Concurrent Cerise - A Program Logic for Multi-core Capability Machine

Bastien Rousseau (ENS Rennes)
supervised by Lars Birkedal and Aïna-Linn Georges (Aarhus University)

July 11, 2022

Abstract

Memory safety is a major source of vulnerabilities in computer systems. To prevent these memory safety issues, capability machine are a type of security-oriented CPU that offer a fine-grained compartmentalization of memory using hardware capabilities, a machine word that include a form of authority over the memory. Arm is designing and building Morello, a prototype of capability machine implementing the CHERI capability-supported architecture. A single flaw in design can void all security guarantees, which make the architecture a good target for verification. In particular, the formal methods, based on mathematical foundations, are essential to have strong confidence on the verification. Previous works already propose formal methods to reason about security properties of capability machines, such as Cerise. Cerise is a program logic implemented in the Coq proof assistant, built on top of the Iris separation logic framework. Cerise is able to prove strong properties of CHERI-like machine, *e.g.,* the safety of a stack-based calling convention. In particular, it can verify programs involving interactions between known code and unknown, arbitrary code. Because real-world capability machines are complex, Cerise uses a model of a simplified single-core capability machine, with a small set of instructions. In order to lift these simplifying instructions, we propose Concurrent Cerise, a program logic for multi-core capability machine, with a sequentially consistent memory model. In addition to the extension of the model, we show that reasoning about the security properties already defined in Cerise is orthogonal to the concurrent behaviors. We design different scenarios targeting different threats, involving concurrency and synchronization mechanism, and prove safety properties using Concurrent Cerise.

1 Introduction

Computer systems are omnipresent in our world and our society. Errors and security breaches are costly and harmful. Microsoft has reported that around 70% of their security updates address memory safety vulnerabilities each year [6]. In order to prevent these kinds of vulnerabilities, the University of Cambridge, SRI International and Arm are collaborating to design and build the Morello board, an industrial prototype demonstrator of a security-oriented architecture, CHERI [2, 4]. CHERI is a *capability*-based architecture that offers a fine-grained protection of the memory. The purpose of the Morello board is to provide a platform for researchers to explore new security features that can be implemented in top of this architecture.

In combination with hardware development, we need to identify, mathematically capture and prove strong security guarantees, using formal methods. Formal methods are important when it concerns security. Indeed, a mis-specification can void the security protections that the hardware is meant to provide. Moreover, whether the security properties hold depends on the minute details of the system we are verifying, and formal methods ensure that all details are checked. In particular, mechanized proofs make it possible to get high confidence. Cerise [7] is a project at Aarhus University to develop formal methods and models to reason about CHERI-like machines. Cerise has been used to develop new security mechanisms to inform the evolution of CHERI, and to prove that they provide new, strong security guarantees. The proofs in Cerise are mechanized in the Coq proof assistant and the Iris framework, which provides several guarantees, but the use of formal methods for such low-level model is challenging. Cerise makes simplifying assumptions on the model of the capability machine, but manages to show strong properties and paves the way for larger-scale model. In this work, we propose Concurrent Cerise, an extension of Cerise for multi-core capability machine.

1.1 Context

Current compartmentalization of the memory uses page tables mechanism. Page tables are too coarse-grained to protect individual data structures, individual buffers, or individual stack frames, both in terms of space granularity — a page is much bigger than many individual data structures — and in terms of code granularity — they protect one program from another because the OS distinguishes programs, but the OS does not distinguish one function call from another. This coarse-grained compartmentalization of the memory is thus the root of most memory safety issues like buffer overflow or stack corruption (Return-Oriented Programming, etc.). The CHERI project defines a novel architecture design and hardware implementation with high-security guarantees and a more fine-grained compartmentalization of the memory, supported by the hardware. They extend the conventional architecture with *hardware-supported capabilities.* Capabilities are unforgeable pointers that embed meta-information, such as permissions and bounds. This mechanism allows a fine-grained compartmentalization of the memory, and makes it possible to have stronger security guarantees.

For such major security-oriented architecture, it is essential to establish strong guarantees, using formal methods to get high confidence.

However, defining a formal model and reasoning on low-level Instruction Set Architecture (ISA) raises several challenges, which several lines of work have been tackling over the past few years:

- Mainstream engineering relies on test-and-debug methods, with specifications defined in prose. That may be ambiguous, leading to misunderstanding and poor test coverage.
- Existing formal methods (including Cerise) usually covers only a small part of the ISA, based on handwritten assembly or machine-code semantics. Real-world ISAs are huge, and formal models scale badly. That makes it very difficult to use them, even for basic verification. [5, 11, 12];
- Concurrency in hardware exhibits very relaxed behavior. At user level, relaxed memory model are well-studied [14] whereas it is still unclear at privileged level, *e.g.,* for page tables. Therefore, low-level verifications targeting relaxed memory model for system code is work in progress [13].

Moreover, one of the key challenges of formal methods on security properties is to have a model that captures security properties involving arbitrary, unknown, adversarial code, such as *capability monotonicity* or *local encapsulation.*

This work is based on previous work on formal methods for CHERI-like architectures: the Cerise model. Cerise leverages the strength of Iris [3], a separation logic framework mechanized within the Coq proof assistant. Iris has a very expressive higher-order language of assertions that makes it possible to define program logics and logical relations for different programming languages. It has been used successfully to define program logic for different applications like Rust, distributed systems, fine-grained concurrent data structures, etc.

Iris also well-suited for low-level code, in particular for capability machine architectures with Cerise. The Cerise program logic has been used to prove safety properties for a stack calling convention [8, 9] — in particular the local state encapsulation and well-bracketed control flow, for both integrity and confidentiality.

1.2 Challenges

Cerise is a step towards modelling CHERI-like machines formally, and to reason about the interaction between known and unknown code, but there is still a huge gap between the formal model and a real-world machine. As a first step to bridge this gap, we extend Cerise to model a multi-core capability machine, introducing concurrency. The introduction of concurrency in Cerise brings new challenges, at different levels.

1. **Threat model and consequences for safety** — The concurrency brings new threats. Indeed, on a multi-core machine, the reachability of a capability may depend on the execution order of the cores. For instance, if two cores share a memory buffer concurrently, they may share new

capabilities through this buffer. Thus, we need to define precisely the threat model we want to consider, which behaviors we want to capture and evaluate the consequences of the concurrency on the safety properties already defined.

2. **Motivating examples** — To evaluate the relevance of our threat model, we need to define interesting and complete motivating examples, involving interactions with adversary code and concurrent behaviors. This is non-trivial because some programs, for example, dynamic allocation of memory *malloc*, are safe in a single-core setting, but become unsafe in a concurrent setting.
3. **Operational semantic and program logic** — Once the threat model is well-defined, we need to model the concurrency in the operational semantics of the machine, and adapt the program logic accordingly. In a single-core setting, the behavior of the machine is sequential and deterministic. Cerise leverages such properties by using special mechanisms, such as non-atomic invariant, which are only sound for sequential programs. However, in a multi-core setting, the behavior of the machine is concurrent and therefore non-deterministic. Thus, we need to find new solutions to replace these mechanisms. Moreover, modeling concurrent behaviors is not trivial. Fortunately, the Iris [3] framework offers good tools to reason about such behaviors. Indeed, it has already been used to define program logic for concurrent languages.
4. **Synchronization mechanisms** — Finally, concurrency introduces some synchronization issues. We need to add some synchronization mechanisms in order to capture more safety properties. However, to implement synchronization mechanisms (*e.g.,* a lock library), we need atomic instructions such as *Compare-And-Swap* or *Compare-And-Set*. The current model of Cerise does not provide such instructions, since they are not necessary in a single-core machine. In addition to the extension of the model, we need to prove that the properties of the model still holds, and provide tools to reason about the synchronization mechanisms.

1.3 Goal & Proposal

The goal of this work is to add concurrency to the Cerise model, using a sequentially-consistent memory model. This memory model consists of a non-deterministic interleaving of the instructions. Even though sequential consistency is not a realistic model, it lays the foundations for further work, involving more realistic relaxed memory models.

We propose to tackle each of the challenges previously listed. In particular, we define an acceptable threat model involving unknown, arbitrary code, using motivating examples. These examples are the guideline of the current work, *i.e.,* we want a model complete enough to capture the security properties of these examples. We thus enhance the operational semantic of Cerise, as well as the program logic and the whole model according to this extension. Finally, we demonstrate how to use our extended model scenarios involving concurrency.

1.4 Contributions

Our contributions are:

- We extend the operational semantics of Cerise with concurrency. It models a CHERI-like machine with an arbitrary, but fixed, number of cores. The semantics is non-deterministic and follows a sequentially-consistent memory model.

- We update the Cerise program logic in order to reason about concurrent known programs, and the Cerise logical relation to reason about concurrent unknown, arbitrary programs.

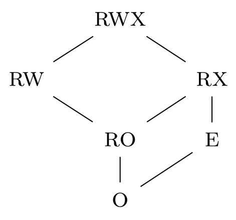
- We design, specify and formally prove safety of different scenarios involving the concurrency, each targeting a different possible threat.

- We extend the ISA of the model with a *Compare-And-Swap* instruction to implement synchronization mechanisms.

- We mechanized the results and the examples in the Iris framework. The sources are available online at `https://github.com/logsem/cerise/tree/concurrency`.

Capability: (p, b, e, a)

$p \in \{\text{O}, \text{RO}, \text{RX}, \text{RW}, \text{RWX}, \text{E}\}$	permission
$b \in \mathit{Addr}$	base address
$e \in \mathit{Addr}$	end address
$a \in \mathit{Addr}$	current address



Lattice over permissions

Figure 1: Model capability

The remaining of this report is organized as follows. First, we introduce the necessary background on capability machines and the Cerise program logic in Section 2. In Section 3, we propose three different scenarios highlighting new different possible threats to the integrity of a secret value, introduced by concurrency. In Section 4, we formally define the new operational semantic and program logic of Concurrent Cerise. We will use these tools in Section 5 to formally prove the two first scenario of Section 3. The last scenario requires a synchronization mechanism, involving *concurrent-safe memory accesses instructions*. We thus extend the set of instructions with a *Compare-And-Swap* instruction, design a lock library and prove the last scenario in Section 6. Finally, the related work is covered in Section 7, and we present further work directions in Section 8.

2 Background

2.1 Capability Machine

A capability machine is a special kind of security-oriented architecture. It offers a mechanism allowing a fine-grained memory compartmentalization and privileges separation: *hardware capabilities*, which are unforgeable token of authority. Capability is a special machine word, distinct from the machine integers, to be used as a pointer. In addition to the memory address to which it points, the capability embeds metadata such as permissions or range validity. In this context, the integers are used for numerical computation only, while the memory accesses and pointer arithmetic require capabilities. In particular, integers cannot be used as pointers. We say that a capability has *authority* over a memory fragment. Indeed, to perform a memory access, one needs to use a capability with enough permission over the memory address. In order to ensure its integrity, a capability is unforgeable: the architecture provides the instructions to derive a capability from another one. The derivation enforces the permission of the new capability to be *less or equal* than the permission of the source capability. Thus, from a given starting state, the machine cannot get more permissions during the execution than it had at the starting state, up to a controlled domain transition. Furthermore, the capabilities work at the level of the virtual memory, the addresses encoded in a capability are virtuals.

