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

AÏNA LINN GEORGES, Aarhus University, Denmark
ARMAËL GUÉNEAU, Aarhus University, Denmark
THOMAS VAN STRYDONCK, KU Leuven, Belgium
AMIN TIMANY, Aarhus University, Denmark
ALIX TRIEU, Aarhus University, Denmark
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

A capability machine is a type of CPU allowing fine-grained privilege separation using *capabilities*, machine words that represent certain kinds of authority. We present a mathematical model and accompanying proof methods that can be used for formal verification of functional correctness of programs running on a capability machine, even when they invoke and are invoked by unknown (and possibly malicious) code. We use a program logic called Cerise for reasoning about known code, and an associated logical relation, for reasoning about unknown code. The logical relation formally captures the capability safety guarantees provided by the capability machine. The Cerise program logic, logical relation, and all the examples considered in the paper have been mechanized using the Iris program logic framework in the Coq proof assistant.

The methodology we present underlies recent work of the authors on formal reasoning about capability machines [Georges et al. 2021; Skorstengaard et al. 2019a; Van Strydonck et al. 2021], but was left somewhat implicit in those publications. In this paper we present a pedagogical introduction to the methodology, in a simpler setting (no exotic capabilities), and starting from minimal examples. We work our way up to new results about a heap-based calling convention and implementations of sophisticated object-capability patterns of the kind previously studied for high-level languages with object-capabilities, demonstrating that the methodology scales to such reasoning.

ACM Reference Format:
Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2021. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. 1, 1 (October 2021), 55 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A capability machine is a type of CPU that enables fine-grained memory compartmentalization and privilege separation through the use of *capabilities*. This type of hardware architecture has been studied since the 60ies [Dennis and Van Horn 1966; Levy 1984], and in particular more recently as part of the CHERI project [Watson et al. 2020]. Capability machines offer fine-grained and scalable

Authors' addresses: Aïna Linn Georges, ageorges@cs.au.dk, Aarhus University, Denmark; Armaël Guéneau, armael@cs.au.dk, Aarhus University, Denmark; Thomas Van Strydonck, thomas.vanstrydonck@cs.kuleuven.be, KU Leuven, Belgium; Amin Timany, timany@cs.au.dk, Aarhus University, Denmark; Alix Trieu, alix.trieu@cs.au.dk, Aarhus University, Denmark; Dominique Devriese, dominique.devriese@vub.be, Vrije Universiteit Brussel, Belgium; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Denmark.

XXXX-XXXX/2021/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

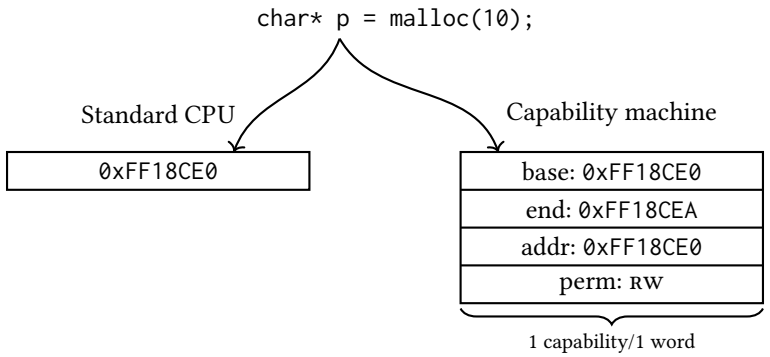


Fig. 1. Representation of a pointer in a standard architecture vs. a capability machine. A capability is similar to a pointer with extra meta-data.

privilege separation at the hardware level and they are a compelling target for secure compilation [Chisnall et al. 2017; El-Korashy et al. 2020; Skorstengaard et al. 2019b; Van Strydonck et al. 2019].

Capability machines distinguish, at the level of hardware, between machine integers and capabilities; and a capability can be understood as a pointer with associated metadata, cf. Fig 1. A machine word containing an integer value can only be used for numerical computations and cannot be used as a pointer to access memory. On the other hand, a machine word containing a capability can be used to access a given portion of memory, depending on the metadata contained in the capability. We also say that the capability *has authority over* some fragment of memory.

A capability thus corresponds to a native machine value, and can be stored in a CPU register or in memory. While this might seem wasteful due to the amount of extra metadata that needs to be carried around, for realistic capability machines a lot of work has been dedicated to the design of compressed representations for capabilities, see, e.g., [Carter et al. 1994; Woodruff et al. 2019]. In this paper, we will abstract from these details and represent capabilities in their uncompressed form, as a tuple carrying the metadata.

A capability machine guarantees the integrity of capabilities: one cannot create fresh capabilities out of thin air or modify the metadata of existing capabilities in arbitrary ways. For instance, CHERI associates tags to machine words to identify whether they represent a capability or an integer. Such a tag bit is checked and set by the machine, and is not directly accessible by software. More generally, new capabilities can only be derived from existing capabilities using a restricted set of operations provided by the machine. As such, all capabilities on the system are recursively derived from the full-authority capabilities that are initially provided to software at boot time. Intuitively, the machine ensures that a given program cannot forge capabilities and obtain more authority than it held previously, a property sometimes referred to as capability monotonicity [Nienhuis et al. 2020].

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

Note, however, that capabilities are low-level, flexible, building blocks, which operate at the level of the machine code and whose metadata "just" triggers some additional runtime checks by the machine. This means that the *properties* we can actually enforce using capabilities crucially depend

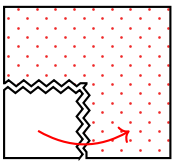
on how we *use* capabilities: the variety of properties that can be enforced stems from how one can use and combine capabilities.

In this paper we show how we can formally *prove* that security properties are enforced for some known verified code, *even when* that code is linked with unverified untrusted third-party code. Our model of interaction between the known and unknown code is very simple: we assume the code is in the same address space and that control is transferred from one to the other using an ordinary jump instruction. We focus on a restricted subset of the capabilities present in the CHERI architecture (using only "normal" read/write capabilities and a kind of so-called sentry capabilities, which provide a basic form of encapsulation, see Section 2.4). Because the security properties we consider hold even in the presence of unverified unknown code, they are sometimes referred to as *robust safety* properties [Swasey et al. 2017]. The security properties we focus on are centered around memory compartmentalization, in particular, local state encapsulation. We consider a range of examples, starting with very basic examples (sharing a buffer with some unknown code), through implementations of closures with encapsulated state, and end up with a quite sophisticated low-level implementation of an interval library, for which we show that certain representation invariants are preserved, even when interacting with unknown code.

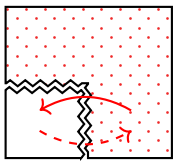
We proceed as follows:

- We first explain informally how one can program with capabilities and use capabilities to enforce memory compartmentalization (Section 2).
- We then introduce the formal operational semantics of a simple capability machine with sentry capabilities (Section 3).
- We define the Cerise program logic which can be used to formally verify the correctness of programs running on the capability machine. Our program logic is defined by instantiating the Iris framework [Jung et al. 2018], which provides an expressive separation logic with powerful reasoning principles, including, in particular, the notion of a *logical invariant* (Section 4).
- We define, using our program logic, the specification of what a "safe" capability and a "safe" program is. Intuitively, a capability (respectively, a program) is "safe" if it cannot be used to invalidate an invariant at the logical level. Hence, safe capabilities can be shared freely with unknown code. Safety of a capability is defined in the program logic as a unary logical relation (Section 5).
- We show that if a program only has access to "safe" values, then running the program itself is also "safe". This is a global property of the capability machine, expressing its capability safety: it is not possible to increase one's authority beyond what was available initially, independently of the sequence of instructions that one executes (Section 5). Concretely, the theorem takes the form of a contract that holds for arbitrary code,[1] and which can be combined in the program logic with manual proofs for trusted code. The last piece of the puzzle is then a so-called Adequacy theorem (Section 4), which relates invariants established in the program logic to the operational semantics of the machine. Given a concrete scenario (typically, a complete system mixing known verified code with unknown untrusted code), this makes it possible to obtain a theorem about the execution of the system which only depends on the operational semantics of the machine (not on the program logic).
- In Section 6 we then return to the examples from Section 2 and show how to use Cerise to formally prove that the desired memory compartmentalization results really do hold.
- In Section 7 we consider more sophisticated examples, which involve dynamic memory allocation. We focus on the low-level implementation of ML-like programs, and introduce

[1]Because it holds for arbitrary code, we sometimes refer to this as a *universal contract.*



(a) Scenario 1: passing control to untrusted code



(b) Scenario 2: being called by untrusted code (possibly many times)

Fig. 2. Two scenarios where a (trusted) component interacts with its (untrusted) context. The trusted component is represented with a plain background, while the untrusted context is represented with a red dotted background.

a heap-based calling convention for closures implementing ML functions. We extend the earlier Adequacy theorem to account for dynamically allocated memory.
- In Section 8 we demonstrate how to use our methodology to establish correctness of object capability patterns (OCPs) from the literature. In particular, we consider the OCP of dynamic sealing, as presented by [Swasey et al. 2017] in the context of a high-level language and we demonstrate that Cerise can be used to prove similar results about a low-level implementation of their example.
- Section 9 offers some perspectives on the relevance of our technical contributions and how we envision them being used in the development of secure systems.
- Finally, we discuss related work in Section 10.

This paper pedagogically introduces and explains the methodology underlying a sequence of recent research papers [Georges et al. 2021; Skorstengaard et al. 2018, 2019a; Van Strydonck et al. 2021], in the form of the Cerise program logic, but also contributes new material. The operational semantics, program logic and logical relation discussed in Sections 3, 4 and 5 are based on those used by [Georges et al. 2021] (but we have removed local and uninitialized capabilities as well as Kripke indexing for simplicity and instead added much more extensive explanations and proofs). Sections 2 and 6 are new; they provide a clear and accessible introduction to capability machine programming and our reasoning tools. The examples in Sections 7-8 are also new and represent a non-trivial verification effort.

The results and examples presented here have been fully formalized in Coq, and are available online: https://github.com/logsem/cerise. The development can also be viewed online at https://logsem.github.io/cerise/journal/; we use circled numbers such as ① to link directly to corresponding Coq formal statements in the following.

2 PROGRAMMING WITH CAPABILITIES

Let us give a taste of how one might use capabilities when writing programs with the goal of enforcing some additional memory protection or encapsulation guarantees. We consider a fairly simple but quite general adversarial model, where we wish to verify the correctness of a *known component* interacting with a possibly adversarial *third-party component* whose code is unverified and untrusted.

In this section we detail two concrete example programs, which use capabilities in two different scenarios. In the first scenario, illustrated in Figure 2a, we consider a program that eventually passes control to the untrusted third-party code, but uses capabilities to protect a region of memory

containing some secret data from being accessed by the untrusted code. In the second scenario (Figure 2b), we consider the case of a verified component being called by the third-party code. The goal is then for the verified component to use capabilities to protect (or "encapsulate") a piece of private memory, which it may access during its execution, but which should remain inaccessible to the unverified code.

2.1 Anatomy of a capability (in our model)

We are interested in a subset of the capabilities available in a CHERI capability machine. We thus work with a simplified machine model, featuring basic capabilities that are used to give access to a range of memory, as well as so-called "sealed entry" capabilities (abbreviated as "sentry" capabilities [Watson et al. 2020, §3.8]) that provide encapsulation features. The sentry capabilities were also called "enter" capabilities in earlier work, e.g., in the M-Machine by [Carter et al. 1994].

Concretely, we model capabilities as 4-tuples (p, b, e, a). In actual hardware, capabilities are encoded as fixed-size binary words, but here we abstract over their concrete representation.

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

A capability (p, b, e, a) represents a machine word that can be used to access memory within the region $[b, e)$ delimited by its base address b and end address e. The permission p specifies what is possible to do within this memory range: permission O specifies that the capability actually gives no access rights, RO grants read-only access to memory, RX grants the right to read and execute the contents of the memory, RW gives read and write access, and RWX gives read, write, and execute access. Capabilities with permission E behave a bit differently (they are used to provide a form of encapsulation), and will be explained later in Section 2.4.

A capability is meant to be used as a pointer, and thus additionally points to a specific address a (typically, but not necessarily, belonging to the range $[b, e)$). Each time the capability is used to access memory, the machine will automatically check that a is between bounds b and e, and that the access is permitted according to p. From a capability (p, b, e, a) it is easy to derive another capability (p, b, e, a') pointing to a different address a' also within range $[b, e)$ – in other words, while a capability points to a specific address, it really holds authority over the whole region delimited by its beginning and end address.

Note that, on a capability machine, machine words can represent not only binary-encoded capabilities, but also traditional fixed-size integers. However, unlike on a traditional computer architecture, integers cannot be used as pointers. In other words, without holding a capability, one cannot access memory at all. In this paper, we rely on difference in notation to distinguish between capabilities and integers. In actual hardware, this is done by associating an extra one-bit tag to each word to distinguish capabilities from integers.

2.2 Sometimes, failure is a good thing

It is worth pointing out a sometimes counter-intuitive aspect of reasoning about security of programs running on a capability machine, especially for readers with a background in reasoning about safety in higher-level languages. For a high-level language, program safety can be seen as the absence of undefined behavior or runtime errors. For instance, an out-of-bounds array access is undefined behavior in C, and it leads to a runtime error, such as raising an exception, in

memory-safe languages such as Rust or OCaml. We are instead interested in *security* properties for which a runtime failure can actually be considered a good thing.

Generally speaking, a low-level machine has many cases where it can fail at runtime, stopping the normal course of execution. In a standard (non-capability) machine, this can happen, e.g., if the machine attempts to execute an invalid instruction which cannot be decoded. The addition of capabilities only adds more possibilities for runtime faults: each time a capability is used, the capability machine will check that it has adequate permission and bounds, and raise a runtime fault otherwise.

Now, the point is that, from a security perspective, these additional runtime faults are a good thing. Using these additional checks, the capability machine turns dangerous behavior (out-of-bounds accesses leading to buffer overflow attacks, etc.) into proper faults before they can cause damage. Thus, for our purposes, it is always safe for the machine to fail: it means that an illegal operation may have been attempted, and the execution has been stopped in response.

Of course, when writing concrete programs, we will typically want to verify that we do not involuntarily trigger faults, as this would make our programs less useful. But when interacting with adversarial code, this is a possibility that we have to take into account anyway: we cannot prevent unknown code from shooting itself in the foot, e.g. by trying to access memory it does not have a valid capability for, or by decoding illegal instructions.

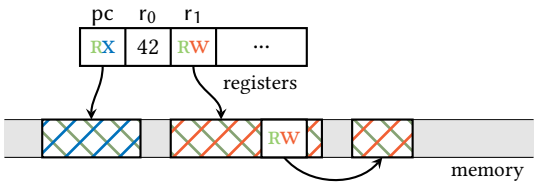
To sum up, in this work we reason about security properties that are not violated in the case of machine failure. This includes, for example, integrity of private data: no data can be compromised if the machine stops running. It is therefore useful to keep in mind that we consider failure to be trivially safe!

2.3 Restricting access to memory by constraining available capabilities

Consider Scenario 1 from Figure 2a: how can one write a program which passes control to untrusted code while protecting some secret data? That is, we wish to write a program that sets up capabilities so that its secrets are preserved even after it runs untrusted code.

The key intuition is that, at any point of the execution, one can only access the part of memory that is accessible using the currently available capabilities. In other words, the authority of a running program comes from the set of capabilities which are transitivitely reachable from the CPU registers.

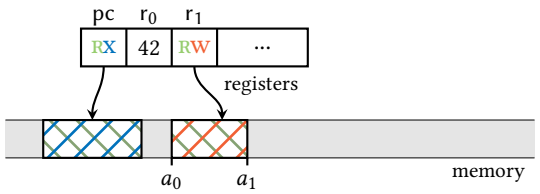
This is illustrated below, in a scenario where the pc register ("program counter") contains a capability with permission RX pointing to some memory region (containing the code of the program being executed), and register r_1 contains a capability with permission RW, pointing to a region of memory, which itself contains a RW capability pointing to another memory region. The collection of the "hatched" memory regions corresponds to the overall subset of memory accessible by the program.



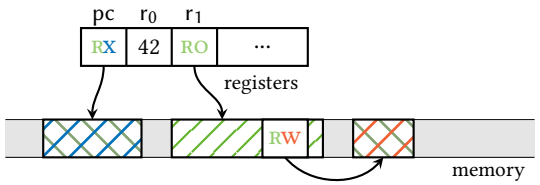
If one wishes to reduce the set of available memory or its associated access rights—for instance to protect secrets from being leaked to an adversary—then it is be enough to constrain the capabilities currently available. This can be done in a few different ways.

First, one can simply remove a capability from registers in order to remove access to the memory it was giving access to. For instance, after executing the instruction "`mov r1 0`", which overwrites the contents of register r_1 with the integer 0, one loses access to the memory regions which were previously accessible from the capability stored in that register.

