	983	2	8	8	7	25

	Machine	**Program logic**	**World/ Logical mem.**	**Logrel & FTLR**	**Macros & infra**	**Proofmode & utils**	**Total**
Cerise with KLR [15]	531	2,386	4,767	1,713	2,284	1,128	12,809
Base Cerise [14]	518	2,157	N/A	1,402	1,121	1,795	6,993
Amarena	502	2,052	922	1,763	1,961	1,283	8,483

D Evaluation Measures and Lemma Counts

Table 5 repeats the evaluation table from the paper for reference.

The top panel of table 5 compares the two VAE proofs by size and by uses of their distinctive machinery. Specification and proof lines come from `coqwc`, excluding comments and blank lines. A proof sentence is a command ending in a period after `Proof` or `Next Obligation`, including the command that closes the proof but excluding the opener. The scripts approximate this count, treating a chain of tactics joined by semicolons as one sentence. Sentence counts reduce sensitivity to line breaking, but still depend on proof organization and automation.

The lemma counts record explicit uses of the selected machinery of each development, including its initialization. For Amarena, examples include logically reclaiming a region, establishing agreement between ghost stack names, and recalling that a safe global capability is safe at every index. For prior work, they include constructing and inspecting worlds, proving that one world is a future of another, and carrying assertions across those relations. The transfer columns divide the closure proof at its two calls and return. Amarena gives each transfer its own specification; the prior proof gives one specification for the closure, so the script splits its proof at the corresponding jumps. Uses inside helper proofs are counted in the block that invokes the helper. The remaining column counts uses outside the closure body.

Table 6 lists every counted lemma with its number of uses, so the tallies can be audited lemma by lemma. The scripts accompanying this supplement regenerate it, and their `README` states the counting rules and the exclusions applied to both sides.

The protocol that tracks the closure's local variable is counted on both sides. Ours is a standalone one-shot ghost theory (`examples/sts.v`), used four times. The prior proof tracks the same variable through the custom component of its world, whose lemmas account for 11 of its 222 uses. The initialization sites recorded as `alloc_version_map` and `alloc_ghost_stack` construct the version map and ghost stack once at adequacy, the counterparts of prior work's `gen_sts_init` and `heap_init`, and both sides count them. Lemma `interp_global_all` recalls that the adversary is safe with respect to any index. Prior work invokes a similar lemma, `interp_monotone`, at the same proof points, because in that setting safety must be carried across a future world relation.

Table 6. Every lemma behind the lemma use counts of table 3, with its number of uses.

Amarena: revocation machinery and one-shot protocol (25 uses, 15 distinct)

4	insert_range_vers_cur	3	reclaim_range
2	gns_reopen	2	interp_global_all
2	lend_range	2	shoot
2	version_mapstoM_switch_range	1	alloc_ghost_stack
1	alloc_version_map	1	gns_agree
1	interp_wl_restore	1	pending_init
1	pending_not_shot	1	restore_range_inv
1	version_mapstoM_extract_range_l		

Georges et al. [15]: worlds and region states (222 uses, 63 distinct)

24	std_sta_update_multiple_lookup_same_i	15	std_sta_update_multiple_lookup_in_i
15	std_update_multiple_loc_sta	14	std_update_multiple_loc_rel
13	related_sts_priv_trans_world	8	close_list_std_sta_revoked
8	rel_agree	6	close_list_std_sta_same
6	related_sts_pub_priv_world	6	revoke_lookup_Temp
5	std_update_multiple_sta_dom_subseteq	4	close_list_std_sta_same_alt
4	full_sts_static_all	4	interp_monotone
4	related_sts_priv_refl_world	4	revoke_monotone_lookup_same
4	std_rel_pub_rtc_Static	4	sts_full_rel_loc
4	sts_full_state_loc	3	close_list_dom_eq
3	related_sts_priv_pub_trans_world	3	related_sts_pub_world_fresh_loc
3	revoke_dom_eq	2	close_list_related_sts_pub
2	close_list_std_sta_is_Some	2	extract_temps_split
2	interp_monotone_nl	2	monotone_close
2	monotone_revoke_keep_some	2	region_monotone
2	region_revoked_to_static	2	region_state_pwl_monotone
2	related_sts_priv_world_static	2	related_sts_pub_world_static
2	related_sts_std_pub_refl	2	revoke_related_sts_priv
2	revoke_std_sta_lookup_Some	2	std_rel_pub_rtc_Temporary
2	std_sta_update_multiple_is_Some	2	std_update_multiple_revoke_commute
2	sts_update_loc	1	extend_region_perm_sepL2
1	extend_region_static_single_sepM	1	gen_sts_init
1	heap_init	1	monotone_revoke_list_region_def_mono_same
1	override_uninitializedW_lookup_nin	1	override_uninitializedW_lookup_some
1	region_close_static_to_temporary_alt	1	region_has_static_addr_Forall
1	region_open_static	1	region_static_to_revoked
1	related_sts_priv_world_revoked	1	related_sts_pub_priv_trans_world
1	related_sts_pub_refl_world	1	related_sts_pub_world_static_to_temporary
1	related_sts_std_priv_refl	1	revoke_lookup_Static
1	revoke_monotone	1	std_update_multiple_proj_eq
1	std_update_multiple_std_sta_eq	1	std_update_mutiple_related_monotone
1	sts_alloc_loc		