Capabilities machine supports the capability mechanism at the hardware level. It provides dynamic checks over the capability permissions. Several projects explored the design of realistic and efficient capability machines. Recently, the CHERI project [2] proposed an architectural and micro-architectural design of a capability machine, leading to the UKRI Digital Security by Design (DSbD) challenge, a collaboration between Arm, SRI International and the University of Cambridge, to develop a real-world processor supporting hardware capabilities, Morello. Moreover, few works target a formal verification of security properties over the CHERI and Morello capabilities machines [5, 11], focusing on the *capability monotonicity*. It is necessary to the continue this effort with stronger properties, involving interactions between known and unknown code, *e.g.,* local encapsulation.

To simplify reasoning about capabilities, we get rid of a concrete representation of the capability using an abstract representation: Figure 1 shows an overview of the abstract model of a capability. A capability is represented as a 4-tuple *(p,b,e,a)*. This capability points to the memory address a and has the permission p on the range of memory $[b, e)$. In order to use the capability *(p,b,e,a)*, a needs to be in bounds: $b \leq a < e$ (therefore, a capability (p,b,e,a) where $a \leq b$, may exist, but cannot be used as such). We distinguish four permissions: R read, W write, X execute and a special permission E for the *sentry capability.* A sentry capability provides a way to encapsulate a memory fragment. It is impossible to perform memory accesses instructions or pointer arithmetic operations over a sentry capability. The only way to use a sentry capability is to jump at the pointed address: the program counter capability is replaced by the sentry capability, with the *RX* permission.

From a capability *(p,b,e,a)*, we can derive a new capability *(p,b,e,a')*, such as the address *a'* is within the range $[b, e)$. Moreover, we can restrict the permission *p* with a less permissive permission *p'*, according to the order given by the lattice in Figure 1, or restrict the range $[b, e)$ with a range $[b', e')$ such that $[b', e') \subseteq [b, e)$. Thus, all derivation of a capability prohibits the possibility to give more permissions than the source capability.

The machine dynamically checks the permission, and the use of an invalid capability may lead to a machine failure. But from a security point of view, failure is a good behavior. In usual machine, a failure may happen because of runtime fault error, *e.g.,* the machine cannot decode the instruction. However, in a capability machine, there is more runtime checks, which can turn a dangerous behavior (for instance an out-of-bounds access) into a runtime fault. The failure prevents illegal instructions to cause damage. Of course, a programmer have to make sure to not write code that could lead the machine to a failure. However, we cannot prevent unknown code to avoid this behavior, *e.g.,* by trying to access memory addresses it is not allowed to. Therefore, we focus on memory safety properties not violated in case of failed, such as integrity of a data. As consequence, a failed machine is safe, because a failed machine cannot compromise data.

2.2 Cerise

Cerise is a program logic to reason formally about CHERI-like capabilities machines. It provides a way to reason about concrete known code interacting with unknown, possibly adversary code.

Cerise models a restricted subset of a CHERI-like Instruction Set Architecture (ISA). It supports about twenty instructions, including usual arithmetic operations over integers, direct and indirect jump. More importantly, it includes memory store and load via capabilities, and capabilities derivation instructions. The machine manipulates machine words, that can be either an integer or a capability. The machine contains a register file (PC + 32 registers), mapping each register name to its content, and a memory which maps each address to its content. The machine has three execution states: Running, Halted or Failed. Each execution step updates the register file, the memory, and the execution state. If the machine is in a Running state, the execution steps fetches the instructions to which the program counter (PC) registers points-to, decodes it, and executes the instruction according to the given operational semantic. If a dynamic check fails, the machine enters in a Failed state (a realistic machine would throw an exception). There is no execution rule for a Halted or Failed machine.

Cerise aims to prove properties of concrete programs, at the level of the operational semantic of the machine. The usual and direct method to prove such property is to establish this property directly on a sequence of reduction of the operational semantics. However, on low-level semantic, it becomes intractable even for simple programs. Instead, Cerise leverages the Iris framework to derive a program logic with a system of rules more convenient to use, rather than reasoning directly on the operational semantic. Moreover, Iris provides an *adequacy theorem*, which given a theorem expressed in terms of the program logic, extracts an equivalent theorem at the level of the operational semantics.

The rules of the program logic are convenient to reason about known code, but as soon as it has to interact with unknown code, the rules are unusable. Logical relations are useful to prove semantic properties and fit well with program logic. Thus, on top of the program logic, Cerise defines a (unary) logical relation to express the *safety* of a machine word. It distinguishes the *safe-to-share* and *safe-to-execute* relations, mutually recursively defined. Intuitively, a word that is *safe-to-share* if it can be shared with an arbitrary code without breaking the invariant. Similarly, a word is *safe-to-execute* if the machine can execute the word without breaking the invariant. The *safe-to-share* relation (more commonly called the *value relation* in the literature), $\mathcal{V}(w) : Word \rightarrow iProp$, ($iProp$ is the type of the Iris proposition) gives only access, transitively, to other *safe-to-share* words. The *safe-to-execute* relation (more commonly called the *expression relation* in the literature), $\mathcal{E}(w) : Word \rightarrow iProp$, allows the machine to safely and completely execute the word in any safe context. The *safe-to-execute* relation ensures that, if the machine executes the word starting from a register file containing only safe-to-share words, the machine will preserve the invariant throughout the execution. In other words, the machine will not execute illegal instruction, according to the operational semantic. Moreover, the Fundamental

Theorem of the Logical Relation (FTLR), $\forall w, \mathcal{V}(w) \Rightarrow \mathcal{E}(w)$, asserts that any *safe-to-share* word is also *safe-to-execute*. As a corollary of the FTLR, any capability pointing to a program, which contains only instructions (as opposed to capabilities), is trivially safe-to-share, and thus safe-to-execute. In particular, any arbitrary, unknown code is safe to share, and to execute. Hence, using the logical relation, the FTLR, and its corollary, it is possible to reason about unknown code.

Nevertheless, the whole model of Cerise abstracts a single-core capability machine.

3 Motivating Examples

In this section, we define the threat model using different motivating examples. Each example represents a scenario capturing a possible adversarial threat. Moreover, we want to show that, the security properties we are considering are orthogonal to the concurrency.

3.1 Concurrency and security

From a security point of view, we believe that properties like reachability of capabilities and local encapsulation should be orthogonal to concurrency. Therefore, we want to show that reasoning on security properties in a concurrent setting is similar to reasoning on the same security properties in a sequential setting.

Concurrency may introduce new threats that do not exist in a sequential setting. We define three scenarios using concurrency to highlight different possible adversarial threats. In further sections, for each scenario, we will prove that the integrity of the secret data in never corrupted by the adversarial code, at the level of the operational semantic.

In order to endanger the integrity of the secret data, the scenario usually defines a known code that shares a public memory buffer with an adversary code, in a configuration where the secret data is vulnerable to memory attacks, such as buffer overflow. We then show that the capabilities ensure the protection of the secret data.

In a sequential setting, some known code shares a capability with some adversary code by giving the control flow of the machine to the adversary code through a register containing a capability granting access to the shared buffer. However, in a concurrent setting, we also want to be able to share a memory buffer between multiples core executing concurrently. As a consequence, in a multi-core setting, the notion of *sharing a memory buffer* may be ambiguous. In order to clarify this ambiguity, we define different notions of *sharing*. We name *sequentially-shared buffer* a memory buffer being shared with an adversary code on the same core. In that case, the buffer is shared through a capability stored in a register before performing the jump to the adversarial code. We name *concurrently-shared buffer* a memory buffer being shared with multiple cores of the machine. The buffer may be shared between known, trusted, verified code, or with an unknown, arbitrary, adversarial code running on a different core. The buffer is thus shared with a capability stored in the registers of the different core of the machine. The *sequential-sharing* and *concurrent-sharing* are compatibles notions: a buffer may be *concurrently-shared* with another core, and the authority of this buffer may later be transferred to an adversary code through *sequential-sharing* by one of the cores. We emphasize the orthogonality of the concurrency and the security properties with these notions of sharing, by showing that reasoning about a *sequentially-shared* buffer with some adversary code is similar to reasoning about a *concurrently-shared* buffer with some concurrent adversary code.

3.2 Isolation against unknown code running in parallel

In this first scenario, two cores run their own program in parallel and in isolation. The first core executes its own program, for which we can give a specification (at least, partially). The verified code has access to a secret buffer, containing sensitive data. It sequentially-shares a public part of the buffer with an adversary. The second core executes an unknown program, containing only instructions but not capabilities.

The scenario, sketched in Figure 2, highlights the possibility to run in parallel in complete isolation[1].

[1]In a real-world machine, this isolation may be ensured by the coarse-grained memory compartmentalization mechanism provided by the virtual memory and the page tables. However, who can do more, can do less.

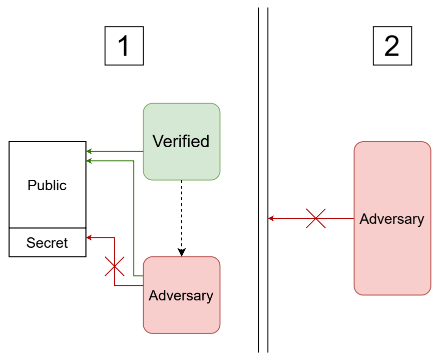


Figure 2: Scenario 1 - Verified code and unknown code running parallel

```
b_buf: 'H', 'i', 0, ; public
           0, 0,     ; secret
e_buf:
```

```
; r0 : wadv
init:
mov r1 PC
lea r1 [data-init]                ; r1: (RWX, init, end, data) ; C1
load r1 r1                        ; r1: (RWX, b_buf, e_buf, b_buf) ; C2
store r1 42
subseg r1 [b_buf] [b_buf+3]       ; r1: (RWX, b_buf, b_buf+3, b_buf) ; C2'
jmp r0
data1: (RWX, b_buf, e_buf, b_buf+3)    ; C2, secret buffer
end:
```

Figure 3: Code restriction sub-buffer for isolation

In particular, the adversary running on the same core cannot corrupt the secret data, and neither can the adversary running on another core.

Example — Sequentially-shared sub-buffer We define a simple program that implements the above scenario. The first core stores a secret data in a buffer and restricts the permissions to protect the secret data, whereas the second core executes an unknown, adversary code. The memory of the adversary contains only instructions and data, and no capabilities: it can derive capabilities from the program counter capability.

In this example, we want to formally prove — at the level of the operational semantic — that the secret buffer is never corrupted, neither by the adversary running sequentially on the same core as the verified program, nor by the adversary running concurrently on the other core.

Figure 3 shows the verified code, as well as its buffer. Initially, the register r_0 contains the adversary word, and PC contains the capability of the program. It has authority over the code part as well as the data. The data contains the capability pointing to the secret buffer. The program executes as follows: if first derives a capability C1 from the PC in order to load the capability C2 in the data section. This capability C2 gives access to the shared buffer. It then stores the secret value in the buffer, and restricts the range of the capability C2 to the public buffer. The restricted capability C2' gives only access to the public buffer. The code finally jumps to the adversary code, giving the capability C2' through a register.

3.3 Concurrently-shared sub-buffer

In this scenario Figure 4, two cores concurrently-share a buffer, in parallel. Each core executes some known, verified code that store a secret data in the *secret* part of the buffer. The *secret-buffer* is concurrently-shared between the known parts of the code. Each code trusts the verified code executed

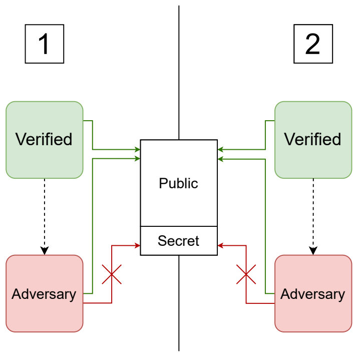


Figure 4: Scenario 2 - Shared buffer

```
init2:
mov r1 PC
lea r1 [data-init]                ; r1: (RWX, init, end, data)
load r1 r1                        ; r1: (RWX, b_buf, e_buf, b_buf)
store r1 (-42)
subseg r1 [b_buf] [b_buf+3]       ; r1: (RWX, b_buf, b_buf+3, b_buf)
jmp r0
data2: (RWX, b_buf, e_buf, b_buf+4) ; secret buffer
end2:
```

Figure 5: Code for the second core

on the other core. The trusted code restricts the buffer to its *public* part, which contains only non sensitive information. The *public-buffer* is then sequentially-shared, by both cores, with unknown, adversary code. We notice that the *secret-buffer* is exclusively *concurrently-shared* with the verified code, whereas the *public-buffer* is both *concurrently-shared* and *sequentially-shared*.