Alternatively, it is possible to restrict the range of a capability to point to a smaller memory region. This changes the set of accessible memory to a subset of what was previously available. For instance, starting from our initial scenario and running the instruction "`subseg r1 a0 a1`" will change the range of the capability stored in register r_1 to $[a_0, a_1)$. (The machine will check that $[a_0, a_1)$ is indeed included in the range of the original capability.) In our example scenario (illustrated below), we then only keep the beginning of the region accessible from r_1, and this entails that the third region of memory becomes inaccessible, since it was only reachable from a capability stored at the end of the region accessible from r_1.



Finally, one can restrict the permission of a capability to a permission that grants less access rights. For instance, running the instruction "`restrict r1 RO`" in our initial scenario modifies the capability stored in r_1 to only grant read-only access to its corresponding memory region. Note that we still have read-write access to the last memory region, as we can still read the capability (with permission RW) pointing to it.



Example: sharing a sub-buffer with unknown code. Using some of the mechanisms detailed above, we can implement a very simple program that shares a buffer with unknown, possibly adversarial, code while using capabilities to protect some data that would otherwise be vulnerable to buffer overflow attacks.

The assembly code for the program is shown in Figure 3. It consists of a code section containing the instructions of the program, followed by some data which (for simplicity) we simply assume to be statically allocated. The data section holds the zero-terminated string "`Hi`", which we wish to share with the untrusted code, and the integer 42 which represents our secret data.

Initially, we assume the program counter to contain a RWX capability for the whole region holding our program. This capability serves two purposes: it allows the machine to execute our

```
; initially, PC = (RWX, code, end, code)
;            r0 = (unknown) pointer to the continuation
code:
  mov r1 PC                 ; r1 = (RWX, code, end, code)
  lea r1 [data-code]        ; r1 = (RWX, code, end, data)
  subseg r1 [data] [data+3] ; r1 = (RWX, data, data+3, data)
  jmp r0                    ; jump to unknown code: we give it read-write
                            ; access to the first 3 words of the data,
                            ; but not the secret value
data:
  ; the first 3 data words contain public data that will be passed
  ; to the unknown code (the "Hi" string)
  'H', 'i', 0,
  ; they are followed by secret data (the integer 42)
  42
end:
```

Fig. 3. Program sharing a buffer with possibly adversarial code.

program, but can also be manipulated by the program itself to derive a capability pointing to its own data. By convention, the register r_0 is assumed to contain a pointer to the continuation of the program, i.e. other code that the program will pass control to after it is done executing. As no assumption is made about the contents of r_0, it is conservatively assumed to point to unknown, arbitrary code.

Our program executes as follows: it first loads the capability held by the program counter into register r_1. Then, using the `lea` instruction, it changes the "current address" of the capability to point to the `data` label (`lea` modifies a capability by adding an offset to its "current address"). In assembly programs, we use the brackets notation `[...]` to denote an arithmetic expression that is computed statically when assembling the program.

At this point, the capability held in r_1 points to the start of the `"Hi"` string, but has (RWX) authority over the whole code and data section. This capability would be unsafe to share with the untrusted code, as they could simply use `lea` to increment the capability's current address past the end of the string, and obtain a valid capability to the secret value (thus performing a basic "buffer overflow" attack). To prevent this from happening, we use the `subseg` instruction to obtain a capability whose range of authority is restricted to the sub-buffer holding the `"Hi"` string. Finally, we pass control to the untrusted code by using the `jmp` instruction, loading the contents of register r_0 into pc.

This example illustrates that even a basic mode of use of capabilities (restricting them appropriately) can easily prevent buffer overflow attacks. In Section 6.1, we show how we can formally prove that, for any untrusted code, the value of the secret data will be equal to 42 at every step of the execution, including after control has been passed to the untrusted code. We have also developed a relational model, which can be used to prove that the secret value cannot even be read by the unknown code, but the details of this relational model are out of scope of this paper.

2.4 Securely encapsulating code and private capabilities

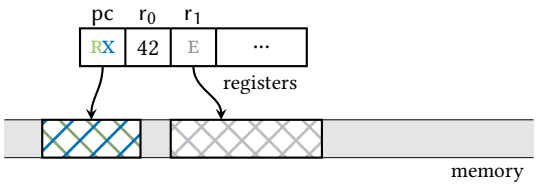
The previous example illustrates how to restrict available capabilities to prevent an adversary from accessing secret data. However, what if we additionally want our program to be called back by the

untrusted code, as in Scenario 2b? In that case, when the trusted code gets invoked again we would like to recover access to the capabilities it previously had to its private state.

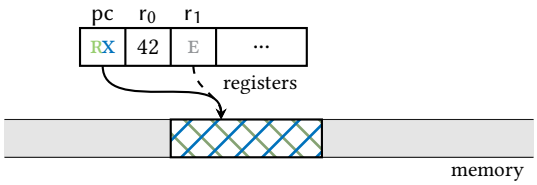
This is unfortunately not achievable with the capabilities that we have described so far. If we remove capabilities to private memory before passing control to untrusted code, then there is no way for us to get them back later on: the only capabilities we will get access to in a further invocation are capabilities the untrusted code itself has access to.

Sentry capabilities provide this missing feature. They implement a form of encapsulation that resembles the use of closures with encapsulated local state in high-level languages, and they allow implementing *compartments* which encapsulate private state and capabilities but can be called from untrusted code. From a security perspective, sentry capabilities allow setting up protection boundaries: the code executing before and after an invocation of a sentry capability has different authority and thus represent distrusting components. We denote sentry capabilities with permission E (for "Enter", a terminology originating from the M-machine [Carter et al. 1994]).

One typically creates a sentry capability pointing to a region of memory describing a compartment containing executable code and local state (or private capabilities to that local state). A sentry capability is opaque: it cannot be used to read or write to the memory region it points to, and it cannot be modified using `restrict` or `subseg`. It can thus be safely shared with untrusted third-parties: they will not be able to access the encapsulated code and data. In the figure below, the memory region pointed to by r_1 (hatched in gray) is not accessible for either reading or writing.



The only possible operation is to "invoke" the sentry capability using the `jmp` instruction, thus passing control to the code held in the region pointed to by the capability (in other words, "running" the compartment). When `jmp` is called on a sentry capability, it turns the capability into a capability with permission read-execute (RX) over the same memory region, and puts it into the program counter register pc. This simultaneously runs the encapsulated code, and gives access to the data and capabilities stored there, which were previously inaccessible. Running instruction `jmp r1` on the scenario of the previous figure leads to the machine state shown below.



Register pc now contains an RX capability to the previously opaque region, meaning that code contained in that region can execute. Furthermore, it may access other capabilities stored in that region, which can in turn be used to transitively access other private regions of memory.

Example: a counter compartment. To illustrate the use of sentry capabilities, let us consider the example of a simple secure compartment implementing a counter. An instance of the counter holds a private memory cell containing the current (integer) value of the counter. Every time the code in the counter's compartment is invoked, it increases the value stored in the memory cell. Using

```
; initially, PC = (RWX, init, end, init)
;            r0 = (unknown) pointer to the context
init:
  mov r1 PC              ; r1 = (RWX, init, end, init)
  lea r1 [data-init]     ; r1 = (RWX, init, end, data)
  mov r2 r1              ; r2 = (RWX, init, end, data)
  lea r2 1               ; r2 = (RWX, init, end, data+1)
  store r1 r2            ; mem[data] <- (RWX, init, end, data+1)
  lea r1 [code-data]     ; r1 = (RWX, init, end, code)
  subseg r1 [code] [end] ; r1 = (RWX, code, end, code)
  restrict r1 E          ; r1 = (E, code, end, code)
  mov r2 0               ; r2 = 0
  jmp r0                 ; jump to unknown code: we only give it access
                         ; to an enter capability pointing to 'code'
; when 'code' gets executed from the E capability,
;   PC = (RX, code, end, code)
;   r0 = (unknown) return pointer to the continuation
code:
  mov r1 PC              ; r1 = (RX, code, end, code)
  lea r1 [data-code]     ; r1 = (RX, code, end, data)
  load r1 r1             ; r1 = (RWX, init, end, data+1)
  load r2 r1             ; r2 = <counter value>
  add r2 r2 1            ; r2 = <counter value> + 1
  store r1 r2            ; mem[data+1] <- <counter value> + 1
  mov r1 0               ; r1 = 0
  jmp r0                 ; return to unknown code
data:
  0xFFFF, ; will be overwritten with (RWX, init, end, data+1), i.e.
          ; a read-write capability to the counter value
  0       ; our private data: the current value of the counter
end:
```

Fig. 4. Program implementing a secure counter

a sentry capability, one can expose the counter to an untrusted context, without giving it direct access to the counter value.

It is worth pointing out that this is similar to the use of closures encapsulating local state in high-level languages. Typically, a similar counter program could be implemented in a high-level language as follows, using a function closure to encapsulate a reference holding the counter value.

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda(). \; x := !x + 1; !x)$$

As before, our actual counter program is implemented in assembly, and its code appears in Figure 4. Its implementation is divided into two parts. First, the code starting at label `init` (and ending at `code`) is used to set up the counter compartment; it is intended to run only once at the beginning of the program. Then, the region between `code` and `end` corresponds to the contents of the counter compartment itself, including its executable code (between `code` and `data`) and private data (between `data` and `end`).

The role of the initialization code is to create a sentry capability encapsulating the code–end region, and then pass control to the (untrusted) context, giving it access to the newly created sentry capability. Additionally, the initialization code stores at address data a capability giving read-write access to the compartment's region, and pointing to the counter's value at address data+1.

One might wonder why we have this extra indirection to the counter's value through the capability in data. Recall that after calling jmp on a sentry capability, the program counter is only provisioned with an RX capability. For the counter code to be able to actually increment the counter value (at address data+1), it needs to have write access to it. The additional RWX capability stored at address data by the initialization code is thus used to "promote" read access on the compartment's region into read-write access to that same region.

The code of the counter's compartment can then run many times, once each time the context chooses to invoke the sentry capability it got from the initialization code. At each invocation, the counter's implementation (at address code) reads the RWX capability stored in the data section, uses it to increment the value of the counter, and passes control back to its caller.

Let us walk through the details of the code. The initialization code is assumed to run starting with a program counter giving RWX access over the whole program region. The first four instructions derive, from the program counter, RWX capabilities pointing to addresses data and data+1. Then, using the store instruction, the capability (RWX, init, end, data+1) is stored at address data. Next, after using lea and subseg to adjust the address and bounds of the capability, a sentry capability is created pointing to the compartment's region [code, end). This is done using the restrict instruction, turning a capability with permission RWX into a capability with permission E. Register r_2 is then cleared, to make sure that the RWX capability pointing to the counter value is not leaked to the context. Finally, the initialization code jumps to the pointer in r_0, which by convention points to the context.

The compartment's code (starting at address code) then gets executed each time the context invokes the sentry capability. Because we have only shared a sentry capability (E, code, end, code) with the context, we know that when the compartment gets executed, the program counter must contain (RX, code, end, code). By reading the program counter, the first two instructions of the code then derive an RX capability pointing to address data, and use it (with load) to read the capability that was stored there, granting RWX access to data+1. The subsequent load, add and store instructions use this second capability to increment the value of the counter. Finally, before returning to the context by jumping to r_0, the program takes care of clearing register r_1, overwriting its contents with 0. This is quite crucial, as otherwise an RWX capability would be leaked to the context, giving it direct access to the counter's private state!

To sum up, our example program carefully selects which capabilities it shares with unknown code, and leverages the encapsulation properties of sentry capabilities provided by the machine. Consequently, it should seem clear, at least informally, that the integrity of the counter's value is guaranteed through the execution. More precisely, we should be able to formally prove some invariant about it: for instance, that it is nonnegative at every step of the execution, for any untrusted context. In Section 6.2, we show in more detail how to formally establish this property.

In this section, we have showcased how one might program with capabilities in order to obtain security guarantees, and make it possible to interact with adversarial code while protecting private data and invariants.

In the rest of this paper, we show how we can make the intuitions that we have developed so far more precise, and formally prove capability safety for machine code programs that interact with untrusted code. Namely:

- We expect to have some concrete known code, which has some private data and invariants, and interacts with untrusted code.
- We formally define the operational semantics of the capability machine that we consider (Section 3). This precisely defines the behavior of the machine on which the rest of our framework is built.
- Then we develop (Section 4) a program logic which supports formally verifying correctness properties about known code. Given some verified known code, we would then like to be able to conclude some result about a complete execution of the machine, when it runs a combination of the known code and some arbitrary untrusted code.
- To that end we need a way of formally capturing the fact that the machine effectively restricts the behavior of arbitrary code at runtime, by limiting the capabilities it has access to. We do this (Section 5) by defining a logical relation capturing "capability safety" of arbitrary code.
- By combining the Adequacy theorem of our program logic and the Fundamental theorem of our logical relation, we can prove safety of concrete examples (Section 6) and obtain theorems about complete executions of the machine.

3 OPERATIONAL SEMANTICS OF A CAPABILITY MACHINE

The very basis of our framework is a formal description of the capability machine we consider: which instructions it supports, and its behavior when it runs and executes programs. Technically speaking, this description corresponds to the operational semantics of the machine, upon which the program logic described next in Section 4 is built.

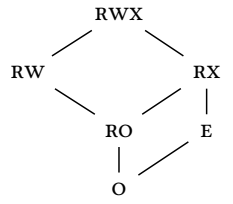
Our capability machine draws inspiration from CHERI [Watson et al. 2020], albeit in a simplified form, and only covers a subset of the features found in CHERI machines. Compared to a realistic CHERI machine, we consider a number of simplifications: our instruction set is minimal, our machine does not have virtual memory or different privilege levels, machine words can store unbounded integers, every instruction can be encoded in a single machine word, we do not consider memory alignment issues, and we abstract away from the binary encoding of capabilities. Nevertheless, our semantics does capture many of the aspects that make reasoning about machine code programs challenging: our machine has a finite amount of memory, a fixed number of registers, and there are no distinctions between code and data nor structured control flow for programs, owing to the fact that program instructions are simply encoded and stored in memory as normal integers.

Figure 5 gives the basic definitions that will play a role in the operational semantics of machine instructions. The set of memory addresses Addr is finite, and corresponds to the integer range $[0, \mathrm{AddrMax}]$. A memory word $w \in \mathrm{Word}$ is either an (unbounded) integer or a capability c. Capabilities are of the form (p, b, e, a), giving access to the memory range $[b, e)$ with permission p, while currently pointing to a. The permissions p are ordered according to the lattice appearing at the top-right of the figure, inducing a bottom-to-top partial order $\preccurlyeq$ on permissions. There are six different permissions; the null (O), read-only (RO), enter (E), read-write (RW), read-execute (RX) and read-write-execute (RWX) permissions.

The state of the machine is modeled by the semantics as a pair of an execution state s and a configuration φ. An execution state flag indicates whether the machine is presently running (Running), has successfully halted (Halted), or has stopped execution by raising an error (Failed). A configuration φ contains the state of the registers $\varphi.reg$ and the memory $\varphi.mem$. A register file reg consists of a map from register names r to machine words, while the memory m maps addresses to words.

Next, Figure 5 shows the list of instructions of our machine. An instruction i typically operates on register names r, but can also sometimes take integer values as parameters; ρ denotes an instruction parameter which can be either a register name or a constant integer. Our machine

$$
\begin{array}{lcll}
a & \in & \text{Addr} & \triangleq [0, \text{AddrMax}] \\
p & \in & \text{Perm} & ::= \textsc{o} \mid \textsc{e} \mid \textsc{ro} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{rwx} \\
c & \in & \text{Cap} & \triangleq \{(p, b, e, a) \mid b, e, a \in \text{Addr}\} \\
w & \in & \text{Word} & \triangleq \mathbb{Z} + \text{Cap} \\
reg & \in & \text{Reg} & \triangleq \text{RegName} \to \text{Word} \\
m & \in & \text{Mem} & \triangleq \text{Addr} \to \text{Word} \\
s & \in & \text{ExecState} & ::= \text{Running} \mid \text{Halted} \mid \text{Failed} \\
\varphi & \in & \text{ExecConf} & \triangleq \text{Reg} \times \text{Mem}
\end{array}
$$



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

$$
\begin{array}{l}
r \in \text{RegName} ::= \mathtt{pc} \mid \mathtt{r}_0 \mid \mathtt{r}_1 \mid \ldots \mid \mathtt{r}_{31} \qquad\qquad \rho \in \mathbb{Z} + \text{RegName} \\
i \;::=\; \mathtt{jmp}\ r \mid \mathtt{jnz}\ r\ r \mid \mathtt{mov}\ r\ \rho \mid \mathtt{load}\ r\ r \mid \mathtt{store}\ r\ \rho \mid \mathtt{add}\ r\ \rho\ \rho \mid \mathtt{sub}\ r\ \rho\ \rho \mid \\
\qquad \mathtt{lt}\ r\ \rho\ \rho \mid \mathtt{lea}\ r\ \rho \mid \mathtt{restrict}\ r\ \rho \mid \mathtt{subseg}\ r\ \rho\ \rho \mid \mathtt{isptr}\ r\ r \mid \mathtt{getp}\ r\ r \mid \\
\qquad \mathtt{getb}\ r\ r \mid \mathtt{gete}\ r\ r \mid \mathtt{geta}\ r\ r \mid \mathtt{fail} \mid \mathtt{halt}
\end{array}
$$

Fig. 5. Base definitions for the machine's words, state, and instructions.

features general purpose registers ($\mathtt{r}_0$ – $\mathtt{r}_{31}$), on top of the pc register, which points to the address in memory where the currently executing instruction is stored. (Technically speaking, pc must point to a memory cell containing an integer which can be successfully decoded into an instruction.) pc should therefore always contain a capability with at least permission RX; in any other case, the machine fails immediately.