The bottom panel of table 5 groups proof lines by module. Table 7 lists the source files behind each module, following each development's directory structure.

Table 7. The source files behind each module of table 3. File names drop the `.v` extension. All three developments exclude obsolete files and example programs.

Module	Cerise with KLR [15]	Base Cerise [14]	Amarena
Machine & op. sem.	addr_reg, cap_lang, machine_base, machine_parameters, machine_run	addr_reg, cap_lang, machine_base, machine_parameters, machine_run	addr_reg, cap_lang, machine_base, machine_parameters, machine_run
Program logic	rules, rules_base, rules_AddSubLt, rules_Get, rules_IsPtr, rules_Jmp, rules_Jnz, rules_Lea, rules_Load, rules_Mov, rules_Restrict, rules_Store, rules_Subseg	rules, rules_base, rules_AddSubLt, rules_Get, rules_IsPtr, rules_Jmp, rules_Jnz, rules_Lea, rules_Load, rules_Mov, rules_Restrict, rules_Store, rules_Subseg	rules, rules_base, rules_AddSubLt, rules_Get, rules_Jmp, rules_Jnz, rules_Lea, rules_Load, rules_Mov, rules_Restrict, rules_Store, rules_Subseg
World / logical mem.	region, region_invariants, region_invariants_allocation, region_invariants_revocation, region_invariants_static, sts, monotone, multiple_updates, mono_ref, monocmra	N/A	machine_logical, revocation, machine_logical_base, machine_logical_corr, machine_logical_update
Logrel & FTLR	logrel, fundamental, ftlr_base, interp_weakening, AddSubLt, Get, IsPtr, Jmp, Jnz, Lea, Load, Mov, Restrict, Store, Subseg	logrel, fundamental, ftlr_base, interp_weakening, AddSubLt, Get, IsPtr, Jmp, Jnz, Lea, Load, Mov, Restrict, Store, Subseg	logrel, fundamental, ftlr_base, interp_weakening, AddSubLt, Get, Jmp, Jnz, Lea, Load, Mov, Restrict, Store, Subseg
Macros & infra	addr_reg_sample, malloc, region_macros, scall, stack_macros, stack_macros_helpers	addr_reg_sample, malloc, assert, macros_new, macros_helpers, call	macros, addr_reg_sample, assert_subroutine, malloc_subroutine, sts, assert_macro, call, crtcls, fetch, malloc_macro, mclear, prepcap, rclear, reqloc, reqperm, reqsize, retepl, scall, store_locals
Proofmode & utils	iris_extra, stdpp_extra, contiguous, disjoint_regions_tactics	iris_extra, stdpp_extra, NamedProp, class_instances, classes, contiguous, disjoint_regions_tactics, map_simpl, map_simpl_test, mkregion_helpers, proofmode, proofmode_instr_rules, region, solve_addr, solve_addr_extra, solve_pure	iris_extra, stdpp_extra, NamedProp, class_instances, classes, contiguous, disjoint_regions_tactics, map_simpl, map_simpl_base, mkregion_helpers, proofmode, proofmode_instr_rules, region, register_tactics, solve_addr, solve_addr_extra, solve_pure, tactics_helpers

For Cerise with KLR, the counts leave out the files that exist only for uninitialized capabilities, which neither of the other two developments supports. Support for these capabilities inside shared files stays counted. For base Cerise, the counts leave out its binary logical relation, its adequacy templates, `monotone`, which only its case studies use, and `macros`, an earlier copy of `macros_new`.