The capabilities protect this buffer against buffer-overflow attacks. From previous work on Cerise already proved the protection of the capabilities for such a buffer in a sequential setting. However, we want to ensure the protection of the secret buffer in a concurrent setting, when the buffers are concurrently-shared between the cores.

Example — Concurrently-shared sub-buffer In this motivating example, we define a simple program that implement the above scenario. We consider a machine with two cores, each core executing its own program in parallel. The known programs are very similar to the one proposed in the previous example. Thus, Figure 3 shows the initial concurrently-shared buffer, as well as the code running on the first core. Figure 5 shows the code running on the second core. The program instructions correspond to the *Verified* program in Figure 4. The memory fragment of the first core corresponds to $[\mathsf{init}, \mathsf{end})$, and the memory fragment of the second core corresponds to $[\mathsf{init}_2, \mathsf{end}_2)$. The concurrently-shared buffer corresponds to $[\mathsf{b}_{\mathsf{buf}}, \mathsf{e}_{\mathsf{buf}})$. The second program is similar to the first one, except the value of the stored value and the address in the secret buffer.

On the contrary to the previous example, virtual memory and page tables do not ensure any coarse-grained compartmentalization of the concurrently-shared memory.

3.4 Memory allocation

In this last scenario Figure 6, two cores run their own program in parallel. The first core executes a verified program, while the other executes an adversary program, as in Section 3.2. Moreover, they both have a linking table containing capabilities pointing to shared macros, for instance the *malloc* macro. *malloc* provides a way to dynamically allocates new memory. Cores may use the macros concurrently. If the implementation of the macro is safe even with the concurrency (via a synchronization mechanism), each call of the macro should provides an allocated memory buffer disjoint from the others.

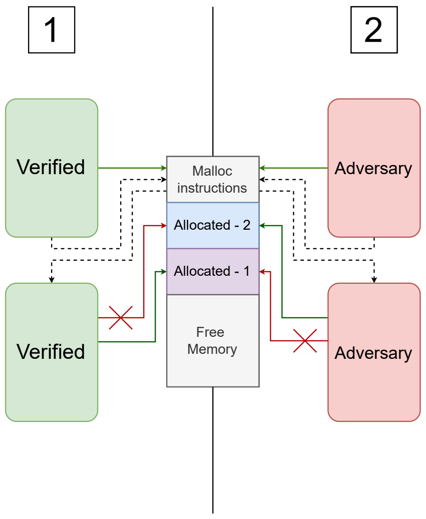


Figure 6: Scenario 3 - Memory allocation

```
init:
malloc 1        ; Allocated memory in r1
store r1 42     ; Store a secret in the allocated region
mov r5 42       ; Prepare the assertion
load r4 r1      ; An adversary may have concurrently changed the value
assert r4 r5    ; (r4 = 42)
halt
end:
```

Figure 7: Code secret data in allocated memory

Example — Secret data in a dynamically allocated buffer In this motivating example, we define a program that implements the above scenario. We consider a 2-cores machine: the first core executes verified code, while the other executes unknown, possibly adversarial code. They both have access to a linking table that contains the capability to a safe *malloc* macro.

Figure 7 shows the instructions of the known code. It allocates a memory address, puts a secret in the newly allocated address, and uses the *assert*[2] macro to ensure that the content of the secret address is not modified.

If the *malloc* macro is *concurrent-safe*, the allocated memory for the verified code should always be disjoint from the other allocated memory. Concretely, the concurrent adversary never have the authority to access to the sensitive data. If a concurrent adversary had the authority over the allocated memory, it might be able to modify the secret value between the execution of the store and the execution of the *assert*. Conversely, if the integrity of the secret value holds, then the adversary does not have the authority over the allocated memory. In order to dynamically check that the assertion holds, it suffices to show that the *assert flag* is invariant during the whole execution.

We define a *thread-safe* malloc macro, and prove a strong specification for it with the program logic in Section 6.4, and prove the scenario holds, *i.e.,* the secret value stored in the dynamically allocated buffer is not modified by another core running concurrently, in Section 6.5

4 Concurrent Cerise

In this section, we formally define the model of Concurrent Cerise. In particular, we define the new operational semantic of the multi-core capability machine, as well as the program logic and the logical

[2]The *assert* macro, defined in Cerise, allows checking the value of a memory address unknown statically (*i.e.,* dynamically allocated). It maintains a flag at a statically known address, which is modified if the assertion does not hold.

a	$\in$	Addr	$\triangleq$	$[0, \mathrm{AddrMax}]$
p	$\in$	perm	$::=$	$\mathrm{O} \mid \mathrm{E} \mid \mathrm{RO} \mid \mathrm{RX} \mid \mathrm{RW} \mid \mathrm{RWX}$
c	$\in$	Cap	$\triangleq$	$\{(p, b, e, a) \mid b, e, a \in \mathrm{Addr}\}$
w	$\in$	Word	$\triangleq$	$\mathbb{Z} + \mathrm{Cap}$
reg	$\in$	Reg	$\triangleq$	$(\mathcal{N} \times \mathrm{RegName}) \to \mathrm{Word}$
m	$\in$	Mem	$\triangleq$	$\mathrm{Addr} \to \mathrm{Word}$
$\mathcal{C}$	$\in$	CoreState	$::=$	Running | Halted | Failed
$\mathcal{E}$	$\in$	ExecState	$\triangleq$	$\mathcal{N} \to \mathrm{CoreState}$
φ	$\in$	Conf	$\triangleq$	$\mathrm{ExecState} \times \mathrm{Reg} \times \mathrm{Mem}$

Figure 8: Base definition machine words and state

relation.

4.1 Operational semantic

Definition The operational semantic models a multi-core capability machine with an arbitrary but fixed number of cores, $\mathcal{N}$. Figure 8 shows the basic elements needed to define the machine configuration. The changes made from the Cerise definitions are highlighted in red. In the new setting, the set of registers *Reg* maps a register to its content, where a register is now identified by its name and its core. In a multi-core machine, each core has its own execution state. Thus, we model the execution state of the whole machine $\mathcal{E}$ as a mapping from a core to its own execution state. Finally, a machine configuration is a tuple that describes the execution state of the machine, the registers and the shared memory.

We now define the reduction relation between configuration. Since we work in a concurrent setting, we define two reductions:

- a per-core reduction on the core i: $(\mathrm{CoreState} \times \mathrm{Reg} \times \mathrm{Mem}) \xrightarrow[core]{i} (\mathrm{CoreState} \times \mathrm{Reg} \times \mathrm{Mem})$
- a configuration reduction: $Conf \xrightarrow[conf]{} Conf$

The per-core reduction expresses an execution step on the core i. The relation corresponds to the ExecSingle relation in Cerise [7], where the registers are identified by the pair $(i, r) \in \mathcal{N} \times \mathrm{RegName}$. Appendix A describes the detailed definition of ExecSingle.

The configuration reduction is the following:

$$\frac{(s, r, m) \xrightarrow[core]{i} (s', r', m')}{(\mathcal{E}[i \mapsto s], r, m)) \xrightarrow[conf]{} (\mathcal{E}[i \mapsto s'], r', m')}$$

The configuration reduction expresses an execution step of the entire machine. The memory model that we choose for our machine is sequential consistency [10] : "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". In other words, it corresponds to all interleavings of the instructions of each core, executed sequentially. The configuration reduction encodes this interleaving. Indeed, the configuration reduction chooses non-deterministically a core i, and performs a reduction step on this core.

In order to instantiate Iris, the operational semantic needs to respect some properties. Moreover, Iris requires a language defined with expressions and values. Hence, we need to adapt the definition of the operational semantic to fit the Iris Language requirements.

Instance of the Iris Language for Concurrent Cerise Iris was originally designed to reason about high-level languages. The syntax of the language expected by Iris thus requires the usual *Val* and *Expr*. It also requires *State*, a set of *state*. The operational semantics has to describe a *core-local language*. The

$$\begin{array}{lcl} \mathit{Val} & \triangleq & \mathcal{N} \times \{\text{HaltedV} \mid \text{FailedV} \mid \text{NextIV}\} \\ \mathit{PreExpr} & \triangleq & \text{Instr } (\mathit{cs} : \text{CoreState}) \mid \text{Seq } (\mathit{pe} : \mathit{PreExpr}) \\ \mathit{Expr} & \triangleq & \mathcal{N} \times \mathit{PreExpr} \\ \mathit{State} & \triangleq & (\text{Reg} \times \text{Mem}) \end{array}$$

Exec instr

$$\frac{(\text{Exec}, \mathit{regs}, m) \xrightarrow[\mathit{core}]{i} (s', \mathit{regs}', m')}{((i, \mathit{Instr}\ \mathsf{Exec}), (\mathit{regs}, \mathit{mem})) \xrightarrow[\mathit{prim}]{} ((i, \text{Instr } s'), (\mathit{regs}', \mathit{mem}'))}$$

Seq NextI

$$((i, \text{Seq } (\text{Instr } \mathsf{NextI})), \sigma) \xrightarrow[\mathit{prim}]{} ((i, \text{Seq } (\text{Instr } \mathsf{Exec})), \sigma)$$

Seq Halted

$$((i, \text{Seq } (\text{Instr } \mathsf{Halted})), \sigma) \xrightarrow[\mathit{prim}]{} ((i, (\text{Instr } \mathsf{Halted})), \sigma)$$

Seq Failed

$$((i, \text{Seq } (\text{Instr } \mathsf{Failed})), \sigma) \xrightarrow[\mathit{prim}]{} ((i, (\text{Instr } \mathsf{Failed})), \sigma)$$

Figure 9: Instance Iris Language for Cerise

definition of the reduction relation uses the per-core reduction $\xrightarrow[\mathit{core}]{i}$ previously defined. Iris provides a machine reduction relation, where the machine is a pair of core-pool (usually named *thread-pool* in Iris) and a machine state. A machine reduction takes non-deterministically an expression in the core-pool, and performs a reduction on it.

We instantiate Iris with the language syntax[3] defined in Figure 9. The machine code does not have notion of *value* or *expression* as it exists in higher-level language. The result of an operation corresponds to the execution state of the core. A value $(i, v) \in \mathit{Val}$ describes the execution state of the core i, and so does the expression $(i, e) \in \mathit{Expr}$. Indeed, the semantic of a machine is only a sequence of transition from a state to another one. A *State* represents the memory shared between all the cores, whereas the registers are linked to a specific core. The primitive reduction step gets the core id i from the expression and performs a core reduction on the i^{th} core.

4.2 Program Logic

Cerise provides a program logic drawn from previous works in program logic and separation logic. It describes the machine state — the machine memory and the machine registers — using separation logic assertions. Because Cerise is built on top of the Iris framework, it also inherits some features such as invariants. In the following, we describe the program logic of Concurrent Cerise, based on the Cerise program logic, and we highlight the main changes.