Figure 6 defines the small-step operational semantics for the capability machine. The rule ExecSingle describes how a single instruction gets executed. An execution step requires an executable and in-bounds capability in the pc register, and fails otherwise. It expects the memory cell pointed to by the capability to store an integer z, decodes it into an instruction and executes the instruction on the current state φ; the new configuration is denoted $[\![decode(z)]\!](\varphi)$. The table making up most of Figure 6 defines the operational behavior $[\![i]\!](\varphi)$ for each instruction i of the machine.

Most instructions use the auxiliary function updPC to increment the pc register after their proper operations. Because the address space is finite, pointer arithmetic such as incrementing pc can result in illegal addresses. To avoid notational clutter, we will always write as if arithmetic operations succeed, and consider that otherwise the machine transitions to a Failed state. The auxiliary function getWord is used to get the value corresponding to the argument ρ of an instruction: either its corresponding integer value if it is an immediate integer, or the contents of the corresponding register if it is a register name. The auxiliary function updatePcPerm is used in the definition of the behavior of the `jmp` and `jnz` instructions to unseal sentry capabilities. As mentioned previously, an additional effect of these jump instructions is to unseal sentry (E) capabilities into RX capabilities.

We now describe the semantics of the instructions of the machine, as formally defined in the table of Figure 6. The `fail` and `halt` instructions stop the execution of the machine, in the Failed and Halted state respectively. `mov` r ρ copies ρ (either an immediate value or the contents of the corresponding register name) into register r. The instructions `load` and `store` allow reading and writing memory: `load` r_1 r_2 reads the value pointed to by the capability in r_2 provided it has the permission R and points within its bounds; `store` r ρ stores ρ to the location pointed to by the capability in r provided it has the W permission and points within bounds. The `jmp` and `jnz` instructions correspond to an unconditional and conditional jump respectively, thus loading

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

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

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

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

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

Fig. 6. Operational semantics: execution of a single instruction.

the provided capability into pc. Using updatePcPerm, in the case of a sentry (E) capability, they unseal it into a RX capability first. Three instructions allow deriving new capabilities from existing ones. `restrict` r ρ allows restricting the permission of a capability (where ρ provides an integer encoding of the desired permission), provided it is less permissive than the current permission according to $\leqslant$. `subseg` r ρ_1 ρ_2 restricts the range of authority of the capability stored in r, provided it is a subset of the current range of the capability. `lea` r ρ modifies the current address of the capability in r, by adding to it the integer offset ρ. As should be expected, `subseg` and `lea` fail for sentry capabilities. Arithmetic operations are provided by the `add`, `sub` and `lt` instructions, which implement addition, subtraction, and comparison on integers, respectively. Finally, a number of instructions allow inspecting machine words and capabilities. `isptr` can be used to query whether a machine word is an integer or a capability, and `getp`, `getb`, `gete`, and `geta` return the different parts of a capability (permission, bounds and address). (More precisely, `getp` returns an integer encoding the permission, as given by encodePerm.) If any of the capability checks for an instruction are not satisfied, the machine fails.

An important aspect of our operational semantics is how it explicitly accounts for errors: when a capability check fails (for instance when a program tries to use a capability outside of its range), the semantics does not get stuck (meaning that it would not be able to reduce): instead, it explicitly transitions to a state with the Failed execution state flag.

4 PROGRAM LOGIC

The operational semantics presented in the previous section formally define the behavior of our machine when it runs and executes code. Based on that, we expect to be able to formally verify concrete programs running on the machine.

The most direct approach would be to manually establish properties of sequences of reduction steps, based on the sole definition of the operational semantics. We do not follow this approach, because it would quickly become very tedious even for simple programs.

Instead, we draw from previous research in program logics and separation logic, and define Cerise: a program logic which provides a convenient framework in which to modularly reason about programs running on our machine. Indeed:

- It is typically more convenient to devise a system of proof rules for programs, rather than work directly at the level of abstraction provided by the bare operational semantics. Such rules form a program logic, which can be proved sound according to the operational semantics, and then can be used to verify properties of concrete programs.
- Separation logic, a family of program logics, has been widely used to reason about programs manipulating shared mutable state (such as memory). On our capability machine, not only do all programs access a mutable shared memory, but programs are themselves represented as unstructured data in memory; so the use of separation logic is particularly called for. Separation logic enables modular reasoning about programs that operate only on a sub-part of the global state, allowing them to be freely composed with programs that operate on a disjoint part of the state.

The first step is to consider what part of the machine state should be described by separation logic assertions. Here, the machine state consists of both the machine memory and the machine registers. Indeed, it is useful to modularly reason about programs operating on both a subset of memory and a subset of the available registers.

Technically speaking, we build the Cerise program logic on top of the Iris framework [Jung et al. 2018], which provides us with additional useful features, such as invariants. In the following we introduce both the basic separation logic assertions describing the machine state and additional

$$
\begin{array}{rl}
P, Q \in \mathit{iProp} ::= & \\
\quad \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \ldots & \text{higher-order logic} \\
\quad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ulcorner \phi \urcorner \mid \square P \mid \triangleright P & \text{separation logic} \\
\quad \mid \mathsf{a} \mapsto w \mid r \Mapsto w \mid \vec{a} \mapsto \vec{l} & \text{machine resources} \\
\quad \mid \boxed{P} & \text{invariants} \\
\quad \mid \langle P \rangle \rightarrow \langle s.\, Q \rangle \mid \{P\} \rightsquigarrow \{s.\, Q\} \mid \{P\} \rightsquigarrow \bullet & \text{program logic}
\end{array}
$$

Fig. 7. The syntax of our program logic.

features inherited from Iris (Section 4.1). Then, we describe the rules that are used to specify the execution of machine instructions and programs (Section 4.2).

Note that the program logic is, in a sense, only a technical device. The end goal is to obtain theorems that only refer to reductions in the operational semantics of our machine. To that end, we present (Section 4.3) an Adequacy theorem for our logic, which allows us to "extract" a correctness theorem expressed in terms of the operational semantics of the machine from a proof established in the program logic.

4.1 Basic resources

Figure 7 shows the syntax of our Cerise program logic based on Iris. We write *iProp* for the universe of propositions. These feature the standard connectives of higher-order logic and separation logic, including the separating conjunction $*$ and the magic wand $\mathrel{-\!\!*}$ (read as an implication). The proposition $\ulcorner \phi \urcorner$ asserts that the pure proposition ϕ holds, where ϕ is a proposition from the meta logic.

Iris assertions can be divided in two categories: *ephemeral* assertions and *persistent* assertions. Ephemeral assertions describe facts or resources that are available at a given point but might become false or unavailable later. Persistent assertions describe facts that never cease to be true. The assertion $\square P$, read "persistently P", is persistent, and asserts ownership over resources whose duplicable part satisfies P. In other words, $\square P$ is like P except that it does not assert any exclusive ownership over resources. As the knowledge associated with a persistent assertion can never be invalidated, persistent assertions can be freely duplicated.

The modality $\triangleright P$ expresses (roughly) that the assertion P holds after one "logical step" of execution. In this paper, we mainly use it to define recursive predicates using guarded recursion. It is not necessary to understand how the modality behaves in detail and the reader can safely ignore it for the most part and just recall that it supports an abstract accounting of execution steps.

Our logic includes resources (predicates) that describe parts of the current state of the machine. The assertion $\mathsf{a} \mapsto w$ expresses that the memory cell at address a contains the machine word w. Furthermore, this assertion should be read as giving *unique ownership* over location a, giving the right to freely read and update the corresponding memory cell. Similarly, the assertion $r \Mapsto w$ asserts ownership of a CPU register r containing the word w. We write $\vec{a} \mapsto \vec{l}$ for the ownership of contiguous memory cells at addresses $\vec{a}$ containing $\vec{l}$.

A key feature of the logic is the notion of an invariant. The assertion $\boxed{P}$ asserts that P should hold at all times, now and for every future step of the execution (where P can be any separation logic assertion). An invariant is a persistent assertion. An invariant $\boxed{P}$ can be created (or "allocated") by handing over the resources for P, turning them into $\boxed{P}$. Then, whenever we know that $\boxed{P}$ holds, we can get access to the resources P held in the invariant, but only for the duration of one program step. Indeed, since the invariant must hold at every step of the execution, when accessing

its resources, one needs to show that it holds again no later than one program step after. A more precise rule for accessing invariants is given next in Section 4.2 (rule Inv).

4.2 Program specifications

The predicates for machine resources we just presented allow describing the state of the machine. Our logic, moreover, includes assertions that can be used to specify machine executions, similar to *Hoare triples* used in program logics for high-level languages. Because we work with a low-level machine (where code is located in memory), we distinguish between three different types of program specifications:

$$\begin{array}{ll} \langle P \rangle \rightarrow \langle s.\, Q \rangle & \text{single instruction} \\ \{P\} \rightsquigarrow \{s.\, Q\} & \text{code fragment} \\ \{P\} \rightsquigarrow \bullet & \text{complete safe execution.} \end{array}$$

In each case, P are Q are separation logic assertions describing the state of the machine (registers and memory). P corresponds to a pre-condition, Q a post-condition, and s binds in Q the corresponding execution state (of type ExecState, see Figure 5).

Informally, $\langle P \rangle \rightarrow \langle s.\, Q \rangle$ holds if, starting from a machine state satisfying P, the machine can execute one step of computation, and reach a state satisfying Q in an execution state s. The predicate $\{P\} \rightsquigarrow \{s.\, Q\}$ holds if, starting from a state satisfying P, then the machine can diverge (i.e. loop) or reach a state satisfying Q in an execution state s. This is typically used to describe the execution of a code fragment. Finally, $\{P\} \rightsquigarrow \bullet$ holds if, starting from a machine state satisfying P, then the machine loops forever or runs until completion, ending in either a Halted or Failed state. In this case, we say that the initial configuration described by P is *safe*. (Not every configuration is safe: the resources in P describing registers and memory must suffice for the machine to run and execute the code pointed to by pc: we do not have $\{pc \Mapsto w\} \rightsquigarrow \bullet$ in general.)

Additionally, these three specifications *require the logical invariants to be preserved at every step of the execution.* This requirement is implicit in the definition of invariants, but it is a crucial reasoning principle that we will leverage.

Echoing back to Section 2.2, note that our program specification for a complete safe execution allows the program to fail (or diverge). Indeed, we will capture the preservation of security properties by preserving *invariants* throughout execution and having the machine fail is both fine (invariants are trivially preserved when the machine ends up in a failure state) and unavoidable (we cannot prevent unknown code from triggering a capability check failure). Similar considerations apply for divergence.

Notations. In the rest of the paper, we will rely on a couple of additional notations when writing program specifications. Because we often want to reason about the case where an instruction (or program fragment) does not fail, we write $\langle P \rangle \rightarrow \langle Q \rangle$ (respectively $\{P\} \rightsquigarrow \{Q\}$) to denote a resulting execution state equal to Running:

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

When writing pre- and post-conditions, we will often need to include a points-to resource describing the contents of the pc register. We introduce a short-hand notation for that purpose, and write $w; P$ to assert P and additionally that pc is set to w:

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

Using these two notations, the specification for a single instruction, in a case where it does not fail, is written as $\langle w_0; P\rangle \rightarrow \langle w_1; Q\rangle$ (typically, we have $w_1 = w_0 + 1$, except in the case of the `jmp` and `jnz` instructions, or when explicitly writing to the pc register).

Properties. Our program specifications satisfy the well-known "frame rule" of separation logic, which permits local reasoning, and asserts that it is always possible to extend a specification by adding arbitrary resources not accessed by the program.

$$\textsc{FragFrame}\ \frac{\{P\} \rightsquigarrow \{s.\, Q\}}{\{P * R\} \rightsquigarrow \{s.\, Q * R\}} \qquad \textsc{StepFrame}\ \frac{\langle P\rangle \rightarrow \langle s.\, Q\rangle}{\langle P * R\rangle \rightarrow \langle s.\, Q * R\rangle} \qquad \textsc{FullFrame}\ \frac{\{P\} \rightsquigarrow \bullet}{\{P * R\} \rightsquigarrow \bullet}$$

Program specifications can also be composed using sequencing rules. In order to establish a specification of the form $\{P\} \rightsquigarrow \{s.\, Q\}$, one typically uses single-instructions rules ($\langle R\rangle \rightarrow \langle s.\, S\rangle$) in a sequence, one for each instruction of the relevant code block. Specifications for two program fragments that follow each other can also be combined to obtain a specification for the sequence of the two fragments. We prove general sequencing rules for our three kind of specifications; for simplicity, we only reproduce here restricted rules that deal with successful executions (relying on the notations introduced before):

$$\textsc{SeqFrag}\ \frac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}} \qquad \textsc{SeqFull}\ \frac{\{P\} \rightsquigarrow \{Q\} \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet}$$

$$\textsc{StepFull}\ \frac{\langle P\rangle \rightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \bullet}{\{P\} \rightsquigarrow \bullet} \qquad \textsc{StepFrag}\ \frac{\langle P\rangle \rightarrow \langle Q\rangle \qquad \{Q\} \rightsquigarrow \{R\}}{\{P\} \rightsquigarrow \{R\}}$$

When reasoning about a single execution step, one can additionally access resources held in known invariants. This is done using the Inv rule, given below: [2]

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

Example specifications. As illustrative examples, Figure 8 shows specifications for the `subseg`, `load` and `store` instructions, as well as the `rclear` macro which is used to clear the contents of a number of specified registers. The first rule shows a specification for the `subseg` instruction. It states that if the program counter contains a capability pointing to a memory location a_{pc}, if that location contains an integer n which decodes into `subseg` r z_1 z_2, and if the register r contains a capability, then assuming that the program counter is valid (ValidPC(...)) and that z_1 and z_2 are valid new bounds (ValidSubseg(...)), the machine successfully increments the program counter and restricts the capability held in register r with new bounds z_1 and z_2.

The second rule is also a specification for `subseg`, but in a case where it fails a bound check, i.e. ValidSubseg(p, b, e, z_1, z_2) does not hold. (For instance, when the new bounds z_1 and z_2 would allow accessing more memory than what is available through the original capability.) Then, the rule does give us a specification for an execution step, but with a resulting execution state of `Failed`, meaning that the execution cannot continue afterwards.

[2]For clarity of the presentation, we choose to omit additional details related to Iris invariant namespaces and masks. We refer to the Coq development for the full details (2).

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \mathrm{ValidSubseg}(p, b, e, z_1, z_2) \qquad \mathtt{decode}(n) = \mathtt{subseg}\ r\ z_1\ z_2}{\begin{array}{l}\left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\quad a_{pc} \mapsto n * r \Mapsto (p, b, e, a)\right\rangle \rightarrow \\ \left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc}+1)\ ;\ a_{pc} \mapsto n * r \Mapsto (p, z_1, z_2, a)\right\rangle\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \neg\mathrm{ValidSubseg}(p, b, e, z_1, z_2) \qquad \mathtt{decode}(n) = \mathtt{subseg}\ r\ z_1\ z_2}{\begin{array}{r}\left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ a_{pc} \mapsto n * r \Mapsto (p, b, e, a)\right\rangle \rightarrow \\ \left\langle s.\ \ulcorner s = \mathsf{Failed}\urcorner * \left((p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\ a_{pc} \mapsto n * r \Mapsto (p, b, e, a)\right)\right\rangle\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \mathrm{ValidLoad}(p, b, e, a) \qquad \mathtt{decode}(n) = \mathtt{load}\ dst\ src}{\begin{array}{l}\left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\quad a_{pc} \mapsto n * dst \Mapsto - * src \Mapsto (p, b, e, a) * a \mapsto w\right\rangle \rightarrow \\ \left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc}+1)\ ;\ a_{pc} \mapsto n * dst \Mapsto w * src \Mapsto (p, b, e, a) * a \mapsto w\right\rangle\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) \qquad \mathrm{ValidStore}(p, b, e, a) \qquad \mathtt{decode}(n) = \mathtt{store}\ dst\ src}{\begin{array}{l}\left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc})\ ;\quad a_{pc} \mapsto n * dst \Mapsto (p, b, e, a) * src \Mapsto w * a \mapsto -\right\rangle \rightarrow \\ \left\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc}+1)\ ;\ a_{pc} \mapsto n * dst \Mapsto (p, b, e, a) * src \Mapsto w * a \mapsto w\right\rangle\end{array}}$$

$$\frac{\forall i \in [0, n),\ \mathrm{ValidPC}(p, b, e, a_i) \qquad n = \mathrm{length}(\mathrm{rclear\_instrs}\ l)}{\begin{array}{l}\left\{(p, b, e, a_0); \mathbin{\text{✻}}_{r \in l} r \Mapsto - * \mathbin{\text{✻}}_{i \in [0,n)} a_i \mapsto (\mathrm{rclear\_instrs}\ l)[i]\right\} \rightsquigarrow \\ \left\{(p, b, e, a_n); \mathbin{\text{✻}}_{r \in l} r \Mapsto 0\ * \mathbin{\text{✻}}_{i \in [0,n)} a_i \mapsto (\mathrm{rclear\_instrs}\ l)[i]\right\}\end{array}}$$

$$\begin{array}{lcl}\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}) & \triangleq & \textsc{rx} \preccurlyeq p_{pc} \wedge b_{pc} \le a_{pc} < e_{pc} \\ \mathrm{ValidSubseg}(p, b, e, z_1, z_2) & \triangleq & p \neq \textsc{e} \wedge b \le z_1 \wedge 0 \le z_2 \le e \\ \mathrm{ValidLoad}(p, b, e, a) & \triangleq & \textsc{ro} \preccurlyeq p \wedge b \le a < e \\ \mathrm{ValidStore}(p, b, e, a) & \triangleq & \textsc{rw} \preccurlyeq p \wedge b \le a < e \\ \mathrm{rclear\_instrs}\ l & \triangleq & \mathrm{map}\ (\lambda r.\ \mathtt{encode}(\mathtt{move}\ r\ 0))\ l\end{array}$$

Fig. 8. Specifications for the machine instructions subseg, load and store and for the rclear macro that sets a given list of registers to zero. Changes to the machine state are highlighted in red.

The third and fourth rules give specifications for the load and store instructions (in non-failing cases). The specification for load states that load *dst src* loads a word from memory pointed to by a capability in register *src* and stores its contents in register *dst*. The specification for store states that store *dst src* reads a word from the *src* register and writes it into the memory location pointed to by the capability in *dst*.

Note that these specifications for subseg, load and store are not in fact the most general specifications for these instructions. They assume that some side-conditions hold, and specify the behavior of the instruction in the case of either a "normal" successful execution, or a failing one. These specifications are typically useful for reasoning about the correctness of a concrete program. We have also proved in Coq (e.g., ③ for the subseg instruction) "most general" specifications, covering in one lemma all possible cases for a given instructions. These are useful for deriving the more specific rules shown previously. Furthermore, we use them directly in the proof of the Fundamental Theorem (Theorem 2), for specifying the behavior of arbitrary instructions that might or might not fail.

The last rule of Figure 8 shows a derivable specification for a program composed of several instructions, the rclear macro. This macro (meaning, a small program that is typically inserted inline as part of a larger program) clears a number of registers by setting their content to 0. It is

parameterized by a list l of register names, and its code consists of a sequence of instructions `move` r 0 for each register name r in l. We state `rclear`'s specification using the program specification for code fragments. This specification is provable using the basic reasoning rules for `move`. It requires that the body of the macro ("rclear_instrs l") is laid out contiguously in memory range $[a_0, a_n)$, while the program counter initially points to a_0. When the program counter eventually points to a_n, the address immediately after the macro's instructions, then all the registers in l have been cleared and now contain 0. (The "big star" $\ast$ denotes an iterated separating conjunction, here over the registers r in list l.)

4.3 Adequacy theorem

After establishing program specifications and properties at the level of our program logic, we ultimately want to transfer these results into properties of a program execution at the level of the operational semantics of the bare machine. Generally speaking, we prove using the rules of the Iris logic a statement of the form $\boxed{P} \vdash Q$, where P and Q are Iris propositions (read "Q holds assuming invariant P"). From this, we want to deduce that some mathematical proposition Φ holds (as a Coq proposition, in our case), where Φ describes some property of the machine execution expressed in terms of its operational semantics.

Because we are interested in establishing *invariants* about a program execution, we typically want to obtain in Φ that at every step of the execution, the state of the machine satisfies an invariant corresponding to the Iris assertion P.

Deriving mathematical facts from Iris proof derivations is made possible thanks to the so-called *adequacy* theorem of Iris ④. This theorem has a very general but intricate statement. In this section, we describe a simpler but more specialized adequacy theorem for our capability machine, which we can use to reason about the examples introduced in Section 2. (We also describe in Section 7 a more advanced adequacy theorem, suitable for reasoning about programs such as the case study of Section 8.) This specialized adequacy theorem is itself established on top of the general Iris adequacy theorem. When it applies, it is more convenient to use; but in the general case, it is always possible to directly leverage the general adequacy theorem.

We now present our specialized adequacy theorem. We first define a notion of *memory invariant* (Definition 1), which corresponds to a predicate over a finite subset of the machine memory. Typically, we will consider predicates of the form: "the value at this specific memory address holds a positive integer" (for instance, the value of the counter of Section 2.4). A memory invariant is given by a predicate I over machine memory and a set of addresses D (the "domain" of the invariant); we then require that I is not impacted by changes outside of D.

Definition 1 (Memory invariant ⑤). *We say that I is a memory invariant over D if I is a predicate over machine memory, D a finite set of addresses, and:*

$$\forall m\, m'.\ (\forall a \in D.\ m(a) = m'(a)) \Longrightarrow I(m) \Leftrightarrow I(m').$$

We now present the statement of our specialized adequacy theorem; we explain the ingredients in the theorem statement below. Given a memory invariant I over a set D, our adequacy theorem (Theorem 1) can be used to show that I indeed holds of the memory at every step of the execution, provided we can show that it holds as an invariant in Iris.

Theorem 1 (Adequacy ⑥). *Given a memory invariant I over D, a memory fragment $prog$: $[b, e) \to$ Word, a memory fragment adv : $[b_{adv}, e_{adv}) \to$ Word, an initial memory mem, and an initial register file reg, assuming that:*

(1) *the initial state of memory mem satisfies:*

$$prog \uplus adv \subseteq mem \qquad D \subseteq \mathrm{dom}(prog) = [b, e)$$

(2) *invariant I holds of the initial memory:*

$$I(mem)$$

(3) *the adversary region contains no capabilities:*

$$\forall a \in \mathrm{dom}(adv).\ adv(a) \in \mathbb{Z}$$

(4) *the initial state of registers reg satisfies:*

$$reg(\mathrm{pc}) = (\textsc{rwx}, b, e, b), \quad reg(\mathrm{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}), \quad reg(r) \in \mathbb{Z} \text{ otherwise}$$

(5) *the proof in Iris that the initial configuration is safe given invariant I:*

$$\forall reg,\ \boxed{\exists m, \circledast_{(a,w)\in m}\, a \mapsto w * \lceil \mathrm{dom}(m) = D \rceil * \lceil I(m) \rceil} \vdash \left\{ (\textsc{rwx}, b, e, b);\ \begin{array}{l} \mathrm{r}_0 \Rightarrow (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \circledast_{(r,v)\in reg,\ r\notin\{\mathrm{pc},\mathrm{r}_0\}}\, r \Rightarrow z * \lceil z \in \mathbb{Z} \rceil * \\ \circledast_{(a,z)\in adv}\, a \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \circledast_{(a,w)\in prog,\ a\notin D}\, a \mapsto w \end{array} \right\} \rightsquigarrow \bullet$$

Then, for any reg′, mem′, if $(reg, mem) \longrightarrow^* (reg', mem')$*, then* $I(mem')$*.*

Theorem 1 establishes that, starting from an initial machine state (reg, mem), any subsequent machine state (reg', mem') satisfies $I(mem')$. This is subject to a number of conditions, in particular about the initial state of the machine.

First, the initial memory must be provisioned with relevant code and data. This means that the program that we wish to verify (both its code and data) given by memory fragment $prog : [b, e) \to \text{Word}$ should be included in the initial memory. Moreover, some additional "adversarial code" given by $adv : [b_{adv}, e_{adv}) \to \text{Word}$ should be included in the initial memory. Indeed, we are not only interested in reasoning about the execution of our verified program in isolation, but also its interaction with unverified, possibly adversarial code. The initial memory *mem* should therefore include *prog* and *adv*, in disjoint regions. Furthermore, the domain of the invariant I should be included in the program's region $[b, e)$. The intent is that I specifies an invariant about some private data of the verified program, and thus should not depend on other parts of memory.

Second, as should be expected, the invariant I must hold of the initial memory *mem*.

Third, the adversary memory *adv* is required not to contain any capabilities. This conservatively ensures that *adv* does not contain any "rogue" capability that would give undesired access to the verified program's private state. No further assumption is made about *adv*, which is thus free to contain arbitrary code (i.e. instructions encoded as integers). Furthermore, note that the absence of capabilities in *adv* does not mean that code in *adv* will not be able to access memory at all: at runtime, it will still get access to a capability to its own region through the program counter pc.

Then, the initial register file *reg* should be provided with a RWX capability to the verified program in pc (meaning that it executes first), and a capability to the unverified code in register r_0 (as we have seen in Section 2, by convention r_0 holds the pointer to a program's continuation). Other registers are conservatively required not to contain any capabilities.

Finally, one needs to establish at the level of the program logic that the program is safe to run under invariant I. Concretely, one needs to prove a specification for a complete safe execution (of

the form $\{P\} \rightsquigarrow \bullet$), given "points-to" resources in the pre-condition that correspond to the initial state of registers and memory. In particular, we get access to points-to resources for the adversary region (along the fact that they contain integers) and points-to resources for the region containing the program to execute.

Note that no resources are given for the domain of I as part of the initial resources for the complete-execution specification. Instead, these resources are part of the logical invariant under which the specification must be established (inside $\boxed{\ldots}$). This corresponds to the intuition that these resources should only be modified in a way that preserves invariant I. This logical invariant therefore specifies that there exists a subset of memory m, which covers the memory region defined by D, such that the invariant holds the corresponding points-to resources and such that $I(m)$ holds, i.e. the memory invariant I holds of this memory subset. (Recall from Section 4.1 that $\lceil\phi\rceil$ denotes an Iris proposition that asserts that the mathematical proposition ϕ holds.)

The reader may be surprised to notice that the region containing "adversarial" code has no special status. Indeed, it simply corresponds to a memory region containing (a priori unknown) integers. Nevertheless, remember that we ultimately want our program to be able to pass control to the unknown adversary code by jumping to the capability in r_0, as we have seen our example programs do. This means we need to have a way of reasoning about "what it will do", at least to ensure that it will not break our program's invariants.

In the next section, we show how to reason about whether unknown code can be considered "safe to execute", so that we can pass control to it while preserving previously established invariants.

5 REASONING ABOUT UNTRUSTED CODE IN CERISE

Code running on a capability machine is constrained by the set of capabilities it has access to. This is a crucial idea for reasoning about adversarial code. Whatever code the machine is running, if this code does not have access to a capability for, e.g., writing to a memory region, then it will not be able to modify memory in that region. In other words, one can prove a theorem describing the behavior of arbitrary code depending only on the capabilities it has access to.

One major technical contribution of this work is to formulate and mechanize such a theorem. Specifically, we are concerned with the preservation of invariants established in the program logic. We will thus give a definition of which machine words that are "safe" to share with unknown code. Informally, a word is safe if it cannot be used to break any previously established logical invariants. We will then prove that, as long as some arbitrary code only has access to safe machine words, its execution indeed preserves logical invariants.

Interestingly, we can establish this result while staying within the framework of the Cerise program logic exposed in the previous section. This illustrates the generality of said program logic: verifying specifications for known programs or specifying the behavior of arbitrary code are only two of its possible applications.

5.1 Logical Relation

Our formal definition of what makes a machine word *safe*, meaning "safe to share with unknown code", appears in Figure 9. It takes the form of a unary logical relation, defining simultaneously the notions of a machine word that is "safe to share" ($\mathcal{V}$) and "safe to execute" ($\mathcal{E}$). The names $\mathcal{V}$ and $\mathcal{E}$ originate from the tradition of logical relations, corresponding respectively to the "value relation" and the "expression relation", although this interpretation is perhaps less obvious in the setting of low-level machine code. We explain the definition in detail below. The intuition is that:

- *A value which is safe to share* only gives transitive access to other values that are safe to share, or code that is safe to execute (in the case of a sentry capability).

$$\mathcal{V}(w) \quad \begin{cases} \mathcal{V}(z), \mathcal{V}(\text{O}, -, -, -) & \triangleq \text{ True} \\ \mathcal{V}(\text{E}, b, e, a) & \triangleq \triangleright\, \square\, \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(\text{RW}/\text{RWX}, b, e, -) & \triangleq \ast_{a \in [b,e)} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(\text{RO}/\text{RX}, b, e, -) & \triangleq \ast_{a \in [b,e)} \exists P,\ \boxed{\exists w,\ a \mapsto w * P(w)} * \triangleright\, \square\, (\forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w)) \end{cases}$$

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

Fig. 9. Logical relation defining "safe to share" and "safe to execute" values.

- *A value which is safe to execute* allows the machine to run while preserving logical invariants (by definition of $\{\cdot;\cdot\} \rightsquigarrow \bullet$), provided the registers contain safe values.

Technically speaking, this informal definition is circular. Luckily, we can define it properly with the help of the "later" modality $\triangleright$. Iris provides us with a fixed-point operator that only requires recursive occurences to be guarded under a $\triangleright$, and we use that to formally define $\mathcal{V}$ and $\mathcal{E}$. Except for this technical requirement, the reader can in practice ignore the use of $\triangleright$ here.

Let us more closely examine the definition of $\mathcal{V}$, which is defined by case analysis on the shape of the given machine word w. If w is an integer (z), then it is always safe to share, since it cannot be used to access memory. Similarly, opaque capabilities with permission O are always safe as they also do not give access to memory.

A sentry capability E is safe to share if the code it encapsulates is safe to execute. Such a capability can be invoked at any moment and possibly several times: this is expressed through the use of the persistently modality $\square$. Technically speaking, this means that the property $\mathcal{E}(\text{RX}, b, e, a)$ must be established by only relying on persistent resources (typically, logical invariants) that will remain "available" throughout the entire execution.

A read-write capability RW or RWX gives read and write access to the memory region in its range. It is therefore safe as long as the words stored in the corresponding memory region are safe, and continue to be so when the memory gets modified. We thus say that it is safe when we have an invariant for each memory cell in the capability's region, which asserts ownership over the corresponding memory points-to resource, and asserts validity of its contents.

Finally, a capability with permission RO/RX cannot be used by unknown code to modify the memory words in its range. Therefore, these words can obey any property P as long as it entails safety ($\mathcal{V}$). Intuitively, the words in the interval have to be safe to share, because the adversary can read them. But since the adversary cannot modify them, it is possible to guarantee a stronger invariant about them. For instance, $P(w)$ could be the predicate "$w = 42$", describing that a value in the range stays equal to the integer 42.

Notice that this definition of safety does not distinguish between capabilities with permission RO and RX, or RW and RWX. This seems to strangely imply that permissions with the execute bit X have no additional expressive power over permissions without the execute bit. And indeed, *in terms of our model*—which "only" captures the ability to break memory invariants—their expressive power is the same![3] The crux of our main theorem (presented in the next sub-section) is that executing arbitrary code does not produce capabilities with more access to memory than was available before. Thus, being able to execute code within a memory region does not yield additional access to memory compared to what was available by simply reading the memory region (it only leads to additional machine behaviors).

[3]Having read-only permission over a region also allows one to simply copy the contents of the region into any other read-execute region and execute them here.

Is this definition of safety trivial? One might wonder whether the definition in Figure 9 is trivial, meaning that any machine word w will in fact be considered safe. This is thankfully not the case; let us illustrate concrete cases where a memory word w is *not* considered safe to share with unknown code.