Syntax Figure 10 presents the syntax of Concurrent Cerise. The difference with the Cerise program logic are highlighted in red. The propositions of the program logic lives in the universe of the Iris logic proposition, *iProp*. It contains the standard proposition of the higher-order logic, as well as the connective of the separation logic.

Intuitively, in separation logic, the proposition describes a set of resources (*e.g.,* the resources of the heap). The separating conjunction $P * Q$ describes the resources that can be split up into two disjoint resources P and Q. The magic wand $P \mathrel{-\!\!*} Q$ describes the resources such that, when combined with the resource P, gives the resources Q. The magic wand is also known as the *separating implication.*

[3]For the sake of readability, we omit additional details in the primitive step reduction (such as the list of observations and the forked-off list).

$$
\begin{array}{lr}
P, Q \in \mathit{iProp} ::= & \\
\quad \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \ldots & \text{higher-order logic} \\
\quad \mid P * Q \mid P -\!\!* Q \mid \lceil \varphi \rceil \mid \Box P \mid \triangleright P & \text{separation logic} \\
\quad \mid \boxed{P} & \text{invariants} \\
\quad \mid \mathsf{a} \mapsto w \mid (i, r) \Mapsto w & \text{machine resources} \\
\quad \mid \langle P \rangle \xrightarrow{i} \langle s.\, Q \rangle \mid \{P\} \overset{i}{\rightsquigarrow} \{s.\, Q\} \mid \{P\} \overset{i}{\rightsquigarrow} \bullet & \text{program logic}
\end{array}
$$

Figure 10: The syntax of our program logic.

$\lceil \varphi \rceil$ asserts that the pure proposition φ from the meta-logic holds. In Iris, assertions are a priori *non-persistent*: it may become unavailable or false later. They are additionally *non-duplicable*: $P \vdash P * P$ does not holds. A *non-persistent* assertion may be understood as a consumable resource. Some assertion can be shown to be *persistent*. The *persistent* assertions describe facts that are always true, throughout the whole execution. In this way, the Iris persistently modality $\Box P$ asserts the ownership over the *persistent part* of P : $\Box P$ is persistent and can be duplicated $\Box P \vdash \Box P * \Box P$ holds for all P. The Iris later modality $\triangleright P$ asserts that P holds after one execution step. The main use of the later modality in this work is to define recursive predicate using guarded recursion. Another feature inherited from the Iris logic is the notion of invariant. An invariant $\boxed{P}$ can turn any separation logic proposition P into a persistent proposition, via the *invariant allocation*. From the allocation, the invariant holds now and for any future execution step. In order to access the resource P, one can open the invariant. The resource is available for one atomic execution step, and requires a proof that the resource still holds after the execution step.

The logic also contains the resources of the operational semantic. The assertion $\mathsf{a} \mapsto w$ expresses the ownership of the memory address a, pointing to the machine word w. It is important to notice that the ownership of the location is *unique*, allowing to read and update the memory cell freely. The second resource of the machine is the register file. $(i, r) \Mapsto w$ expresses the ownership of the CPU register r of the core i, containing the word w. It is similar to the register points-to predicate in sequential Cerise, $r \Mapsto w$.

Program specification In addition to the resources describing the state of the machine, the logic contains assertions to specify the machine execution, *à la* Hoare triples for higher level programming languages. However, because we work on low-level machine, the code is stored in the memory. Thus, the specification can only express how the machine execution steps from a machine state to another. We define 3 types of specification:

- single execution step on the core i: $\langle P \rangle \xrightarrow{i} \langle s.\, Q \rangle$
- code fragment on the core i: $\{P\} \overset{i}{\rightsquigarrow} \{s.\, Q\}$
- complete safe execution on the core i: $\{P\} \overset{i}{\rightsquigarrow} \bullet$

P and Q are assertions of the logic, where P is the pre-condition, Q the post-condition, and s binds the core state $s \in (\mathcal{N} \times \mathcal{C})$ to the assertion Q. $\langle P \rangle \xrightarrow{i} \langle s.\, Q \rangle$ holds if, starting from the initial machine state satisfying P, satisfies the machine state Q, after one unique execution step on the core i. $\{P\} \overset{i}{\rightsquigarrow} \{s.\, Q\}$ holds if, starting from the initial machine state satisfying P, the core i either diverges, or arrives in a state satisfying Q. $\{P\} \overset{i}{\rightsquigarrow} \bullet$ holds if, starting from the initial machine state satisfying P, the core i either diverges, or terminates in the CoreState Halted or Failed.

The new features in the program logic is the identifier of the core i. Because the program logic models a concurrent machine, it requires to know which core performs the execution steps. Moreover, the adequacy theorem provided by Iris allows one to combine each specification and to reason about the entire multi-core machine.

As in sequential Cerise, the program logic satisfy the "frame rule" of separation logic, which permits to extend the specification with arbitrary resources not required by the program. It allows local reasoning.

$$
\textsc{FragFrame}\quad \frac{\{P\} \overset{i}{\rightsquigarrow} \{s.\, Q\}}{\{P * R\} \overset{i}{\rightsquigarrow} \{s.\, Q * R\}}
$$

The triples of the program logic can be composed together, as long as they relate to the same core i.

$$\textsc{SeqFrag}\quad \frac{\{P\} \overset{i}{\rightsquigarrow} \{Q\} \qquad \{Q\} \overset{i}{\rightsquigarrow} \{R\}}{\{P\} \overset{i}{\rightsquigarrow} \{R\}}$$

With the sequencing rules, one can specify the code modularly. For instance, the rule SeqFrag combines two code fragment specifications on the core i. We refer the reader to Cerise [7] to have the detailed list of rules.

Because we often reason about instructions that do not fail, we introduce the following notations.

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

The notation asserts that the post condition Q and the core is stills running.

Moreover, the interesting specification includes the points-to predicate which describes the program counter register pc. Thus, we also introduce a notation which combines the points-to predicate of the pc with the predicate P.

$$(i, w); P \triangleq (i, \mathsf{pc}) \Mapsto w * P$$

Moreover, if the context is unambiguous, the core of the pc register may be implicit. For instance:

$$\{w; P\} \overset{i}{\rightsquigarrow} \{w'; Q\} \quad \triangleq \quad \{(i, \mathsf{pc}) \Mapsto w * P\} \overset{i}{\rightsquigarrow} \{(i, \mathsf{pc}) \Mapsto w' * Q\}$$

Invariant The invariant transforms any predicate R — including non duplicable predicates — into a duplicable predicate $\boxed{R}$. The only way to access the resource R is by using the rule Inv.

$$\textsc{Inv}\quad \frac{\langle P * \triangleright R\rangle \xrightarrow{i} \langle s.\, Q * \triangleright R\rangle}{\boxed{R} \vdash \langle P\rangle \xrightarrow{i} \langle s.\, Q\rangle}$$

Opening the invariant permits one to access the resource during one atomic execution step. In particular, the invariant has to be re-establish after the execution step. The later modality is essential for the consistency of the logic, but one can usually get rid off of it.

When specifying a program, the machine resources such as the memory points-to $a \mapsto w$ means that, the current core *owns* the resources. In particular, it means that the others core does not own them, and thus cannot use them for their own specification. In a concurrent setting, one wants to share resources such as memory fragments. As the invariant is a duplicable resource, it can be shared between the different cores. In order to share a points-to memory predicate, we can encapsulate it into an invariant and share the invariant. When an instruction needs to access the resource R, we can open the invariant using the Inv rules. The invariant have to be re-established after the execution step: we have to prove that the predicate R still holds, and the resource goes back into the invariant.

4.3 Logical relation

The triples defined in the program logic allow specifying known code, but are inefficient when reasoning about unknown code, as they require to know the instruction currently executing. Therefore, Cerise includes a theorem to describe the behavior of arbitrary code. Concretely, Cerise defines a logical relation to express the safety of a machine word, built on top of the program logic. In this work, we adapt the logical relation to the new program logic, which captures the execution of a multi-core machine.

Figure 11 shows the formal definition of the new logical relation, where the changes made from Cerise are in red.

$$
\mathcal{V}(w)\ \begin{cases}
\mathcal{V}(z), \mathcal{V}(\text{O}, -, -, -) & \triangleq \text{True} \\
\mathcal{V}(\text{E}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\text{RX}, b, e, a) \\
\mathcal{V}(\text{RW}/\text{RWX}, b, e, -) & \triangleq \mathop{\ast}_{a \in [b,e)} \boxed{\exists w,\ a \mapsto w \ast \mathcal{V}(w)} \\
\mathcal{V}(\text{RO}/\text{RX}, b, e, -) & \triangleq \mathop{\ast}_{a \in [b,e)} \exists P,\ \boxed{\exists w,\ a \mapsto w \ast P(w)} \ast \triangleright \Box\, (\forall w,\ P(w) \twoheadrightarrow \mathcal{V}(w))
\end{cases}
$$

$$
\mathcal{E}(w) \quad \triangleq \quad \forall i\ reg,\ \left\{ w\ ;\ \mathop{\ast}_{\left\{((i,r),v)\ \middle|\ \substack{((i,r),v) \in reg \\ r \neq \text{pc}}\right\}} (i,r) \Mapsto v \ast \mathcal{V}(v) \right\} \overset{i}{\rightsquigarrow} \bullet
$$

Figure 11: Logical relation defining “safe to share” and “safe to execute” values.

Safe-to-share We recall that the *safe-to-share* relation $\mathcal{V}(w)$ captures the transitive accesses from the word w to other *safe-to-share* words. The definition of *safe-to-share* does not change. In Cerise, the logical relation already captures the sequentially-shared memory, because it does not involve concurrency. Thus, the main new feature that concurrency brings is the possibility for a piece of memory to be concurrently-shared between multiple core at the same time.

If a memory fragment is intended to be both sequentially-shared and concurrently-shared another core may store a non-safe capability in the shared buffer. In the program logic, the only way to concurrently-share a memory fragment is to encapsulate the memory points-to predicate in an invariant. Because the points-to predicate is not duplicable, only one invariant can encapsulate the predicate. Once the resource has been encapsulated into an invariant, the only way to access the resource is by using the Inv rule in 4.2. The invariant may contain the points-to predicate of the memory fragment, as well as a predicate describing the resource. Because we want to both sequentially-share and concurrently-share the resource, the predicates must describes the resource for each kind of sharing. In particular, the property describing a resource shared with an adversary has to be at least *safe-to-share*.

The *safe-to-share* relation defined in Cerise already captures this property. Indeed, as soon as a memory address is shared with an unknown code, it has to be *safe-to-share* for all the cores. In particular, it ensures that any value stored in the memory address have to be safe to share.

Safe-to-execute The *safe-to-execute* relation $\mathcal{E}(w)$ expresses the fact that the machine may execute safely and completely with the PC pointing to the word w, in any *safe context*. A *safe context* means that every registers of the core executing the word contains *safe-to-share* words. In a single-core setting, there is no choice about the core. However, in a multi-core setting, the word can be executed on any core. As consequence, the relation generalizes the executing core i.

5 Reasoning about unknown code in a concurrent setting

In this section, we define a method to formally prove a memory invariant at the level of the operational semantic, using the program logic, the logical relation, and the adequacy theorem. We the use this method to formally prove the two first motivating examples defined in Section 3.

5.1 Generic proof approach

First, we need to define the property $I_0 : Mem \to \mathbb{P}$ at the level of the operational semantic, and an equivalent[4] property in terms of the program logic, $I : iProp$ which will be the invariant. The method then follows two mains steps: (1) for each core, prove that it executes safely and completely in the program logic under the invariant I, and (2) use the adequacy theorem to combine all the cores and extract a theorem in terms of the operational semantic.

Executes safely and completely We may distinguish two forms of program in the first step:

[4]Equivalent regarding the state interpretation

1. The program does not involve adversary code. It suffices to use the rules of program logic to prove the specification.
2. The program involves adversary code. It requires the FTLR to conclude the proof of the specification.

We will focus on the second case, with a program involving adversary code. Consider a known program at the addresses $[\mathsf{b_p}, \mathsf{e_p})$, which jumps into an adversary code $(\textsc{rwx}, \mathsf{b_{adv}}, \mathsf{e_{adv}}, \mathsf{a_{adv}})$. Using the program logic, we define a specification of the known program

$$\boxed{I} \vdash \{(\textsc{rwx}, -, -, \mathsf{b_p}); P\} \overset{i}{\rightsquigarrow} \{(\textsc{rwx}, \mathsf{b_{adv}}, \mathsf{e_{adv}}, \mathsf{a_{adv}}); Q\}$$

where P and Q are properties that depends on the program we are specifying. The point is to get a specification of the memory fragment between the beginning of the program at $\mathsf{b_p}$, and the beginning of the adversary code at $\mathsf{a_{adv}}$.

By using the sequencing rule, if we prove

$$\boxed{I} \vdash \{(\textsc{rwx}, \mathsf{b_{adv}}, \mathsf{e_{adv}}, \mathsf{a_{adv}})); Q\} \overset{i}{\rightsquigarrow} \bullet$$

we can conclude the proof.

Because the adversary code is unknown, we have to use (the corollary of) the FTLR, which ensure that, starting from a *safe context*, the program executes safely and completely. It suffices to show that Q is a *safe context*, and apply the FTLR. Formally, a *context* is safe if all the registers of the core are safe to share. Thus, it only remains to prove

$$Q \twoheadrightarrow \mathop{\Huge\ast}_{(i,r) \in reg \setminus \{PC\}} \exists w,\ (i,r) \Mapsto w * \mathcal{V}(w)$$

Adequacy theorem As we want to prove the property at the level of the operational semantics, we define the theorem at the level of operational semantic.

Theorem 1 *Suppose an initial memory mem and an initial registers map reg, which both respects some conditions over the initial state. In particular,* $I_0(mem)$ *holds. Then, for any intermediate state* $(\mathcal{C}', reg', mem')$ *such that* $(\mathcal{C}, reg, mem) \xrightarrow[conf]{}^{*} (\mathcal{C}', reg', mem')$*, the invariant* $I_0(mem')$ *holds.*

To prove this theorem, Iris defines an adequacy theorem, which requires two conditions:

1. For each core in the core-pool, the core safely and completely executes, assuming the invariant I.

2. For each core, the property $\lceil I_0 \rceil$ holds on the resulting memory, at the level of Iris logic. In other words, the resulting state is consistent with the state interpretation and the post-condition.

We proved the former condition in the first step, and we recover the latter condition from the state interpretation of I, using the validity of the heap.

5.2 Case study — Isolation between cores

Using the method previously defined, we formally prove the integrity of the secret data in the scenario of Section 3.2.

First, we define the memory invariant and its equivalent in terms of the program logic:

$$\begin{array}{lcl} I_{buf} & \triangleq & \lambda m.\ (m(\mathsf{buf_b} + 3) = 0 \lor m(\mathsf{buf_b} + 3) = 42) \\ I & \triangleq & (\mathsf{buf_b} + 3 \mapsto 0 \lor \mathsf{buf_b} + 3 \mapsto 42) \end{array}$$

The invariant ensures the integrity of the secret memory address. If it holds, this means that the adversary will never be able to corrupt the secret. Then, using the program logic and the FTLR, we can prove that the program executes completely and safely, assuming the invariant I. The proof is straightforward.

Lemma 1

$$\boxed{\mathsf{buf_b} + 3 \mapsto 0 \lor \mathsf{buf_b} + 3 \mapsto 42} \\ \vdash \left\{ (\mathit{RWX}, \mathsf{init}, \mathsf{end}, \mathsf{init}); \begin{array}{c} (i, \mathsf{r_0}) \Mapsto w_{adv} * \\ \mathop{\Huge *}_{\left\{((i,r),v) \,\middle|\, \substack{((i,r),v) \in reg \\ r \notin \{\mathsf{pc}, \mathsf{r_0}\}}\right\}} (i, r) \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\mathsf{init}, \mathsf{end}) \mapsto \text{prog\_instrs} \end{array} \right\} \overset{i}{\rightsquigarrow} \bullet$$

The last step is to prove the property at the level of the operational semantic. As defined in the method, we use the adequacy theorem to prove the final theorem.

Theorem 2 (Adequacy for buffer isolation) *Given an initial memory mem, initial registers reg, and the following memory fragments:*

$$\begin{array}{lcll} \mathit{prog} & : & [\mathsf{b}, \mathsf{e}) & \to \text{Word} \\ \mathit{adv_1} & : & [\mathsf{b_{adv1}}, \mathsf{e_{adv1}}) & \to \text{Word} \\ \mathit{adv_2} & : & [\mathsf{b_{adv2}}, \mathsf{e_{adv2}}) & \to \text{Word} \\ \mathit{buf} & : & [\mathsf{b_{buf}}, \mathsf{e_{buf}}) & \to \text{Word} \end{array}$$

such that prog points to the instructions of the program, and buf points to the buffer.
Assuming:

1. *the initial state of memory mem satisfies: $\mathit{prog} \uplus \mathit{adv_1} \uplus \mathit{adv_2} \uplus \mathit{buf} \subseteq \mathit{mem}$*
2. *the invariant I_{buf} holds on the initial memory: $I_{buf}(\mathit{mem})$*
3. *the initial state of registers reg satisfies, :*

$$\mathit{reg}(1, \mathsf{pc}) = (\mathit{RWX}, \mathsf{b}, \mathsf{e}, \mathsf{b}), \quad \mathit{reg}(1, r_1) = (\mathit{RWX}, \mathsf{b_{adv1}}, \mathsf{e_{adv1}}, \mathsf{b_{adv1}}), \quad \mathit{reg}(1, r) \in \mathbb{Z} \textit{ otherwise}$$
$$\mathit{reg}(2, \mathsf{pc}) = (\mathit{RWX}, \mathsf{b_{adv2}}, \mathsf{e_{adv2}}, \mathsf{b_{adv2}}), \quad \mathit{reg}(2, r) \in \mathbb{Z} \textit{ otherwise}$$

Then, for any intermediate state $(\mathcal{C}', \mathit{reg}', \mathit{mem}')$ such that $(\mathcal{C}, \mathit{reg}, \mathit{mem}) \xrightarrow[conf]{}^{*} (\mathcal{C}', \mathit{reg}', \mathit{mem}')$, the invariant holds: $I_{buf}(\mathit{mem}')$.

This scenario highlights two important properties: (1) a program can passes control the unknown code, sequentially-sharing a memory buffer, and protecting a secret data — as it was already proved with Cerise ; (2) the memory of the first core is completely isolated from the memory of the second core. To summarize, this example shows that, as soon as the memory reachable from each core is pairwise disjoint, each core behaves as if it was in a single-core setting.

5.3 Case study — Shared sub-buffer

Using the method previously defined, we formally prove the integrity of the secret data in the scenario of Section 3.3.

First, we define the memory invariant and its equivalents in terms of the program logic:

$$\begin{array}{lcl} I_{buf} & \triangleq & \lambda m.\ (m(\mathsf{buf_b}) = 0 \lor m(\mathsf{buf_b}) = 42) \land (m(\mathsf{buf_b} + 1) = 0 \lor m(\mathsf{buf_b} + 1) = -42) \\ I_{secret} & \triangleq & (\mathsf{buf_b} + 3 \mapsto 0 \lor \mathsf{buf_b} + 3 \mapsto 42) \land\ (\mathsf{buf_b} + 4 \mapsto 0 \lor \mathsf{buf_b} + 4 \mapsto -42) \end{array}$$

The invariant ensures the integrity of the secret memory address. If it holds, this means that the adversary will never be able to corrupt the secret.

Moreover, since the public part of the buffer is *concurrently-shared*, we need to define an invariant. Indeed, given that the points-to predicate is not duplicable, the only way to concurrently-share the resource is to encapsulate it into an invariant:

$$I_{public} \triangleq \mathop{\Huge *}_{a \in [\mathsf{buf_b}, \mathsf{buf_b} + 3)} \exists w, a \mapsto * \mathcal{V}(w)$$

I_{public} states that the public buffer (between the memory addresses {$\mathrm{buf_b}$} and {$\mathrm{buf_b}$} + 3) contains value that always are safe to share. This is necessary because the public buffer is not only shared between the two cores, but also with adversary memory. Hence, we cannot assume a stronger property than *safe to share.*

The next step is to prove that each program completely and safely executes, under the assumption of the invariant. Using the program logic and the FTLR, the proof is straightforward.

Lemma 2 (Program 1)

$$\boxed{I_{public}} * \boxed{I_{secret}} \vdash \left\{ (RWX, \mathsf{init}_1, \mathsf{end}_1, \mathsf{init}_1); \begin{array}{l} (i, \mathsf{r}_0) \mapsto w_{adv1} * \\ \displaystyle\mathop{\huge *}_{\left\{((i,r),v) \,\middle|\, \substack{((i,r),v)\in reg \\ r \notin \{\mathsf{pc},\mathsf{r}_0\}}\right\}} (i,r) \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\mathsf{init}_1, \mathsf{end}_1) \mapsto \text{prog\_instrs1} \end{array} \right\} \overset{i}{\rightsquigarrow} \bullet$$

Lemma 3 (Program 2)

$$\boxed{I_{public}} * \boxed{I_{secret}} \vdash \left\{ (RWX, \mathsf{init}_2, \mathsf{end}_2, \mathsf{init}_2); \begin{array}{l} (i, \mathsf{r}_0) \mapsto w_{adv2} \\ \displaystyle\mathop{\huge *}_{\left\{((i,r),v) \,\middle|\, \substack{((i,r),v)\in reg \\ r \notin \{\mathsf{pc},\mathsf{r}_0\}}\right\}} (i,r) \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\mathsf{init}_2, \mathsf{end}_2) \mapsto \text{prog\_instrs2} \end{array} \right\} \overset{i}{\rightsquigarrow} \bullet$$

The last step is to prove the property at the bare-level of the operational semantic, *i.e.,* the memory invariant I_{buf} holds at each execution step. As defined in the methodology, we use the adequacy theorem to prove the final theorem.

Theorem 3 (Adequacy shared buffer) *Given an initial memory mem, initial registers reg, and the following memory fragments:*

$$\begin{array}{llll} prog_1 & : & [\mathsf{b}_1, \mathsf{e}_2) & \to \text{Word} \\ prog_2 & : & [\mathsf{b}_2, \mathsf{e}_2) & \to \text{Word} \\ adv_1 & : & [\mathsf{b}_{\mathsf{adv1}}, \mathsf{e}_{\mathsf{adv1}}) & \to \text{Word} \\ adv_2 & : & [\mathsf{b}_{\mathsf{adv2}}, \mathsf{e}_{\mathsf{adv2}}) & \to \text{Word} \\ buf & : & [\mathsf{b}_{\mathsf{buf}}, \mathsf{e}_{\mathsf{buf}}) & \to \text{Word} \end{array}$$

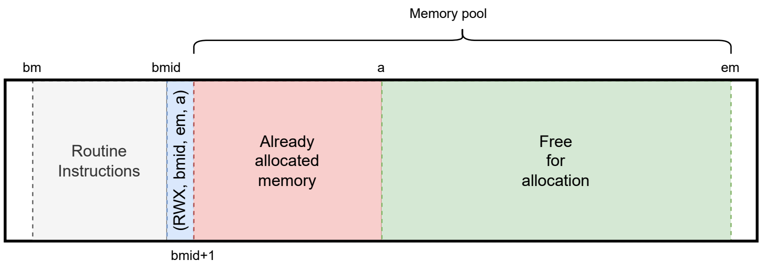
such that $prog_1$ points to the instructions of the first program, $prog_2$ points to the instructions of the second program and buf points to the buffer.