At a high level, $\mathcal{E}$ is not trivial because establishing $\mathcal{E}(w)$ requires proving that a full execution of the machine, starting from w, *preserves logical invariants*. This requirement is not explicit in the definition, but comes from the definition of the Cerise program logic. The definition of $\mathcal{V}(w)$ is also not trivial because, e.g., in the case of an RW capability, it requires the memory points-to $a \mapsto -$ predicate to be part of a specific invariant, $\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$. Since the resource "$a \mapsto -$" is not duplicable, there can be only one resource $a \mapsto -$, which cannot be simultaneously part of two different invariants. Memory cells whose contents evolve according to an invariant more specific (less permissive) than the one above thus cannot be associated with a safe capability (according to $\mathcal{V}$).

What is a concrete example of a capability which is *not* safe? Let us consider a memory cell at address x initialized to 0. Let us assume the following Iris invariant: $\boxed{x \mapsto 0}$. This invariant expresses that x will contain the integer 0 for the rest of the execution. Then, a capability $(\text{RW}, x, x+1, x)$ is not safe to share with an adversary. Indeed, an adversary could use such a capability to write an arbitrary value at address x, thus invalidating the Iris invariant. (However, $(\text{RO}, x, x+1, x)$ would be safe.) A bit more formally speaking, it is not possible to prove $\mathcal{V}(\text{RW}, x, x+1, x)$, because it is not possible to create the invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$, as the resource for the memory cell x is already part of the invariant $\boxed{x \mapsto 0}$, and cannot be extracted to create a different invariant.

Similarly, one cannot prove $\mathcal{E}$ for a code fragment that writes another value than 0 at address x (after getting access to it through the pc register), because the proof would not be able to guarantee that the Iris invariant related to x is preserved at every step.

5.2 Fundamental Theorem

The Fundamental Theorem of our Logical Relation (Theorem 2) (hereafter, FTLR) establishes that any capability that is "safe to share" (in $\mathcal{V}$) is in fact "safe to execute" (in $\mathcal{E}$). In other words, if a capability only gives transitive access to safe capabilities, then it is safe to use it as a program counter capability and execute it: it will not be able to gain extra authority over memory or break any invariants. Importantly, this theorem is independent of the code that the capability points to, even though it is this code that will be executed. Hence the result applies to arbitrary code and we sometimes refer to it as a *universal contract* because of this.

THEOREM 2 (FTLR ⑦). *Let* $p \in \text{Perm}, b, e, a \in \text{Addr}$. *If* $\mathcal{V}(p, b, e, a)$, *then* $\mathcal{E}(p, b, e, a)$.

This is a non-trivial theorem, the proof of which requires checking all the possible cases of the semantics of each instruction of the machine. Indeed, one needs to check that there is no way for some machine instruction to create capabilities with further authority than what was available. This could, for example, happen if some runtime checks were missing, making it possible to create a capability $(\text{RW}, b, e+1, a)$ from a capability (RW, b, e, a). One can imagine how this would break expected security guarantees, and reveal a design or implementation bug of the machine. Therefore, another informal interpretation of the fundamental theorem is that it expresses that the capability machine "works well" or that it is *capability safe*.

The fundamental theorem provides a universal security property satisfied by unknown code, and gives us a way of verifying the correctness of known code that includes calls to possibly malicious code. To sum up, our logical relation characterizes the interface between a piece of verified code which wishes to preserve invariants on some internal state, and "external" arbitrary code whose accessible, safe capabilities have been sufficiently restricted.

It is important to note that the distinction between "known" and "adversary" code only exists at the logical level: there is no such distinction at runtime. We can have two components that have been verified separately, and that do not mutually trust each other. In this case, from the point of view of each component, the other component is considered as being the adversary.

Rules for program verification. From the general statement of the FTLR, we can derive two corollaries, which can be used to instantiate our adequacy theorem (Theorem 1) with a program that passes control to an unknown adversarial code region.

Corollary 1 (Unknown integers are safe (8)). *For* $m : [b, e) \to \text{Word}$,

$$\mathop{\Large *}_{(a,z)\in m} a \mapsto z * \ulcorner z \in \mathbb{Z} \urcorner \;\; -\!\!* \;\; \mathcal{V}(p, b, e, a)$$

Corollary 1 can be used to trade ownership over a memory region of integers to the knowledge that a capability over this region is safe.[4] Since integers can encode program instructions, we can typically use this rule to reason about a memory region containing an (unknown) program. The rule follows directly from the definition of $\mathcal{V}$ for values of p different from E; when $p = \text{E}$, an additional application of the FTLR (Theorem 2) is required.

Notice that the pre-condition of the rule matches the resources that one gets in the Adequacy theorem (Theorem 1) for the adversary region. When using the Adequacy theorem, we will thus be able to conclude that capabilities pointing to the adversary region are safe.

Corollary 2 (Jump to a safe word (9)).

$$\mathcal{V}(w) \;\; -\!\!* \;\; \triangleright \forall reg.\, \left\{\text{updatePcPerm}(w); \mathop{\Large *}_{(r,v)\in reg, r\neq \text{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

Corollary 2 gives us a specification for the execution of the machine after a jump to an unknown word w, assuming that w is safe. Recall that updatePcPerm(w) corresponds to the value of the program counter after jumping to w (see the machine semantics in Figure 6). The full execution specification in the conclusion of the rule requires that the machine registers contain safe values: indeed, we must only share safe words with unknown code.

An important application of Corollary 2 is to reason about the last instruction of a program encapsulated in a sentry (E) capability, where it "returns" and passes control to its caller by calling `jmp` on the (unknown but safe) return pointer held in r_0. In this scenario, the return pointer provided by the caller is unknown but safe, so Corollary 2 gives us a specification for the continuation of the program.

Additionally, Corollary 2 is typically used in combination with Corollary 1 when instantiating the Adequacy theorem. Indeed, in order to prove the complete safe execution specification required by the theorem, one typically needs to justify that one can `jmp` and pass control to an adversary region, given the resources granted by the Adequacy theorem.

5.3 Proving the fundamental theorem

To give a more in-depth perspective of the ideas behind the Fundamental Theorem, we detail in this sub-section one of the interesting cases of its proof. This sub-section can be safely skipped on a first read.

Proof. (FTLR) We begin by unfolding the definition of $\mathcal{E}$.

$$\forall reg.\, \left\{(p, b, e, a); \mathop{\Large *}_{(r,v)\in reg, r\neq \text{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

[4] We simplify the presentation here a bit and omit a view shift from the statement of Corollary 1. See the Coq development for the exact formal statement (8).

We proceed by Löb induction. The Löb rule is a powerful reasoning principle, which Cerise inherits from Iris, and which states that (in any context Q), if from $\triangleright P$ we can derive P, then we can also derive P without any assumptions.

$$\textsc{Löb}\ \frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

The idea of the rule is that "after we do some work", we will be able to remove the $\triangleright$ modality from the assumption, and reach the conclusion. In our case, this means reasoning about one step of execution, for one instruction. Intuitively, if we show that our property holds for the execution of one arbitrary instruction, then it must hold for a sequence of many instructions.

We thus let:

$$\text{IH} \triangleq \forall p, b, e, a.\, \mathcal{V}(p, b, e, a) \mathrel{-\!\!*} \forall reg.\, \left\{(p, b, e, a); \mathop{\ast}_{(r,v)\in reg, r\neq \text{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

and assume $\triangleright$ IH; we then wish to show IH.

First, we consider the case where (p, b, e, a) is not a valid program counter (for instance, if it contains a non-executable capability, or an integer). Then the machine configuration will step into a Failed configuration. In that case, any full execution specification ($\{\cdot;\cdot\} \rightsquigarrow \bullet$) trivially holds, and we are done.

In the case where (p, b, e, a) is a valid program counter, we will have to execute the next instruction of the program, pointed to by a. For (p, b, e, a) to be a valid program counter, the following needs to hold:

$$p \in \{\text{RX}, \text{RWX}\} \tag{1}$$

$$b \leq a < e \tag{2}$$

From (1), we can infer that $\mathcal{V}(p, b, e, a)$ will unfold to (at least) the following:

$$\mathop{\ast}_{a\in[b,e)} \exists P,\ \boxed{\exists w,\ a \mapsto w * P(w)}\ * \triangleright \square\, \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Since we know that a is an address in the range $[b, e)$ (2), we can in particular infer that there exists a predicate P such that $\triangleright \square\, \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w)$, for which the following invariant holds:

$$\boxed{\exists w,\ a \mapsto w * P(w)} \tag{3}$$

Ownership over $a \mapsto w$ is in fact required in order to apply any rule of the program logic (we need to be able to access memory for the instruction pointed to by pc). We will therefore first open the invariant (3) to get access to that resource.

Recall the invariant opening rule Inv (Section 4.2). According to that rule, we can get access to the resources held inside the invariant *now*, as long as we give them back *after one execution step*. Since we wish here to reason about the execution of a single instruction, this is a perfectly good deal.

Once the invariant has been opened, the following propositions are added to our assumptions, for some word w (technically speaking, the Iris context also tracks the fact that these facts come from an invariant and must be given back next, but we choose to hide these details):[5]

$$a \mapsto w \tag{4}$$

$$\triangleright\ P(w) \tag{5}$$

Because pc points to a, and address a contains the word w, w should correspond to the (encoding of the) instruction to execute now. We thus reason by case analysis on decode(w).

[5]Notice that we directly get $a \mapsto w$ rather than $\triangleright\, a \mapsto w$, due to the fact that memory points-to are timeless.

This leads to as many cases as there are instructions in the machine. We will now detail a sub-case for the `load` instruction, which is one of the interesting cases. Many of the other cases are similar in nature.

Case: decode(w) = `load` r_{dst} r_{src}.

We consider here the case where r_{dst} and r_{src} are two different registers, both different from pc. We also only consider the case where r_{src} contains a capability, which we are permitted to load from. In other words, our goal is as follows:[6]

$$\triangleright \text{IH} * a \mapsto w * \triangleright P(w)$$
$$\vdash \left\{ \begin{array}{l} \text{✻}_{(r,v)\in reg, r\neq \text{pc}, r_{dst}, r_{src}}\ r \Rightarrow v * \mathcal{V}(v) \\ (p,b,e,a); \ * \ r_{dst} \Rightarrow w' * \mathcal{V}(w') \\ \qquad * \ r_{src} \Rightarrow (p',b',e',a') * \mathcal{V}(p',b',e',a') \end{array} \right\} \rightsquigarrow \bullet$$

As stated, we assume that (p', b', e', a') permits us to load from a'. We can thus infer the following two properties:

$$p' \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}\} \tag{6}$$
$$b' \leq a' < e' \tag{7}$$

Just like before, we can from $\mathcal{V}(p', b', e', a')$ conclude that the following invariant holds, where P' is a predicate such that $\triangleright \square \forall w,\ P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$:

$$\boxed{\exists w,\ a' \mapsto w * P'(w)} \tag{8}$$

We consider the (more interesting) case where $a \neq a'$. We can thus open the invariant (since it has not been opened already), meaning that we have for some word w_{src} the following (again, plus some invariant-tracking resources not shown here):

$$a' \mapsto w_{src} \tag{9}$$
$$\triangleright\ P'(w_{src}) \tag{10}$$

With these assumptions, we now have all the necessary resources to take a step in the program logic, using the rule for the `load` instruction (Figure 8). A feature of single-instruction rules of our program logic is that they include a built-in $\triangleright$ modality. In other words, after applying a single-instruction rule, we are "one execution step later", and we can remove one occurrence of $\triangleright$ for each assumption of our context. In particular, this means that we can turn $\triangleright$ IH into IH, and similarly for $P(w)$ and $P'(w_{src})$. We now have to show:

$$\text{IH} * a \mapsto w * P(w) * a' \mapsto w_{src} * P'(w_{src})$$
$$\vdash \left\{ \begin{array}{l} \text{✻}_{(r,v)\in reg, r\neq \text{pc}, r_{dst}, r_{src}}\ r \Rightarrow v * \mathcal{V}(v) \\ (p,b,e,a+1); \ * \ r_{dst} \Rightarrow w_{src} \\ \qquad * \ r_{src} \Rightarrow (p',b',e',a') * \mathcal{V}(p',b',e',a') \end{array} \right\} \rightsquigarrow \bullet$$

We now have direct access to IH (our initial goal) as an assumption, so the proof is nearly done. Before we can invoke IH and conclude the goal, we must do two things: (a) close all the open invariants (as required by the invariant opening rule), and (b) show that the contents of all registers satisfies $\mathcal{V}$ (required by the definition of IH). (We actually need to show (b) *before* addressing (a), as we will make use of resources from the open invariants.)

Addressing (b), we already know that the contents of registers satisfy $\mathcal{V}$ for all registers except for r_{dst}—the only register whose contents were changed by the instruction. We must thus prove $\mathcal{V}(w_{src})$. Luckily, w_{src} is not a completely arbitrary word: it was accessible from available memory,

[6] We again omit details involving masks and update modalities, and refer to the Coq formalization for the full details.

so it must be safe as well. More precisely, from the invariant about a' (previously opened), we know that $P'(w_{src})$ holds, and furthermore we know that:

$$\square\, \forall w, P'(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Owing to the fact that $\mathcal{V}(\cdot)$ is persistent, we can shave off the $\square$ modality, and conclude that $\mathcal{V}(w_{src})$ holds, concluding the proof of (b).

Finally, addressing (a) is straightforward, since we did not change the contents of memory at either address a or a'. We can therefore close the invariants again, by giving up the same resources as we initially got from opening them, concluding the proof of (a) and thus the case of the proof for `load`.

In the proof sketch above, we followed one specific subcase of the proof for the `load` instruction. In the complete proof, we must go through all the possible cases of the semantics for the instruction. In some cases, the machine fails which terminates the proof easily (for instance, if the capability in r_{src} does not in fact allow reading memory, or if r_{src} does not in fact contain a capability). In some other cases, the machine does not fail, and the proof is similar to the case highlighted here but slightly different (for instance when r_{dst} and r_{src} are the same register).

The proofs for the other instructions of the machine follow a similar pattern. In particular, in the `store` case, the register state is not modified except for the pc register, but memory is modified. As such, closing the invariants is not as easy since we need to establish that the stored word is at least safe. This is established by using the fact that we assumed that the register only contains safe words. The case of the `restrict`, `subseg` and `lea` instructions require showing that a capability with smaller authority remains in the value relation $\mathcal{V}$, and the `jmp`, `jnz` and `mov` cases show that pc (or other registers) can be updated with arbitrary safe words. The other remaining cases are rather trivial, as they all only change a register state to an integer, which is always safe. $\square$

6 REASONING WITH CAPABILITIES: TWO EXAMPLES

In this section, we return to the motivational examples introduced in Section 2, and show how to prove that they enforce the desired properties, using Cerise's reasoning tools, laid out in the previous sections.

6.1 Sharing a sub-buffer with an unknown adversary

Let us recall (on the right) the code for our buffer-sharing program, previously introduced in Figure 3. The labels `code`, `data`, `secret` and `end` denote addresses in memory. We wish to prove formally that the program can share the data between addresses `data` and `secret` (excluded), while protecting the integrity of the data at address `secret`.

```
code: mov r1 PC
      lea r1 [data-code]
      subseg r1 [data] [data+3]
      jmp r0
data:   'H', 'i', 0, ; public
secret: 42           ; secret
end:
```

Using the reasoning rules from our program logic, we can first prove a specification for the program, specifying its behavior from its first instruction up until the final `jmp`. The corresponding specification is as follows, where code_instrs is the list of integers corresponding to the encoded instructions of the program, i.e., code_instrs = map encodeInstr [mov r_1 pc; ...; jmp r_0].

Lemma 1 (Program specification (10)).

$$\left\{ (\textit{RWX}, \text{code}, \text{end}, \text{code}); \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto - * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} \end{array} \right\} \rightsquigarrow$$
$$\left\{ \text{updatePcPerm}(w_{adv}); \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto (\textit{RWX}, \text{data}, \text{secret}, \text{data}) * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} \end{array} \right\}$$

One can read from the specification that executing the program will store in r_1 an RWX capability to the memory segment between addresses `data` and `secret` (our "buffer"), and pass control to the word w_{adv} found in register r_0.

Proving this specification is easy: it is enough to successively apply the program logic rule of each individual instruction found in the program.

This specification shows that the program ultimately jumps to the word initially passed in register r_0, but does not describe what happens after, in the case where this word points to a region containing unknown code. For this, we use the reasoning principles from Section 5.2 (built on top of the Fundamental Theorem), and derive a specification for a complete execution of the machine, see Lemma 2 below. The lemma specifies that, starting by executing our program, and given that r_0 contains a capability to a region containing unknown integers, then the machine is safe to run. Notice that we do not assume a points-to resource for the secret data: instead, this points-to will be part of an invariant—specifying that it contains the same secret value at every step—and we do not need to access that here.