1. *the initial state of memory mem satisfies: $prog_1 \uplus prog_2 \uplus adv_1 \uplus adv_2 \uplus buf \subseteq mem$*
2. *the invariant I_{buf} holds on the initial memory: $I_{buf}(mem)$*
3. *the initial state of registers reg satisfies, :*

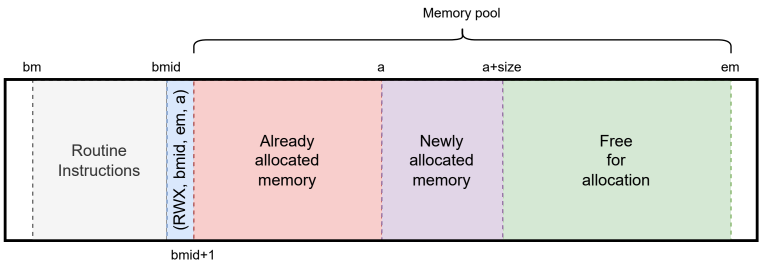
$$\begin{array}{lll} reg(1, \mathsf{pc}) = (RWX, \mathsf{b}_1, \mathsf{e}_1, \mathsf{b}_1), & reg(1, r_0) = (RWX, \mathsf{b}_{\mathsf{adv1}}, \mathsf{e}_{\mathsf{adv1}}, \mathsf{b}_{\mathsf{adv1}}), & reg(1, r) \in \mathbb{Z} \textit{ otherwise} \\ reg(2, \mathsf{pc}) = (RWX, \mathsf{b}_2, \mathsf{e}_2, \mathsf{b}_2), & reg(2, r_0) = (RWX, \mathsf{b}_{\mathsf{adv2}}, \mathsf{e}_{\mathsf{adv2}}, \mathsf{b}_{\mathsf{adv2}}), & reg(2, r) \in \mathbb{Z} \textit{ otherwise} \end{array}$$

Then, for any intermediate state $(\mathcal{C}', reg', mem')$ such that $(\mathcal{C}, reg, mem) \xrightarrow[conf]{}^{*} (\mathcal{C}', reg', mem')$, the invariant holds $I_{buf}(mem')$.

In this scenario, the secret buffer is concurrently-shared between the cores, and the integrity of the secret data holds. Moreover, the public buffer is both sequentially-shared and concurrently-shared. This scenario complements the previous one, by adding concurrent-sharing of the buffer. Moreover, this example highlights the similarity in reasoning about a buffer concurrently-shared with an adversary and a buffer sequentially-shared, which support desired the orthogonality of the concurrency with the integrity property of the secret data.



(a) Malloc before



(b) Malloc after - Single-core

Figure 12: Unsafe malloc

6 Synchronization and concurrent-safe memory allocation

The last scenario, presented in Section 3.4, requires a concurrently-safe implementation of a dynamic memory allocation macro. In this section, we explain why the *malloc* macro presented in Cerise [7] is not concurrently-safe. In order to propose a new implementation of the *malloc* macro, safe regarding the concurrent calls, we extend the model of Cerise with a *Compare-And-Swap* (CAS) instruction, and implement a synchronization mechanism with it, a spinlock.

6.1 Motivation - malloc

In this section, we describe the behavior of the original *malloc* macro of Cerise. From a high-level point of view, we explain why the macro is safe to use in a single-core setting, but becomes unsafe in a multi-core setting. More precisely, we show that, in a multi-core setting, the macro can allocate the same memory fragment multiple times. Therefore, multiple callers, including potentially an adversary, may have the authority over the newly allocated memory region.

The implementation of *malloc* in Cerise is a simple bump-pointer allocator: the routine allocates memory from a given memory pool. An internal pointer points to the first address available in the memory pool to keep track of the allocated memory. At each new allocation, the macro updates the internal pointer. The macro provides no way to de-allocate the memory. The *malloc* macro can be represented by the mapping of the memory showed in Figure 12a. The routine part contains the instructions of the macro. It checks the argument of the call, allocates the memory by deriving a capability from the internal pointer, updates the internal pointer, and returns the capability of the newly allocated memory. The *memory-pool* contains the memory already allocated by the previous call, and a pool of memory addresses available for the future allocations. The address bmid contains a capability giving the authority over the whole memory-pool, and points to the first address available a. When the *malloc* macro is called with the argument *size*, the routine allocates a new area from a to $a + size$, and derives capability of the allocated region from the internal pointer bmid. Figure 12b shows the new memory mapping, after the call.

The new *first address available* becomes $a + size$. In this way, the future call will start to allocate the region from the address $a + size$, and will therefore not overlap the previously allocated memory region. As long as the calls to the *malloc* are sequential, the *first address available* is always updated in between a call and another. Hence, in a sequential setting, the *malloc* never gives the same memory

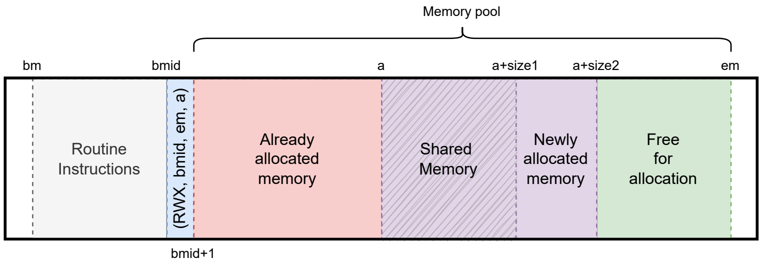


Figure 13: Malloc after - Multi-core

i	$[\![i]\!](\varphi, j)$	Conditions
`cas` r_1 r_2 r_3	if $w = w_2$, then updPC(φ[mem.$a \mapsto \varphi$.reg(j, r_3))][reg.(j, r_2) $\mapsto w$], j) else updPC(φ[reg.(j, r_2) $\mapsto w$], j)	φ.reg((j, r_1)) $= (p, b, e, a)$ and $b \leq a < e$ and $p \in \{\text{RW}, \text{RWX}\}$ and $w =$ getWord(φ.mem(a), j) and $w_2 =$ getWord(φ.reg((j, r_2)), j) and $w_3 =$ getWord(φ.reg((j, r_3)), j)

Figure 14: Operational semantics of Compare-And-Swap.

area at different calls, and a caller can safely use its allocated region.

However, in a concurrent setting, different cores can call the macro concurrently — at the same time — leading to an unexpected, and undesirable share of allocated memory between the callers. The problem is that the *first address available* is always not updated between a load and another (concurrent) one. Thus, multiple concurrent call may load the same first address available a, and allocate the two memory areas $[a, a + \textit{size}_1)$ and $[a, a + \textit{size}_2)$. Therefore, the callers shares the addresses $[a, a + \textit{min}(\textit{size}_1, \textit{size}_2)) \neq \emptyset$

Figure 13 shows a mapping of the memory after the calls in the multi-core setting. This *malloc* macro is therefore not safe anymore in a multi-core setting. An adversary code could call the *malloc* at the same time as a verified code, obtains a memory buffer shared with the verified code, and compromises the integrity of its content.

In order to prevent these attacks, we need to add a synchronization mechanism that blocks a caller if the macro is already executing on another core. A convenient way is to make a *mutual exclusion* over the internal capabilities. A convenient way is to use a *lock mechanism* at the beginning of the routine, such as a *spinlock*. The lock ensures the *mutual exclusion* of the internal capability. Therefore, the internal capability is always up-to-date in between a load and another, even with concurrent calls.

6.2 Compare-And-Swap instruction

To implement a *spinlock* mechanism, we firstly need a new instruction interacting with the memory with some kind of atomicity, such as *Compare-And-Swap* (CAS) or *Compare-And-Set*. The Morello architecture supports the CAS with capabilities [1, section 4.4.21]. The CAS instructions compares the value of a *comparison register* with the content of a memory address. If the values are equal, the new value is written in the memory address. If they are different, the memory is not modified. In either case, the old value contained in the memory address is loaded in the *comparison register*. We can then deduce that, if the value in the register has been modified, the comparison has failed and the memory has not changed.

Figure 14 describes the operational semantic of the CAS instruction, complete completing the operational semantic defined in Appendix A.

6.3 Spinlock

We propose a macro that implements a spinlock. The principle of the spinlock is to repeatedly try to acquire the lock until it succeed. The user needs to provide the capability pointing to the memory address containing the state of the lock.

```
; r0 -> capability pointing to the lock state
init:
mov r1 1        ; value when the lock is acquired
mov r2 PC
lea r2 2        ; r2 -> label loop
loop:
mov r3 0
cas r0 r3 r1   ; atomically compare the value of the lock, and swap if unlocked
jnz r2 r3      ; if the lock was free, then continue , else jump to loop
end:
```

Figure 15: Code of the macro to acquire the static spinlock

Concretely, the lock state can have two states, encoded by 0 or 1. If the lock state contains 0, the lock is free and can be acquired. On the other hand, if the lock state contains 1, the lock is unavailable: another core already owns the lock.

Figure 15 shows the instructions to acquire a spinlock. It first prepares the registers and the waiting loop. When all the registers are ready, it performs a CAS with the lock state. There are two possible scenarios:

1. The lock state is available (contains 0). Then the *CAS* performs the swap and acquire the lock (store 1), in one atomic execution step. The register r_3 now contains the previous value of the lock (0). Because the previous value is 0, we know that the lock was previously free, and we can continue to execute the code (we do not perform the jump to the loop).

2. The lock state was already locked (contains 1). Then the *CAS* does not perform the swap, but still loads the previous value of the lock in r_3 (which is 1). Because the previous value is 1, we know that the lock was not free, and we have to wait. Thus, we jump back to the loop, restore the value of r_3 and continue to try to take the lock until we acquire it.

Releasing the lock simply stores 0 in the lock state. The *CAS* is not necessary, because the lock is already owned by this core and no other core can concurrently modify its state.

In terms of the program logic, we use the following specifications for lock

$$\boxed{[\mathsf{a}_{\mathsf{acquire}}, \mathsf{e}_{\mathsf{acquire}}) \mapsto instrs_{acquire} * is\_lock\ \gamma\ \mathsf{a}_{\mathsf{lock}}\ \ P}$$
$$\vdash \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{a}_{\mathsf{acquire}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ (i, \mathsf{r}_1) \Mapsto w_1 * (i, \mathsf{r}_2) \Mapsto w_2 * \\ (i, \mathsf{r}_3) \Mapsto w_3 \end{array} \right\} \overset{i}{\rightsquigarrow} \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{acquire}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ (i, \mathsf{r}_1) \Mapsto 1 * (i, \mathsf{r}_2) \Mapsto w_{2'} * \\ (i, \mathsf{r}_3) \Mapsto 2 * P * locked\ \gamma \end{array} \right\}$$

and for unlock

$$\boxed{[\mathsf{a}_{\mathsf{acquire}}, \mathsf{e}_{\mathsf{acquire}}) \mapsto instrs_{acquire} * is\_lock\ \gamma\ \mathsf{a}_{\mathsf{lock}}\ \ P}$$
$$\vdash \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{a}_{\mathsf{release}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ P * locked\ \gamma \end{array} \right\} \overset{i}{\rightsquigarrow} \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{release}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \end{array} \right\}$$

is_lock is an abstract, persistent predicate. It describes how to manage and protect the resources described by P, locked by a_{lock}. The name γ is an identifier for the lock. *locked* is an abstract, non duplicable token. It indicates the current owner of the lock γ. Because it is non duplicable, while one core owns the token *locked* γ, no other core can access to the resource P. *Acquiring* the lock gives the ownership of the protected resource P and the token *locked* γ. To release a lock, one need to give back the resource P and the token *locked* γ.

For a concrete implementation of the predicates *is_lock* and *locked*, we refer the reader to Appendix B.

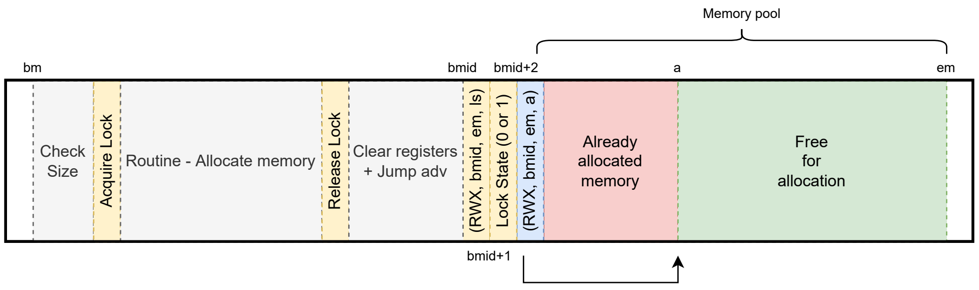


Figure 16: Safe malloc memory map

6.4 Concurrent-safe malloc

As explained in Section 6.1, the version of the *malloc* macro in (sequential) Cerise is not safe in a concurrent setting. Indeed, a part of the routine should be in a critical section: from the load of the internal capability (at bmid) to the update of the capability, after the allocation of the memory. We propose a concurrently-safe implementation of the *malloc* macro, using a spinlock. The macro acquires the lock before loading the sensitive capability, and releases it only after the update of the capability in the memory. In this way, it ensures that two concurrent calls cannot load the same capability and allocate the same memory fragment. Since the new *malloc* uses a spinlock, it needs to store the lock state, as well as the capability allowing to modify it. Figure 16 shows the mapping of the concurrently-safe *malloc* macro. The new parts of the macros are the yellow ones.

This mapping is formally encoded by the following invariant:

$$\text{mallocInv}(\mathsf{b_m}, \mathsf{e_m}) \triangleq \boxed{\begin{array}{l} [\mathsf{b_m}, \mathsf{b_{mid}}) \mapsto \text{malloc\_instrs} \ast \\ \mathsf{b_{mid}} \mapsto (\text{RWX}, \mathsf{b_m}, \mathsf{e_m}, \mathsf{b_{mid}} + 1) \ast \\ \textit{is\_lock}\ \gamma\ (\mathsf{b_{mid}} + \mathit{1}) \ast \\ \left(\begin{array}{ll} \exists \mathsf{a}, & \mathsf{b_{mid}} + 2 \mapsto (\text{RWX}, \mathsf{b_{mid}}, \mathsf{e_m}, \mathsf{a}) \ast \\ & [\mathsf{a}, \mathsf{e_m}) \mapsto [0 \cdots 0] \ast \\ & \lceil \mathsf{b_{mid}} + 2 < \mathsf{a} \leq \mathsf{e_m} \rceil \end{array} \right) \end{array}}$$

where $\mathsf{b_{mid}} = \mathsf{b_m} + \textit{length}\ \text{malloc\_instrs}$

6.5 Scenario involving malloc

In this section, we prove the property of the scenario involving the *malloc* defined in section 3.4.

We recall that a core puts a secret data in a dynamically allocated memory buffer, and then asserts the integrity of the data. As the memory address is not statically known, we use the *assert* macro to check the secret data. *assert* uses a flag at the statically known address $\mathsf{a_{flag}}$, which is modified if the check fails.

We use the methodology previously defined to formally prove the integrity of the *assert flag* at the level of the operational semantic. As the code uses the dynamic assertion, the memory invariant relies on the assertion flag $\mathsf{a_{flag}}$. We define the memory invariant and its equivalent in terms of the program logic:

$$\begin{array}{lcl} I_{\mathit{flag}} & \triangleq & \lambda m.\ m(\mathsf{a_{flag}}) = 0 \\ I & \triangleq & \mathsf{a_{flag}} \mapsto 0 \end{array}$$

The invariant ensures the integrity of the assertion flag. Therefore, it ensures the integrity of the dynamically allocated memory cell. If it holds, this means that the adversary will never be able to corrupt the secret. Then, using the program logic, we can prove that the program executes completely and safely, assuming the invariant I. The program has access to a linking table, containing a capability pointing to the *malloc* macro and a capability pointing to the *assert* macro. Using the program logic, it is straightforward to show the following specification:

$$\begin{array}{l}\forall \mathit{reg}, \\ \boxed{\text{mallocInv}(\mathsf{b_m},\mathsf{e_m})}, \boxed{\text{assertInv}(\mathsf{b_a},\mathsf{e_a},\mathsf{a_{flag}})}, \boxed{\mathsf{a_{flag}} \mapsto 0} \\ \vdash \left\{ (\text{RWX},\mathsf{b},\mathsf{e},\mathsf{b}); \begin{array}{l} \displaystyle\mathop{\Huge *}_{\left\{((i,r),v) \,\middle|\, \substack{((i,r),v)\in \mathit{reg} \\ r\neq\{\mathsf{pc},\mathsf{r_0}\}}\right\}} (i,r) \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \displaystyle\mathop{\Huge *}_{\substack{(a,w)\in \mathit{prog}, \\ a\notin\{\mathsf{data},\mathsf{table},\mathsf{table}+1\}}} a \mapsto w * \\ \mathsf{data} \mapsto (\text{RO},\mathsf{table},\mathsf{table}+2,\mathsf{table}) * \\ \mathsf{table} \mapsto (\text{E},\mathsf{b_m},\mathsf{e_m},\mathsf{b_m}) * \\ \mathsf{table}+1 \mapsto (\text{E},\mathsf{b_a},\mathsf{e_a},\mathsf{b_a}) * \end{array} \right\} \overset{i}{\rightsquigarrow} \bullet \end{array}$$

Because the program does not involve adversary code, the FTLR is not necessary to prove this specification. Indeed, the core halts after the dynamic assertion.

The last step is to prove the property at the level of the operational semantic, *i.e.,* the memory invariant I_{flag} holds at each execution step. As defined in the method, we use the adequacy theorem to prove the final theorem.

Theorem 4 (Adequacy concurrent dynamic allocation) *Given an initial memory mem, initial registers reg, and the following memory fragments:*

$$\begin{array}{llll} \mathit{prog} & : & [\mathsf{b},\mathsf{e}) & \rightarrow \text{Word} \\ \mathit{malloc} & : & [\mathsf{b_m},\mathsf{e_m}) & \rightarrow \text{Word} \\ \mathit{assert} & : & [\mathsf{b_a},\mathsf{e_a}) & \rightarrow \text{Word} \\ \mathit{adv} & : & [\mathsf{b_{adv}},\mathsf{e_{adv}}) & \rightarrow \text{Word} \end{array}$$

such that prog points to the instructions of the known program.
Assuming:

1. *the initial state of memory mem satisfies:* $\mathit{prog} \uplus \mathit{malloc} \uplus \mathit{assert} \uplus \mathit{adv} \subseteq \mathit{mem}$
2. $[\mathsf{b_m},\mathsf{e_m})$ *contains the concurrently-safe malloc routine*
3. $[\mathsf{b_a},)e_a$ *contains the assert routine and the flag assertion* $\mathsf{a_{flag}}$
4. *the invariant* I_{flag} *holds on the initial memory:* $I_{flag}(\mathit{mem}) \triangleq \mathit{mem}(\mathsf{a_{flag}}) = 0$
5. *prog contains a table linking to malloc and assert:*

$$\begin{aligned} \exists \mathsf{data},\mathsf{table}, \mathit{mem}(\mathsf{data}) &= (\mathit{RO},\mathsf{table},\mathsf{table}+2,\mathsf{table}) \\ \mathit{mem}(\mathsf{table}) &= (\mathit{E},\mathsf{b_m},\mathsf{e_m},\mathsf{b_m}) \\ \mathit{mem}(\mathsf{table}+1) &= (\mathit{E},\mathsf{b_a},\mathsf{e_a},\mathsf{b_a}) \end{aligned}$$

6. *the adversary region contains no capabilities except for a table linking to malloc:*

$$\begin{array}{l} \exists \mathsf{data_{adv}},\mathsf{table_{adv}}, \forall a \in \text{dom}(\mathit{adv}) \backslash \{\mathsf{data_{adv}},\mathsf{table_{adv}}\}, \\ \qquad \mathit{adv}(a) \in \mathbb{Z} \\ \qquad \mathit{adv}(\mathsf{data_{adv}}) = \ (\mathit{RO},\mathsf{table_{adv}},\mathsf{table_{adv}}+1,\mathsf{table_{adv}}) \\ \qquad \mathit{adv}(\mathsf{table_{adv}}) = \ (\mathit{E},\mathsf{b_m},\mathsf{e_m},\mathsf{b_m}) \end{array}$$

7. *the initial state of registers reg satisfies, :*

$$\begin{array}{rl} \mathit{reg}(1,\mathsf{pc}) = (\mathit{RWX},\mathsf{b},\mathsf{e},\mathsf{b}), & \mathit{reg}(1,r) \in \mathbb{Z} \text{ otherwise} \\ \mathit{reg}(2,\mathsf{pc}) = (\mathit{RWX},\mathsf{b_{adv}},\mathsf{e_{adv}},\mathsf{b_{adv}}), & \mathit{reg}(2,r) \in \mathbb{Z} \text{ otherwise} \end{array}$$

Then, for any intermediate state $(\mathcal{C}',\mathit{reg}',\mathit{mem}')$ *such that* $(\mathcal{C},\mathit{reg},\mathit{mem}) \xrightarrow[\mathit{conf}]{}^{*} (\mathcal{C}',\mathit{reg}',\mathit{mem}')$*, the invariant holds* $I_{flag}(\mathit{mem}') \triangleq \mathit{mem}'(\mathsf{a_{flag}}) = 0$.

This scenario highlights the possibility to concurrently-share libraries, in particular dynamic allocation. It requires a more complete model than the previous examples, in particular involving synchronization mechanisms. In addition to the integrity properties of a secret buffer, this example emphasizes the concurrent reasoning.

7 Related Work

There are two main line of work on using formal methods to prove security properties of CHERI-enabled architectures. The first line of work targets the reachability and monotonicity properties of the capabilities for a complete ISA, with mechanized and automated proofs method to face future design evolution.

Nienhuis et al. [11] propose a full specification of CHERI-MIPS [17]. The architecture provides 180-odd instructions, described in 6k LoS. They propose a lightweight formal method to formalize and reason about the security properties on CHERI architectures. They thus proved properties of the reachability and monotonicity of the capabilities in the CHERI-MIPS architecture. Their approach is the following: they express the specification using an instruction description language, *e.g.,* L3 or Sail, which automatically generates the definition of the operational semantic for theorem provers. Then, they define the intended properties over these definitions, in terms of basic operation over sets and quantified formulas. They can finally prove the security properties. Similarly, *Bauereiss et al.* [5] propose a full specification of the Morello ISA [1]. Their approach is different from *Nienhuis et al.*, allowing to reason on x10 scale architecture — 409 instructions, described by more than 62k LoS. They provide a proof of the reachable capability monotonicity for the complete Morello ISA. They factorize their proof via an abstraction that captures the essential properties of the CHERI ISA, expressed above a monadic intra-instruction semantic.