Lemma 2 (Full execution specification (11)).

$$\left\{ (\textit{RWX}, \text{code}, \text{end}, \text{code}); \begin{array}{l} \mathsf{r}_0 \Mapsto (\textit{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto - * \\ \ast_{\substack{(r,v)\in reg, \\ r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} * \\ [\text{data}, \text{secret}) \mapsto [\text{'H'}; \text{'i'}; 0] * \\ \ast_{(a,z)\in adv}\, a \mapsto z * \lceil z \in \mathbb{Z} \rceil \end{array} \right\} \rightsquigarrow \bullet$$

Proof. By Lemma 1, the frame rule FragFrame and the sequence rule SeqFull, it suffices to show the following goal, which corresponds to a specification about the execution of the machine *after* the execution of the verified code:

$$\textit{Goal:} \quad \left\{ (\text{RWX}, b_{adv}, e_{adv}, b_{adv}); \begin{array}{l} \mathsf{r}_0 \Mapsto (\text{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathsf{r}_1 \Mapsto (\text{RWX}, \text{data}, \text{secret}, \text{data}) * \\ \ast_{\substack{(r,v)\in reg, \\ r\notin\{\mathsf{pc},\mathsf{r}_0,\mathsf{r}_1\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \ast_{(a,z)\in adv}\, a \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} * \\ [\text{data}, \text{secret}) \mapsto [\text{'H'}; \text{'i'}; 0] \end{array} \right\} \rightsquigarrow \bullet$$

We now rely on the reasoning rules derived from the Fundamental Theorem (Section 5.2). First, from the fact that the adversary region *adv* does not contain capabilities, we get using Corollary 1 that any capability on that region is safe, in particular we have $\mathcal{V}(\text{RWX}, b_{adv}, e_{adv}, b_{adv})$. Then, from Corollary 2 we get a specification for the execution of the machine starting from $\mathcal{V}(\text{RWX}, b_{adv}, e_{adv}, b_{adv})$ (recall that updatePcPerm is the identity on non-E capabilities):

$$\textit{Fact:} \quad \forall reg.\, \left\{ (\text{RWX}, b_{adv}, e_{adv}, b_{adv}); \ast_{(r,v)\in reg, r\neq \mathsf{pc}}\, r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

From this fact, we can prove our goal provided that we show that the contents of all machine registers satisfy $\mathcal{V}$. For registers other than $\mathtt{r}_0$ and $\mathtt{r}_1$, this holds by definition of $\mathcal{V}$, as we know they only contain integers. Register $\mathtt{r}_0$ contains a capability to the adversary region, which we have already proved to be safe using Corollary 1. Finally, register $\mathtt{r}_1$ contains the capability pointing to the public buffer. We can again leverage Corollary 1 to obtain $\mathcal{V}(\textsc{rwx}, \mathsf{data}, \mathsf{secret}, \mathsf{data})$ from the memory points-to for the buffer ($[\mathsf{data}, \mathsf{secret}) \mapsto [\mathtt{'H';'i';0}]$), thus concluding the proof. □

Finally, from Lemma 2, established in the program logic, we wish to obtain a final result in terms of the operational semantics of the machine. The toplevel end-to-end theorem that we obtain is shown in Theorem 3. We consider a machine whose memory is initially loaded with our program and unknown adversarial code, and that starts by executing our verified code. The theorem establishes that the adversary will not be able to tamper with the value held at address `secret`: at every step of the execution, it will be unchanged and equal to 42.

Theorem 3 (End-to-end theorem: integrity of the secret data is preserved (12)). *Starting from an initial state of the machine* (reg, mem) *where:*

- $prog \uplus adv \subseteq mem$, *for* $adv : [b_{adv}, e_{adv}) \to \mathrm{Word}$ *and* $prog : [\mathsf{code}, \mathsf{end}) \to \mathrm{Word}$
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities:* $\forall a.\, adv(a) \in \mathbb{Z}$;
- *the initial state of registers satisfies:*
  $reg(\mathtt{pc}) = (\textsc{rwx}, \mathsf{code}, \mathsf{end}, \mathsf{code})$,
  $reg(\mathtt{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$,
  $reg(r) \in \mathbb{Z}$ *otherwise;*

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(\mathsf{secret}) = 42$.

Proof. We first invoke Theorem 1, choosing the memory invariant I and its domain D to be the invariant I_{buf} and domain D_{buf} defined below, asserting that the value at address `secret` is equal to 42:

$$I_{buf} \triangleq \lambda m.\ m(\mathsf{secret}) = 42$$
$$\text{and}\ \ D_{buf} = \{\mathsf{secret}\}.$$

Most side-conditions of the adequacy theorem can be easily discharged. What remains is the following specification in Iris:

$$\textit{Goal:}\quad \boxed{\exists m, \mathop{\ast}_{(a,w)\in m} a \mapsto w * \lceil \mathrm{dom}(m) = D_{buf} \rceil * \lceil I_{buf}(m) \rceil} \vdash \left\{ (\textsc{rwx}, \mathsf{code}, \mathsf{end}, \mathsf{code});\ \begin{array}{l} \mathtt{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathop{\ast}_{\substack{(r,v)\in reg,\\ r\notin\{\mathtt{pc},\mathtt{r}_0\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \mathop{\ast}_{(a,z)\in adv} a \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \mathop{\ast}_{\substack{(a,w)\in prog,\\ a\notin D_{buf}}} a \mapsto w \end{array} \right\} \rightsquigarrow \bullet$$

We can simplify this goal by unfolding the definition of I_{buf}, D_{buf}, *prog* and massaging the goal to extract relevant points-to resources. The goal then becomes:

$$\textit{Goal:} \quad \boxed{\text{secret} \mapsto 42} \vdash \left\{ (\text{RWX}, \text{code}, \text{end}, \text{code}); \begin{array}{l} \mathrm{r}_0 \Mapsto (\text{RWX}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathrm{r}_1 \Mapsto - * \\ \ast_{\substack{(r,v)\in reg, \\ r\notin\{\mathrm{pc},\mathrm{r}_0,\mathrm{r}_1\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\text{code}, \text{data}) \mapsto \text{code\_instrs} * \\ [\text{data}, \text{secret}) \mapsto [\text{'H'}; \text{'i'}; 0] * \\ \ast_{(a,z)\in adv}\, a \mapsto z * \lceil z \in \mathbb{Z} \rceil \end{array} \right\} \rightsquigarrow \bullet$$

Note how the points-to resource for the `secret` address is held as part of the invariant, asserting that it contains the value 42 at each step. This simplified goal now follows from the full execution specification established earlier in Lemma 2 by applying the rule FullFrame, which concludes the proof. □

6.2 Creating a closure around local state

Let us now come back to the example introduced in Section 2.4, whose code is reproduced below. In this example, the control flow is somewhat more involved, as we have two separate pieces of known code that run at different times. The initialization code between init and code runs first, and creates a sentry capability before passing control to the unknown code. The code and data located between code and end are encapsulated in the sentry capability created by the initialization code. Because the sentry capability is exposed to the unknown code, the code it encapsulates may be invoked several times, incrementing the value of the counter each time.

We wish to prove formally that the value of the counter is correctly encapsulated. We prove that it remains non-negative at every step: starting from zero, it can only get incremented by the code routine encapsulated in the sentry capability.

```
init:
  mov r1 PC
  lea r1 [data-init]
  mov r2 r1
  lea r2 1
  store r1 r2
  lea r1 [code-data]
  subseg r1 [code] [end]
  restrict r1 E
  mov r2 0
  jmp r0
```

```
code:
  mov r1 PC
  lea r1 [data-code]
  load r1 r1
  load r2 r1
  add r2 r2 1
  store r1 r2
  mov r1 0
  jmp r0
data:
  ; will be:
  ; (RWX, init, end, data+1)
  0xFFFF,
  0 ; counter value
end:
```

Using the rules of our program logic, we can first prove a specification for the initialization code, shown in Lemma 3. This specification describes the behavior of the code between init and code, where init_instrs denote the corresponding list of encoded instructions.

Lemma 3 (Specification for the initialization code (13)).

$$\left\{(\mathit{RWX}, \mathsf{init}, \mathsf{end}, \mathsf{init});\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto - * \\ \mathsf{data} \mapsto - * [\mathsf{init}, \mathsf{code}) \mapsto \mathsf{init\_instrs} \end{array}\right\} \rightsquigarrow$$
$$\left\{\mathsf{updatePcPerm}(w_{adv});\ \begin{array}{l} \mathsf{r}_0 \Mapsto w_{adv} * \mathsf{r}_1 \Mapsto (\mathit{E}, \mathsf{code}, \mathsf{end}, \mathsf{code}) * \mathsf{r}_2 \Mapsto 0 * \\ \mathsf{data} \mapsto (\mathit{RWX}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1) * [\mathsf{init}, \mathsf{code}) \mapsto \mathsf{init\_instrs} \end{array}\right\}$$

From this specification, one can read that running the initialization code will store in register r_1 a sentry capability to [code, end), and write at address data an RWX capability pointing to the location holding the counter value. The initialization code then passes control to the unknown word w_{adv} stored in r_0.

We can also use the program logic rules to prove a specification for the code routine in [code, data) which increments the counter, and which will run each time the sentry capability is invoked. The specification appears in Lemma 4, where code_instrs refers to the list of encoded instructions for the routine.

Lemma 4 (Specification for the increment routine (14)).

$$\begin{array}{l} \boxed{[\mathsf{code}, \mathsf{data}) \mapsto \mathsf{code\_instrs}}, \\ \boxed{\mathsf{data} \mapsto (\mathit{RWX}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \boxed{\exists n.\ (\mathsf{data}+1) \mapsto n * \lceil n \geq 0 \rceil} \\ \vdash \{(\mathit{RX}, \mathsf{code}, \mathsf{end}, \mathsf{code}); \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto -\} \rightsquigarrow \\ \quad \{\mathsf{updatePcPerm}(w_{cont}); \exists n.\ \mathsf{r}_0 \Mapsto w_{cont} * \mathsf{r}_1 \Mapsto 0 * \mathsf{r}_2 \Mapsto n\} \end{array}$$

This specification assumes a number of Iris invariants, describing the contents of the [code, end) memory region. Indeed, because the increment routine is invoked by unknown code, it cannot make many assumptions about the state of the machine. The only thing that it can assume is that previously established invariants still hold (because, by definition, capability-safe unknown code has to preserve invariants).

The specification thus assumes, as invariants: 1) that the region [code, data) contains the code of the routine; 2) that data contains the RWX capability to the counter value previously stored there by the initialization code, and finally 3) that the counter value (at address data + 1) is a non-negative integer.

The specification asserts that the routine can run, starting with pc containing an RX capability to the [code, end) region, while preserving the invariants. (In particular, this means that incrementing the counter indeed preserves the fact that it is a non-negative integer.) Recall that the RX permission in pc corresponds to what one gets after jumping to a sentry capability.

Finally, we prove as before a specification proving safety of complete executions, starting from the initialization code, then followed by the execution of unknown code, including its possible invocations of the sentry capability. This specification appears below in Lemma 5.

Lemma 5 (Full execution specification (15)).

$$\boxed{\exists n.\ (\mathsf{data}+1) \mapsto n * \lceil n \geq 0 \rceil}$$
$$\vdash \left\{(\mathit{RWX}, \mathsf{init}, \mathsf{end}, \mathsf{init});\ \begin{array}{l} \mathsf{r}_0 \Mapsto (\mathit{RWX}, b_{adv}, e_{adv}, b_{adv}) * \mathsf{r}_1 \Mapsto - * \mathsf{r}_2 \Mapsto - * \\ \ast_{\substack{(r,v)\in reg, \\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}}\ r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ [\mathsf{init}, \mathsf{code}) \mapsto \mathsf{init\_instrs} * \\ [\mathsf{code}, \mathsf{data}) \mapsto \mathsf{code\_instrs} * \mathsf{data} \mapsto - * \\ \ast_{(a,z)\in adv}\ a \mapsto z * \lceil z \in \mathbb{Z} \rceil \end{array}\right\} \rightsquigarrow \bullet$$

Proof. By using Lemma 3 (the specification for the initialization code), the frame rule FragFrame and sequence rule SeqFull, it is enough to show the following goal, which specifies the execution

of the machine after the initialization code has run:

$$\textit{Goal:}\quad \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \vdash \left\{ \begin{array}{ll} (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}); & \mathsf{r}_0 \Rightarrow (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ & \mathsf{r}_1 \Rightarrow (\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code}) * \mathsf{r}_2 \Rightarrow 0 * \\ & \circledast_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}} r \Rightarrow z * \lceil z \in \mathbb{Z}\rceil * \\ & [\mathsf{init},\mathsf{code}) \mapsto \mathsf{init\_instrs} * \\ & [\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs} * \\ & \mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1) * \\ & \circledast_{(a,z)\in adv} a \mapsto z * \lceil z \in \mathbb{Z}\rceil \end{array} \right\} \rightsquigarrow \bullet$$

We then allocate two new invariants, one containing the code of the sentry capability, the other the points-to resource at address data.

$$\textit{Goal:}\quad \boxed{[\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs}}, \boxed{\mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \vdash \left\{ \begin{array}{ll} (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}); & \mathsf{r}_0 \Rightarrow (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ & \mathsf{r}_1 \Rightarrow (\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code}) * \mathsf{r}_2 \Rightarrow 0 * \\ & \circledast_{\substack{(r,v)\in reg,\\ r\notin\{\mathsf{pc},\mathsf{r}_0..\mathsf{r}_2\}}} r \Rightarrow z * \lceil z \in \mathbb{Z}\rceil * \\ & [\mathsf{init},\mathsf{code}) \mapsto \mathsf{init\_instrs} * \\ & \circledast_{(a,z)\in adv} a \mapsto z * \lceil z \in \mathbb{Z}\rceil \end{array} \right\} \rightsquigarrow \bullet$$

From Corollary 1 and the fact that the adversary region *adv* does not contain capabilities, we get that any capability on that region is safe, and therefore that $\mathcal{V}(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ holds. From Corollary 2, we get that a full execution starting from $(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$ is safe:

$$\textit{Fact:}\quad \forall reg.\,\left\{(\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}); \circledast_{(r,v)\in reg, r\neq \mathsf{pc}} r \Rightarrow v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

In combination with rule FullFrame, this fact allows us to conclude the proof, *provided we can prove safety of values stored in all registers*. We have already proved the capability in r_0 to be safe. Registers r_2 to r_{31} contain integers, so they are safe by definition of $\mathcal{V}$. Safety of the sentry capability created by the initialization code and stored in r_1 remains to be proven.

$$\textit{Goal:}\quad \boxed{[\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs}}, \boxed{\mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \vdash \mathcal{V}(\textsc{e}, \mathsf{code}, \mathsf{end}, \mathsf{code})$$

By definition of $\mathcal{V}$ and $\mathcal{E}$, this goals unfolds to the following:

$$\textit{Goal:}\quad \boxed{[\mathsf{code},\mathsf{data}) \mapsto \mathsf{code\_instrs}}, \boxed{\mathsf{data} \mapsto (\textsc{rwx}, \mathsf{init}, \mathsf{end}, \mathsf{data}+1)}, \boxed{\exists n.\,(\mathsf{data}+1)\mapsto n * \lceil n \geq 0\rceil} \vdash \triangleright \square\, \forall reg,\,\left\{(\textsc{rx}, \mathsf{code}, \mathsf{end}, \mathsf{code}); \circledast_{(r,v)\in reg, r\neq \mathsf{pc}} r \Rightarrow v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

For technical reasons, we can shave off the later modality ($\triangleright$) in front of the goal (we refer to the Coq formalization for more details). The persistent modality ($\square$) is more interesting: it expresses the fact that safety of the callback should only depend on persistent assumptions. This corresponds to the fact that the callback may be invoked several times, in future execution states and because of this it cannot rely on non-persistent assumptions that only hold at the callback's creation time. Fortunately, invariants are persistent, so they remain available for proving the callback's safety.

Then, let us name w_0 the contents of register r_0: we get to assume $\mathcal{V}(w_0)$ (as for the contents of other registers). By using Lemma 4 (the specification for the increment routine) with rules

FragFrame and SeqFull, it is enough to prove the following goal, which asserts safety of the execution *after* passing control back to unknown code by jumping to w_0:

$$\textit{Goal:} \quad \vdash \left\{ \text{updatePcPerm}(w_0);\ \begin{array}{l} \exists n.\, \mathrm{r}_0 \Mapsto w_0 * \mathrm{r}_1 \Mapsto 0 * \mathrm{r}_2 \Mapsto n * \\ \ast_{(r,v)\in reg, r\notin\{\mathrm{pc},\mathrm{r}_0,\mathrm{r}_1,\mathrm{r}_2\}}\ r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Informally, the increment routine returns to the unknown code by passing control to some unknown word provided in r_0: it is safe to do so, since such word can be assumed to be itself safe. Formally speaking, we know $\mathcal{V}(w_0)$, so we apply Corollary 2 which concludes the proof. □

Similarly to the previous example, we derive from Lemma 5 a toplevel theorem which only refers to the operational semantics of the machine, shown below in Theorem 4. We consider a machine initially loaded with our program and unknown adversarial code. The theorem establishes that the value of the counter is properly encapsulated: at every step of the execution, it will be a non-negative integer.

Theorem 4 (End-to-end theorem: integrity of the counter value is preserved (16)). *Starting from an initial state of the machine* (reg, mem) *where:*

- $prog \uplus adv \subseteq mem$, *for* $adv : [b_{adv}, e_{adv}) \to \text{Word}$ *and* $prog : [\text{init}, \text{end}) \to \text{Word}$
- *the contents of prog correspond to the encoded instructions and program data;*
- *the adversary memory contains no capabilities:* $\forall a.\, adv(a) \in \mathbb{Z}$*;*
- *the initial state of registers satisfies:*
  $reg(\mathrm{pc}) = (\textsc{rwx}, \text{init}, \text{end}, \text{init})$,
  $reg(\mathrm{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv})$,
  $reg(r) \in \mathbb{Z}$ *otherwise;*

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(\text{data} + 1) \geq 0$.

Proof. We invoke Theorem 1, with invariant and domain I_{cnt} and D_{cnt} defined as follows:

$$I_{cnt} \triangleq \lambda m.\, m(\text{data} + 1) \geq 0$$
$$\text{and } D_{cnt} = \{\text{data} + 1\}$$

The main step of the proof is to show that the full execution specification for the initial machine configuration holds, as stated by the theorem. After some basic unfolding of definitions, it is easy to show that it follows from the specification we previously established in Lemma 5. □

7 DYNAMIC MEMORY ALLOCATION AND CLOSURES

In the previous sections, we have shown how to use capabilities for memory protection and compartmentalization in the setting of relatively simple scenarios. In particular, the examples that we have presented so far only relied on memory allocated statically as part of the initial program region.

We now investigate how we can use and reason about more complicated programming patterns. More precisely, we show how we can implement features found in higher-level languages, such as dynamic memory allocation and function calls which guarantee encapsulation of local variables. Additionally, we implement an `assert` routine which we use to formally express properties about dynamically allocated memory.

This section focuses on presenting the aforementioned higher-level building blocks (§7.1–7.3), an updated adequacy theorem that incorporates the use of these components (§7.4), then followed by a simple illustrative example (§7.5). In Section 8, we then apply them to build a larger, more significant case study, demonstrating how these building blocks can work at scale.

7.1 Dynamic memory allocation as a library routine

We show how dynamic memory allocation can be implemented as a library, for which: 1) we prove an Iris specification making it usable from verified code, and 2) we show that it is safe to share with untrusted code, so that an adversary can also use the library to allocate memory for its own uses.

Note that this task is made easier by the fact that we do not attempt to provide a way of deallocating memory. As such, memory provided by the allocation routine is never reclaimed. We leave deallocation for future work, as it likely requires a significantly more complex runtime mechanism to ensure that no dangling capabilities remain pointing to previously allocated memory regions [Filardo et al. 2020; Xia et al. 2019].

Concretely, we implement our allocator library as a simple bump-pointer allocator. The library provides a `malloc` entry point, to be called with an integer argument n, which works as follows:

(1) the routine encapsulates a contiguous region of memory $[b, e)$, as well as a capability (RWX, b, e, a) where the interval $[b, a)$ represents already allocated memory, and $[a, e)$ represents memory that can still be allocated;
(2) the routine checks that the input size n is strictly positive;
(3) if $a + n$ is greater than e, the routine fails (there is not enough memory available);
(4) otherwise, it then records that memory has been allocated by updating its internal capability to $(\text{RWX}, b, e, a + n)$, and returns to the caller the capability $(\text{RWX}, a, a + n, a)$.

Figure 10 outlines the code for our simple `malloc` implementation. The code assumes that it is stored in memory in an interval $[b_m, b_{mid})$ and that b_{mid} points to a capability $(\text{RWX}, b_{mid}, e_m, a)$ giving access to: itself (so it can be updated), and the memory pool (between address $b_{mid} + 1$ and e_m). For simplicity, we assume that the non-allocated memory is already initialized to 0. These requirements are represented by the following invariant (17):

$$\text{mallocInv}(b_m, e_m) \triangleq \boxed{\begin{array}{ll} \exists b_{mid}, a, & [b_m, b_{mid}) \mapsto \texttt{malloc\_instrs} * \\ & b_{mid} \mapsto (\text{RWX}, b_{mid}, e_m, a) * \\ & [a, e_m) \mapsto [0 \cdots 0] * \\ & \lceil b_{mid} < a \leq e_m \rceil \end{array}}$$

The core property of our safe `malloc` is that is does not hand out the same addresses across multiple dynamic allocations. This can be expressed elegantly in separation logic, by specifying that `malloc` hands out points-to resources for the allocated memory. Indeed, points-to resources $(a \mapsto w)$ express full ownership over the data at address a: possessing a resource $a \mapsto w$ guarantees that one is the only owner of address a.

Consequently, remark that the invariant holds memory points-to for the region corresponding to non-allocated memory (between a and e_m), but not for the memory that has already been allocated (between $b_{mid} + 1$ and a): these resources have been handed out to previous callers of the library.

We show below the specification for `malloc` (18). First, note that because `malloc` can fail if it runs out of memory or is given a wrong size, the specification documents that the resulting execution state is either Running or Failed. In the case where it does not fail, we can read that `malloc` hands out points-to resources for the allocated range in its post-condition: this expresses the fact that no

```
;; r1: integer determining the number
;; of words to allocate
;;
;; malloc fails if size <= 0 or if it
;; does not have enough space left
;;
;; returns in r1 a capability to the
;; allocated memory
bm:
  lt r3 0 r1 ;; check that size > 0
  mov r2 pc  ;; jmp after fail if
  lea r2 4   ;; yes; continue and
  jnz r2 r3  ;; fail if not
  fail
xm:
  mov r2 pc
  lea r2 [bmid - xm]
  ;; r2 = (RWX, bm, em, bmid)
  load r2 r2 ;; r2 = (RWX, bmid, em, a)
  geta r3 r2
  lea r2 r1
  ;; r2 = (RWX, bmid, em, a+size)
  geta r1 r2
  mov r4 r2
  subseg r4 r3 r1
  sub r3 r3 r1
  lea r4 r3
  mov r3 r2
  sub r1 0 r1
  lea r3 r1
  getb r1 r3
  lea r3 r1   ;; r3 = (RWX, bmid, em, bmid)
  store r3 r2 ;; bmid <- (RWX, bmid, em, a+size)
  mov r1 r4   ;; r1 = (RWX, a, a+size, a)
  mov r2 0
  mov r3 0
  mov r4 0
  jmp r0
bmid: (RWX, bmid, em, a)
;; ... already allocated memory ...
a:
;; ... free memory ...
em:
```

Fig. 10. A simple `malloc` subroutine

piece of code but the caller of `malloc` can access the newly allocated memory.

$$\boxed{\text{mallocInv}(b_m, e_m)}$$

$$\vdash \left\{ (\text{RX}, b_m, e_m, b_m);\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto n * \\ r_2, r_3, r_4 \Mapsto - \end{array} \right\} \rightsquigarrow$$

$$\left\{ s.\ \begin{array}{l} \lceil s = \text{Running} \rceil * \text{pc} \Mapsto \text{updatePcPerm}(w_0) * \\ \quad \exists b_a, e_a, \lceil b_a + n = e_a \rceil * \\ \quad\quad \text{r}_0 \Mapsto w_0 * \\ \quad\quad \text{r}_1 \Mapsto (\text{RWX}, b_a, e_a, b_a) * \\ \quad\quad \ast_{a \in [b_a, e_a)}\ a \mapsto 0 * \\ \quad\quad \text{r}_2, \text{r}_3, \text{r}_4 \Mapsto 0 \\ \quad \vee \lceil s = \text{Failed} \rceil \end{array} \right\}$$

The `malloc` routine can furthermore be encapsulated using a sentry capability, which can be shown to be safe to share with an adversary (Lemma 6).

Lemma 6 (`malloc` is safe (19)). $\text{mallocInv}(b_m, e_m) \mathrel{-\!\!*} \mathcal{V}(E, b_m, e_m, b_m)$

The proof is comparable to the proof that $\mathcal{V}(\text{E}, \text{code}, \text{end}, \text{code})$ on page 33. It relies on the `malloc` specification and the fundamental theorem.

7.2 Runtime checks: an `assert` routine

The final end-to-end theorems presented so far in Section 6 rely on establishing that a certain memory location satisfies a given invariant. This requires the relevant location is statically allocated in memory and thus known in advance, thus making it easy to tie it to an Iris invariant.

However, when using our `malloc` routine, we typically wish to enforce properties about the contents of dynamically allocated memory locations, whose address is, by definition, not known in advance. To address this issue, we implement an `assert` routine, to be linked alongside programs relying on `malloc`. One can invoke `assert` to dynamically test whether the contents of two registers are equal; if the test fails, `assert` sets a flag "assert has failed" at a fixed location in memory.

The idea is then that, to assert that some property holds about a piece of dynamically allocated memory, one can check dynamically whether it holds using `assert`. Then, one can *prove* that each `assert` check succeeds (meaning that the property indeed holds). Consequently, as a property of the whole execution, one gets that, at every step, the assert flag (initialized at 0) remains at 0 and is never set to 1 by `assert`.

The private memory of the `assert` routine is described by the following invariant (20):

$$\mathrm{assertInv}(b_a, e_a, a_{flag}) \triangleq \boxed{\begin{array}{l}\exists a_{cap},\ [b_a, a_{cap}) \mapsto \texttt{assert\_instrs} * \\ \qquad\quad a_{cap} \mapsto (\textsc{rw}, a_{flag}, a_{flag}+1, a_{flag}) * \\ \qquad\quad \lceil a_{cap}+1 = a_{flag} \wedge a_{flag}+1 = e_a \rceil\end{array}}$$

The address a_{flag} denotes the address of the "assert flag", which is initialized to 0 and set to 1 by the routine in case of failure. As we are interested in using `assert` in programs where we can prove that the equality check succeeds, we establish the following specification (21), which asserts in a separate invariant that a_{flag} remains at 0. Registers r_4 and r_5 contain the two integers which are compared by the routine; we thus require that they are equal.

$$\boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$
$$\vdash \left\{ (\textsc{rx}, b_a, e_a, b_a);\ \begin{array}{l} \mathrm{r}_0 \Rightarrow w_0 * \\ \mathrm{r}_4 \Rightarrow n * \\ \mathrm{r}_5 \Rightarrow n \end{array} \right\} \rightsquigarrow \left\{ \mathrm{updatePcPerm}(w_0);\ \begin{array}{l} \mathrm{r}_0 \Rightarrow w_0 * \\ \mathrm{r}_4, \mathrm{r}_5 \Rightarrow 0 \end{array} \right\}$$

Note that, as opposed to `malloc`, the `assert` routine should only be shared with *verified* code, which calls it according to the specification above. Were `assert` shared with an unknown adversary, the adversary could simply call the routine with two different integers, setting the flag to 1, thus invalidating any guarantees established by verified code. Technically speaking, we cannot prove safety of the `assert` routine from the specification above: if we try to prove $\mathcal{V}(\textsc{e}, b_a, e_a, b_a)$, then we get that registers r_4 and r_5 contain two unknown (valid) words, which could be two different integers. In that case, we cannot use the specification above, as we would violate the invariant specifying that a_{flag} stays at 0.

7.3 A secure heap-based calling convention

We define a heap-based calling convention that uses `malloc` to dynamically allocate activation records. An activation record is encapsulated in a closure that reinstates its caller's local state, and continues execution from its point of creation. Conceptually, our heap-based calling convention can be seen as a continuation-passing style calling convention (one passes control to the callee, giving it a continuation for returning to the caller). This is similar to the calling convention that was used for instance in the SML/NJ compiler to implement an extension of Standard ML with call/cc [Appel 1992] (in the setting of a traditional computer architecture).

In the setting of a capability machine, our calling convention is furthermore *secure* in the sense that it enforces local state encapsulation. In other words, one can use it to pass control to unknown adversarial code, while protecting local data of the caller, thanks to the use of sentry capabilities to implement the continuation. Note that this calling convention does not enforce well-bracketed

```
; initially, PC = (RWX, code, end, a)
;            target = register containing the address to jump to
;            locals, params = lists of register names
; locals, params and target are parameters of the macro;
; they are in practice instantiated with concrete values
code:
  ...
a:
  malloc (length locals)        ; 1. allocate and store local state
  store_locals r1 locals
  mov r6 r1
  malloc 7                      ; 2. allocate region for activation record
  mov r0 r1
  store act_instr1              ;    store the activation code
  lea r0 1
  ...
  store act_instr5
  lea r0 1
  store r0 r6                   ;    store the capability to locals
  lea r0 1
x:
  mov r1 pc                     ;    prepare and store the continuation
  lea r1 [cont - x]
  store r0 r1
  lea r0 -6                     ; 3. create the return capability
  restrict r0 E
  rclear RegName\{PC,r0,r1} ∪ params ; 4. clear all registers except parameters
  jmp target                    ; 5. jump to target
cont:
  restore_locals r1 locals      ; 6. reinstate local state
  ...
data:
  (RO, table, end, table)       ; environment table
table:
  (E, bm, em, bm)               ; entry point to the malloc subroutine
  ...                           ; possibly other routines
end:
```

Fig. 11. Heap-based calling convention, with a the first instruction in the call macro

control flow (another desirable property); see [Georges et al. 2021; Skorstengaard et al. 2019a,b] for stack-based calling conventions that do.

We provide a `call` macro implementing the calling convention, invoked as `call` *target locals params*, where *target* is the name of the register containing a pointer to the code to invoke, *locals* is the list of registers whose content corresponds to the local state to reinstate upon return, and *params* is the list of registers containing the parameters to the call (passed to the callee). Its implementation appears in Figure 11, and a representation of the corresponding memory layout in Figure 12. (Because `call` is defined as a macro, its code is used inline as part of a bigger program, here stored between addresses code and end.)

Before passing control to the callee, the call macro does the following:

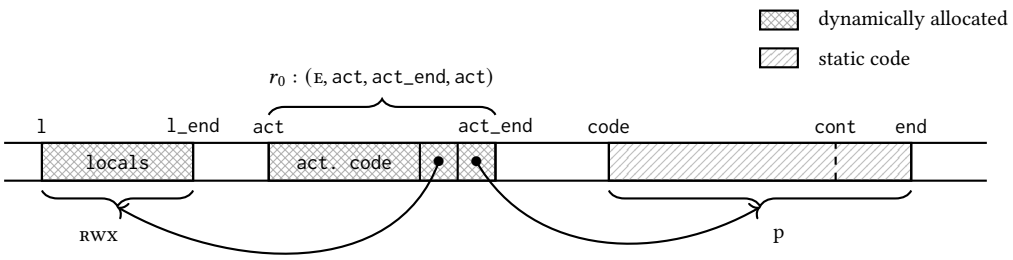


Fig. 12. Memory layout dynamically created by the calling convention

(1) Invoke `malloc` to dynamically allocate a region of memory $[l, l_{end})$ to store the local state from the registers specified in *locals*.
(2) Allocate a region of memory $[act, act_{end})$ to store the activation record, composed of: activation code, a capability to the region $[l, l_{end})$, and a capability to the instruction of the program following the call.
(3) Create a sentry capability (E, act, act_{end}, act) encapsulating the activation record; this is capability for returning to the caller which is passed to the callee.
(4) Clear all registers except those in *params*.
(5) Jump to *target*.

When the callee passes back control to the caller by jumping to the continuation, the code stored in the activation run first. It loads the capability pointing to local state, and returns to the old program counter set up by the call macro. As the last step, the macro will finally:

6. Restore the local state into the relevant registers from the activation record.

We show below the specification for the code of the macro up to step 5 (the jump to the target address) (22). Since the `malloc` routine is invoked by the macro, the specification relies on the corresponding invariant for `malloc`. The parameters of the macro are *params*, *locals* and *target*, respectively denoting the list of registers containing the parameters to the call, the list of registers containing local state, and the register containing the capability to jump to. The list of (encoded) instructions `act_instrs` denote the concrete instructions making up the activation code (in Figure 11 they are written as `act_instr1`...`act_instr5` (23)), which are not shown here for simplicity.