The seconde line of work, the Cerise project [7, 15, 9], explores more complex security properties, involving interaction between known code and unknown code such as the stack safety and the calling convention. They develop a program logic to reason about known code, and a logical relation to reason about unknown code and capture semantic properties of the language. *Strydonck et al.* [16] propose a verification approach for a full-system security, under multiple attackers model, and instantiate their approach by extending Cerise with support for memory-mapped I/O. However, Cerise makes strong assumptions, and thus reasons on an idealized architecture, rather than a full and realistic one.

Both of these line of work consider only the sequential part of the capability architecture. They consider the security properties studied orthogonal to the concurrency. To our knowledge, Concurrent Cerise is the first work studying programs involving the concurrency for CHERI-like machine with formal methods. We consider only a sequentially consistent memory model.

8 Future directions

Concurrent Cerise is a first step to close the gap between the formal model and a real-world capability machine. However, the sequentially consistent memory model is a significant simplification of the real behavior of a multi-core machine. Moreover, concurrency is only one way to extend Cerise toward a more realistic model. If we want strong guarantees about real-world capability machine, we need to continue to bridge the gap, by extending the ideal model of Cerise into a more realistic one. To pursue this goal, we propose several possible directions in which to improve the Cerise model:

- **Concurrency** — Cerise models a single-core CPU. Real-world processors are multi-core. For instance, the current Morello prototype board has 4 cores and exhibits relaxed concurrency. In this work, we extended Cerise with a simple form of concurrency by interleaving. However, this memory model is much simpler than what happens in real-world processors. The Armv8-A or RISC-V relaxed-memory models for priviliged level are currently an active topic of research, and we propose to explore how to make Cerise sound with respect to it, and to investigate how this affects the security properties.

- **Complete model regarding the hardware** — Cerise models a small subset of a real-world architecture. For instance, there is more than 10k pages in the Armv8-A documentation, which is very far from the ~20 instructions in the Cerise model. It is challenging to deal with real-world ISA using formal methods. Past work proposed solutions, *e.g.,* factorizing instructions with similar behavior. We propose to explore solutions that allow one to reason on real-world ISA in Cerise.

- **New security properties** — The main security focus in CHERI was to ensure the monotonicity of the capabilities. Cerise enriched that with more expressive capabilities to enforce stack safety. However, these offer, yet unverified security properties. For instance, security properties involving interrupts and exception handlers. We propose to define new security properties and to extend the Cerise model in a way that captures these new security properties.

References

[1] Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture. page 1294.

[2] CHERI. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[3] Iris Project. https://iris-project.org/.

[4] Morello Program. https://www.arm.com/architecture/cpu/morello.

[5] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N M Watson, and Peter Sewell. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. page 30.

[6] Thomas Gavin. A proactive approach to more secure code – Microsoft Security Response Center.

[7] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. 1(1):55.

[8] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–30, January 2021.

[9] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities. 1:35.

[10] Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[11] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M Norton, Simon W Moore, Peter G Neumann, Ian Stark, Robert N M Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. page 18.

[12] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of Machine Code Against Authoritative ISA Semantics. *San Diego*, page 16, 2022.

[13] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Munich, Germany, Proceedings*, pages 143–173, April 2022.

[14] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures. In Peter Müller, editor, *Programming Languages and Systems*, volume 12075, pages 626–655. Springer International Publishing, Cham, 2020.

[15] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42(1):1–53, March 2020.

[16] Thomas Van Strydonck, KU Leuven, Aına Linn Georges, Armael Gueneau, and Alix Trieu. Proving full-system security properties under multiple attacker models on capability machines. page 16.

[17] Robert N M Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). page 590.

A Operational semantic single instruction

$$\textsc{ExecSingle} \quad ((j, \mathsf{Running}), \varphi) \to \begin{cases} [\![decode(z)]\!](\varphi, j) & \text{if } \varphi.\mathrm{reg}((j, \mathsf{pc})) = (p,b,e,a) \wedge b \le a < e \wedge \\ & \quad p \in \{\mathrm{RX}, \mathrm{RWX}\} \wedge \varphi.\mathrm{mem}(a) = z \\ ((j, \mathsf{Failed}), \varphi) & \text{otherwise} \end{cases}$$

i	$[\![i]\!](\varphi, j)$	Conditions
`fail`	$((j, \mathsf{Failed}), \varphi)$	
`halt`	$((j, \mathsf{Halted}), \varphi)$	
`mov` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho, j)$
`load` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto w], j)$	$\varphi.\mathrm{reg}((j, r_2)) = (p,b,e,a)$ and $w = \varphi.\mathrm{mem}(a)$ and $b \le a < e$ and $p \in \{\mathrm{RO}, \mathrm{RX}, \mathrm{RW}, \mathrm{RWX}\}$
`store` r ρ	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w], j)$	$\varphi.\mathrm{reg}((j, r)) = (p,b,e,a)$ and $b \le a < e$ and $p \in \{\mathrm{RW}, \mathrm{RWX}\}$ and $w = \mathrm{getWord}(\varphi, \rho, j)$
`jmp` r	$((j, \mathsf{Running}),$ $\varphi[\mathrm{reg}.(j, \mathsf{pc}) \mapsto newPc])$	$newPc = \mathrm{updatePcPerm}(\varphi.\mathrm{reg}((j, r)))$
`jnz` r_1 r_2	if $\varphi.\mathrm{reg}(j, r_2) \neq 0$, then $((j, \mathsf{Running}),$ $\varphi[\mathrm{reg}.(j, \mathsf{pc}) \mapsto newPc])$ else $\mathrm{updPC}(\varphi, j)$	$newPc = \mathrm{updatePcPerm}(\varphi.\mathrm{reg}((j, r_1)))$
`restrict` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto w])$	$\varphi.\mathrm{reg}((j, r)) = (p,b,e,a)$ and $p' = \mathrm{decodePerm}(\mathrm{getWord}(\varphi, \rho, j))$ and $p' \preccurlyeq p$ and $w = (p', b, e, a)$
`subseg` r ρ_1 ρ_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto w])$	$\varphi.\mathrm{reg}((j, r)) = (p,b,e,a)$ and for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i, j)$ and $z_i \in \mathbb{Z}$ and $b \le z_1$ and $0 \le z_2 \le e$ and $p \neq \mathrm{E}$ and $w = (p, z_1, z_2, a)$
`lea` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto w])$	$\varphi.\mathrm{reg}((j, r)) = (p,b,e,a)$ and $z = \mathrm{getWord}(\varphi, \rho, j)$ and $p \neq \mathrm{E}$ and $w = (p, b, e, a+z)$
`add` r ρ_1 ρ_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i, j)$ and $z_i \in \mathbb{Z}$ and $z = z_1 + z_2$
`sub` r ρ_1 ρ_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i, j)$ and $z_i \in \mathbb{Z}$ and $z = z_1 - z_2$
`lt` r ρ_1 ρ_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r) \mapsto z])$	for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i, j)$ and $z_i \in \mathbb{Z}$ and if $z_1 < z_2$ then $z = 1$ else $z = 0$
`getp` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto z])$	$\varphi.\mathrm{reg}((j, r_2)) = (p, \_, \_, \_)$ and $z = \mathrm{encodePerm}(p)$
`getb` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto b])$	$\varphi.\mathrm{reg}((j, r_2)) = (\_, b, \_, \_)$
`gete` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto e])$	$\varphi.\mathrm{reg}((j, r_2)) = (\_, \_, e, \_)$
`geta` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto a])$	$\varphi.\mathrm{reg}((j, r_2)) = (\_, \_, \_, a)$
`isptr` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.(j,r_1) \mapsto z])$	if $\varphi.\mathrm{reg}((j, r_2)) = (\_, \_, \_, \_)$ then $z = 1$ else $z = 0$
$\_$	$((j, \mathsf{Failed}), \varphi)$	otherwise

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

$$\mathrm{getWord}(\varphi, \rho, j) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{if } \rho \in (\mathcal{N} \times \mathrm{RegName}) \end{cases}$$

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

Figure 17: Operational semantics of Concurrent Cerise: execution of a single instruction. Changes from (sequential) Cerise are highlighted in red.

B Concrete implementation lock predicates

We recall the specifications for acquire

$$\boxed{[\mathsf{a}_{\mathsf{acquire}}, \mathsf{e}_{\mathsf{acquire}}) \mapsto \mathit{instrs}_{\mathit{acquire}} * \mathit{is\_lock}\ \gamma\ \mathsf{a}_{\mathsf{lock}}\ \ P}$$

$$\vdash \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{a}_{\mathsf{acquire}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ (i, \mathsf{r}_1) \Mapsto w_1 * (i, \mathsf{r}_2) \Mapsto w_2 * \\ (i, \mathsf{r}_3) \Mapsto w_3 \end{array} \right\} \overset{i}{\rightsquigarrow} \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{acquire}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ (i, \mathsf{r}_1) \Mapsto 1 * (i, \mathsf{r}_2) \Mapsto w_{2'} * \\ (i, \mathsf{r}_3) \Mapsto 2 * P * \mathit{locked}\ \gamma \end{array} \right\}$$

and for release

$$\boxed{[\mathsf{a}_{\mathsf{acquire}}, \mathsf{e}_{\mathsf{acquire}}) \mapsto \mathit{instrs}_{\mathit{acquire}} * \mathit{is\_lock}\ \gamma\ \mathsf{a}_{\mathsf{lock}}\ \ P}$$

$$\vdash \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{a}_{\mathsf{release}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \\ P * \mathit{locked}\ \gamma \end{array} \right\} \overset{i}{\rightsquigarrow} \left\{ \begin{array}{l} (i, PC) \Mapsto (\text{RWX}, \mathsf{b}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{pc}}, \mathsf{e}_{\mathsf{release}}) * \\ (i, \mathsf{r}_0) \Mapsto (p_{lock}, \mathsf{b}_{\mathsf{lock}}, \mathsf{e}_{\mathsf{lock}}, a_{\mathsf{lock}}) * \end{array} \right\}$$

The specification is inspired by the lock for Heaplang, in the Iris Lecture Notes (available on the Iris Project website).

locked is an abstract, non duplicable token. It indicates the current owner of the lock γ. Because it is non duplicable, while one core owns the token *locked* γ, no other core can access to the resource P. We define *locked* γ is the ghost state of exclusive resource algebra, such that $\mathit{locked}\ \gamma * \mathit{locked}\ \gamma \Rightarrow \mathit{False}$.

is_lock is an abstract predicate. It describes how to manage and protect the resources described by P, locked by a_{lock}. The name γ is an identifier for the lock. The *is_lock* predicate is defined as follows:

$$\mathit{is\_lock}\ \gamma\ \mathsf{a}_{\mathsf{lock}}\ \ P \triangleq (\mathsf{a}_{\mathsf{lock}} \mapsto 0 * P * \mathit{locked}\ \gamma) \vee (\mathsf{a}_{\mathsf{lock}} \mapsto 1)$$

The predicate expresses the fact that, the lock is either unlocked, meaning that the lock state $\mathsf{a}_{\mathsf{lock}}$ is set at 1, and thus it owns the resources P and the *locked* token ; or is locked, meaning that the lock state $\mathsf{a}_{\mathsf{lock}}$ is set at 0, and thus it is the core that owns the lock which also owns the resource P and the *locked* token. It is important to notice that, if a core owns the lock, it cannot be re-opened: otherwise, it would means that there is two tokens *locked*, which is invalid.