The post-condition of the specification describes the state immediately after the jump, where: the activation record has been allocated and initialized in $[act, act_{end})$; register r_0 contains an enter capability pointing to the activation record, and the local data has been copied to a newly allocated region $[l, l_{end})$.

$$\boxed{\text{mallocInv}(b_m, e_m)}$$

$$\vdash \left\{ (p, \text{code}, \text{end}, \text{a});\ \begin{array}{l} [\text{a}, \text{cont}) \mapsto \texttt{call\_instrs} * \\ \text{data} \mapsto (\textsc{ro}, \text{table}, \text{end}, \text{table}) * \text{table} \mapsto (\textsc{e}, b_m, e_m, b_m) * \\ \mathit{params} \Rrightarrow \mathit{pws} * \mathit{locals} \Rrightarrow \mathit{lws} * \mathit{target} \Rrightarrow w_{adv} * \\ \ast_{\substack{(r,v)\in reg,\\ r\notin\{\text{pc},target\}\\ r\notin params\cup locals}} \quad r \Rrightarrow v \end{array} \right\} \rightsquigarrow$$

$$\left\{ \text{updatePcPerm}(w_{adv});\ \begin{array}{l} \exists act, act_{end}, l, l_{end}, reg', \\ \text{r}_0 \Rrightarrow (\textsc{e}, act, act_{end}, act) * \\ \text{data} \mapsto (\textsc{ro}, \text{table}, \text{end}, \text{table}) * \text{table} \mapsto (\textsc{e}, b_m, e_m, b_m) * \\ \mathit{params} \Rrightarrow \mathit{pws} * \mathit{target} \Rrightarrow w_{adv} * [l, l_{end}) \mapsto \mathit{lws} * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} +\!\!+ [(\textsc{rwx}, l, l_{end}, l_{end}); \\ \qquad\qquad (p, \text{code}, \text{end}, \text{cont})] * \\ \ast_{\substack{(r,v)\in reg',\\ r\notin\{\text{pc},target,\text{r}_0\}\\ r\notin params}} \quad r \Rrightarrow v \end{array} \right\}$$

It is then up to the user of the `call` macro to establish that the capability in r_0 is safe to share with the (possibly unknown) callee. This can be done with the help of the specification for the activation code (24), shown next:

$$\vdash \left\{ (\textsc{rx}, act, act_{end}, act);\ \begin{array}{l} \text{r}_1 \Rrightarrow - * \text{r}_2 \Rrightarrow - * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} +\!\!+ \\ \qquad [(\textsc{rwx}, l, l_{end}, l_{end}); \\ \qquad (p, \text{code}, \text{end}, \text{cont})] \end{array} \right\} \rightsquigarrow$$

$$\left\{ (p, \text{code}, \text{end}, \text{cont});\ \begin{array}{l} \text{r}_1 \Rrightarrow - * \text{r}_2 \Rrightarrow (\textsc{rwx}, l, l_{end}, l) * \\ [act, act_{end}) \mapsto \texttt{act\_instrs} +\!\!+ \\ \qquad [(\textsc{rwx}, l, l_{end}, l_{end}); \\ \qquad (p, \text{code}, \text{end}, \text{cont})] \end{array} \right\}$$

One can read from this specification that the activation code passes control back to the caller (at address cont), while loading in register r_2 a capability to the region holding the local state, which can be then loaded back into the corresponding registers by the `restore_locals` macro (step 6, which we do not detail here).

To sum up, the calling convention presented here allows one to make a "function call" as one would do in a higher-level language, while protecting local data of the caller. The code invoked this way can be completely untrusted: in particular, it does not need to implement the calling convention itself for the local state encapsulation guarantees to hold. (But of course it might never "return" and pass control back to the caller.)

In Section 7.5, we demonstrate the use of this heap-based calling convention on a simple example, showing the interaction of its local state encapsulation guarantees with read-only capabilities.

7.4 Adequacy in the Presence of Dynamically Allocated Memory

We can now provide an updated version of the adequacy theorem (Theorem 1) which directly incorporates the `malloc` and `assert` library routines. Instead of establishing that a memory invariant is always preserved at each step, the new adequacy theorem establishes that the flag held by `assert` is never modified.

Theorem 5 assumes that the `malloc` and `assert` routines are loaded in memory disjoint from both *prog* and *adv*. Furthermore, the `assert` routine must have its flag initialized to 0. The verified

Theorem 5 (Updated adequacy (25)). *Given memory fragments prog* : $[b, e) \longrightarrow$ Word, *malloc* : $[b_m, e_m) \longrightarrow$ Word, *assert* : $[b_a, e_a) \longrightarrow$ Word, *and for any memory fragment adv* : $[b_{adv}, e_{adv}) \longrightarrow$ Word, *assuming that:*

(1) *the initial state of memory mem satisfies:*

$$prog \uplus malloc \uplus assert \uplus adv \subseteq mem$$

(2) $[b_m, e_m)$ *contains the* `malloc` *routine;*
(3) $[b_a, e_a)$ *contains the* `assert` *routine and its flag at address* a_{flag}*;*
(4) *the assertion flag is initially set to 0:*

$$mem(a_{flag}) = 0$$

(5) *prog contains a table linking to malloc and assert:*

$$\begin{aligned} \exists data, table, mem(data) &= (\textsc{ro}, table, table + 2, table) \\ mem(table) &= (\textsc{e}, b_m, e_m, b_m) \\ mem(table + 1) &= (\textsc{e}, b_a, e_a, b_a) \end{aligned}$$

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

$$\begin{aligned} &\exists data_{adv}, table_{adv}, \forall a \in \mathrm{dom}(adv) \setminus \{data_{adv}, table_{adv}\}, \\ &\quad adv(a) \in \mathbb{Z} \\ &\quad adv(data_{adv}) = (\textsc{ro}, table_{adv}, table_{adv} + 1, table_{adv}) \\ &\quad adv(table_{adv}) = (\textsc{e}, b_m, e_m, b_m) \end{aligned}$$

(7) *the initial state of registers reg satisfies:*

$$reg(\mathrm{pc}) = (\textsc{rwx}, b, e, b), \quad reg(\mathrm{r}_0) = (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}), \quad reg(r) \in \mathbb{Z} \text{ otherwise}$$

(8) *the proof in the program logic that the initial configuration is safe given the invariants:*

$$\forall reg,$$

$$\boxed{\mathrm{mallocInv}(b_m, e_m)}, \boxed{\mathrm{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ (\textsc{rwx}, b, e, b); \begin{array}{l} \mathrm{r}_0 \Mapsto (\textsc{rwx}, b_{adv}, e_{adv}, b_{adv}) * \\ \mathop{\Large *}_{\substack{(r,v) \in reg, \\ r \notin \{\mathrm{pc}, \mathrm{r}_0\}}} r \Mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ \mathop{\Large *}_{\substack{(a,w) \in prog, \\ a \notin \{data, table, table+1\}}} a \mapsto w * \\ data \mapsto (\textsc{ro}, table, table + 2, table) * \\ table \mapsto (\textsc{e}, b_m, e_m, b_m) * \\ table + 1 \mapsto (\textsc{e}, b_a, e_a, b_a) * \\ \mathop{\Large *}_{\substack{(a,z) \in adv \\ a \notin \{data_{adv}, table_{adv}\}}} a \mapsto z * \lceil z \in \mathbb{Z} \rceil * \\ data_{adv} \mapsto (\textsc{ro}, table_{adv}, table_{adv} + 1, table_{adv}) * \\ table_{adv} \mapsto (\textsc{e}, b_m, e_m, b_m) \end{array} \right\} \rightsquigarrow \bullet$$

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(a_{flag}) = 0$.

```
; initially, PC = (RWX, code, end, code)
;            r1 = (unknown) pointer to adversary function
code:
  malloc 1                ; r1 = (RWX, b, b+1, b) where b is fresh
  mov r3 r1               ; r3 = (RWX, b, b+1, b)
  mov r4 r1               ; r4 = (RWX, b, b+1, b)
  store r3 1              ; b <- 1
  restrict r4 RO          ; r4 = (RO, b, b+1, b)
  call r1 [r3] [r4]       ; call macro that jumps to r1, keeps r3 as local
                          ; state and passes r4 as parameter
  load r1 r3              ; r1 = 1, as long as b was not changed during call
  mov r2 1
  assert r1 r2            ; assert (r1 = 1)
  halt
data:
  (RO, table, end, table) ; environment table
table:
  (E, bm, em, bm)         ; entry point to the malloc subroutine
  (E, ba, ea, ba)         ; entry point to the assert subroutine
end:
```

Fig. 13. Program passing a read-only capability to unknown callee

program *prog* is given access to both the `malloc` and `assert` routines. The adversary program *adv* is given access to `malloc`. We assume that *prog* contains the code and a table that has been filled by a linker with capabilities giving access to the two routines. Likewise, we assume that *adv* contains its program (arbitrary integers) and a table filled by the linker with the capability to the `malloc` routine. Similarly to the first adequacy theorem, the theorem states that if the capability machine starts with the capability pointing to *prog* in the program counter, and if it has been proved in the program logic that the machine can run until completion, then the assertion flag is *never* modified.

In what follows, Lemma 5 will thus allow us to prove end-to-end theorems saying that the assertion flag will still be unset after a full execution. This corresponds to the end-to-end theorems of Swasey et al. [2017] which are also phrased in terms of an assert primitive (albeit in a high-level language) that untrusted code does not get access to. Of course, such results remain a bit artificial: ultimately, in real systems, we are not directly interested in the contents of assertion flags in the system's memory, but rather in the system's interaction with the outside world: network communication, the content of displays etc. Our approach can be extended to reason about such properties, but we don't go into details here. Instead, we refer to Van Strydonck et al. [2021], where we have done exactly this extension, by adding MMIO and external event traces to our operational semantics and using Iris invariants and ghost state to reason about them. This results in end-to-end theorems that prove security properties about the external event traces of a system, which we regard as a more realistic end goal of a verification effort.

7.5 Application: read-only sharing of dynamically allocated memory

We now present an example program sharing a read-only capability with adversary code, showcasing the combined use of the `malloc` (Section 7.1) and `assert` (Section 7.2) routines, the secure calling convention (Section 7.3), and exercising our updated adequacy theorem (Section 7.4).

Figure 13 shows the implementation of our program of interest. The program dynamically allocates a region of size 1, into which it stores the integer 1. Next, it creates a copy of the newly created capability, which is then restricted to read-only (RO). This restricted capability is shared with an unknown callee, while the original copy is kept as local state. Upon return, an `assert` statement checks that the region indeed still contains 1. We then wish to prove that the final assertion always succeeds.

Notice that in this example, control is passed to untrusted code, corresponding to the first scenario in Figure 2a. However, we also allow the callee to return, i.e. jump to a callback. This is achieved using our calling convention to create a secure two-way boundary between known code and the unknown callee.

In order to prove that the `assert` statement succeeds, we rely on two facts. First, the heap-based calling convention guarantees the encapsulation of $(\text{RWX}, b, b+1, b)$. Second, sharing $(\text{RO}, b, b+1, b)$ with unknown code does not threaten the integrity of b, since RO capabilities cannot be used to write to memory. These two facts are key when proving the following specification:

Lemma 7 (Full execution specification (26)).

$$\boxed{\text{mallocInv}(b_m, e_m)}, \boxed{\text{assertInv}(b_a, e_a, a_{flag})}, \boxed{a_{flag} \mapsto 0}$$

$$\vdash \left\{ \begin{array}{ll} & \mathrm{r}_1 \Rrightarrow w_{adv} * \mathcal{V}(w_{adv}) * \\ & \ast_{(r,v) \in reg, r \notin \{\mathrm{pc}, \mathrm{r}_1\}} r \Rrightarrow w * \\ (\mathit{RWX}, \text{data}, \text{end}, \text{code}); & [\text{code}, \text{end}) \mapsto \texttt{code\_instrs} * \\ & \text{data} \mapsto (\mathit{RO}, \text{table}, \text{table}+2, \text{table}) * \\ & [\text{table}, \text{table}+2) \mapsto [(\mathit{E}, b_m, e_m, b_m); (\mathit{E}, b_a, e_a, b_a)] \end{array} \right\} \rightsquigarrow \bullet$$

Proof. We begin by applying program logic rules until we make it to the call to unknown code. At that point, a (fresh) region has been dynamically allocated and initialized to 1, and thus we have the following Separation Logic resources:

$$r_2 \Rrightarrow (\text{RWX}, b, b+1, b) * b \mapsto 1$$

At the call site, the calling convention creates an activation record, and sets up a sentry capability as the return pointer in r_0. (The "..." on the second line below stands for the address of the continuation after the call.)

$$\begin{array}{ll} & r_0 \Rrightarrow (\text{E}, act, act_{end}, act) * \qquad (11) \\ & [act, act_{end}) \mapsto \texttt{act\_instrs} +\!\!+ [(\text{RWX}, l, l+1, l); (\text{RWX}, \text{code}, \text{end}, \ldots)] * \\ & l \mapsto (\text{RWX}, b, b+1, b) * \\ & \mathrm{r}_2 \Rrightarrow 0 * \\ & \mathrm{r}_3 \Rrightarrow (\text{RO}, b, b+1, b) \qquad (12) \end{array}$$

Note in particular how the RWX capability pointing to b (part of the "local state") is only reachable from the capability (pointing to l) stored in the activation record, while the RO copy is available in register r_3.

The `call` macro then passes control to the adversary by jumping to w_{adv}. To reason about this jump, we apply Corollary 2 (assuming w_{adv} is safe). This requires us to show that all parameters in the current register state are valid. In particular, we must show that the sentry capability set up by the calling convention (11) is safe to execute, and that the read-only capability (12) is safe to share.

The latter is done by allocating an appropriate invariant, which is allowed to be *stronger* than the value relation itself, since the capability in question is read-only. To this end, we will allocate

an invariant that remembers the current integer pointed to by b, namely 1.

$$\boxed{\exists w, b \mapsto w * w = 1}$$

That same invariant is then used to prove that (11) is safe to execute, in particular to show that the assert statement succeeds, and hence does not change the assert flag. □

From this functional specification, we can instantiate our updated adequacy theorem (Theorem 5) to then derive the following end-to-end theorem about our program.

Theorem 6 (End-to-end theorem: the read-only permission guarantees integrity (27)). *Starting from an initial state of the machine* (reg, mem) *assuming that:*

- $prog \uplus adv \uplus malloc \uplus assert \subseteq mem$, *where:*
  $adv : [b_{adv}, e_{adv}) \to \text{Word}$, $prog : [\text{code}, \text{end}) \to \text{Word}$
  $malloc : [b_m, e_m) \to \text{Word}$ *and* $assert : [b_a, e_a) \to \text{Word}$;
- *the contents of prog correspond to the encoded instructions and program data (i.e. table with capabilities to the malloc and assert subroutines);*
- *the adversary memory contains no capabilities except a table with a capability to the malloc subroutine;*
- *malloc contains the implementation of the* `malloc` *subroutine;*
- *assert contains the implementation of the* `assert` *subroutine, with its flag at address* a_{flag}, *initialized to 0;*
- *the initial state of registers satisfies:*
  $reg(\text{pc}) = (\text{RX}, \text{code}, \text{end}, \text{code})$,
  $reg(\text{r}_1) = (\text{RWX}, b_{adv}, e_{adv}, b_{adv})$.

Then, for any reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$, *then* $mem'(a_{flag}) = 0$.

Proof. We apply the updated adequacy theorem (Theorem 5), using the specification proved in Lemma 7. All that remains is to prove the validity of the adversary capability: $\mathcal{V}(\text{RWX}, b_{adv}, e_{adv}, b_{adv})$. This is done in two steps. First, the adversary linking table is proved valid by applying validity of the `malloc` subroutine (Lemma 6). Next, the rest of the adversary region is proved valid through the assumption that it does not contain any other capabilities. The full proof can be found in the Coq mechanisation. □

8 CASE STUDY: A LIBRARY IMPLEMENTING DYNAMIC SEALING AND A CLIENT

We have presented so far a variety of smaller examples enforcing interesting encapsulation properties while interacting with adversarial code. In this section, we demonstrate that our approach scales up to the verification of a larger case study, involving not only the building blocks of Section 7, but using them to build and modularly verify a number of libraries built on top of each other.

We take inspiration from the literature on *object capability patterns* (OCPs) from high-level languages, a technique that enables programmers to protect the private state of their objects from corruption by untrusted code. More precisely, we consider the *dynamic sealing* OCP as presented by [Swasey et al. 2017]. Dynamic sealing enforces a form of data abstraction in the absence of static types. It can be implemented as a library providing pairs of seal/unseal functions, allowing their clients to "seal" private data into opaque objects which can be safely shared with untrusted code, and later unsealed in order to get back the original data.

In the context of a high-level language, [Swasey et al. 2017] present a formally verified implementation of dynamic sealing, equipped with a specification that captures the abstraction guarantees it provides. The authors then use this dynamic sealing library to build and verify a library of abstract integer intervals, where the integrity of an interval value (representing a range $[i, j)$ with

$$
\begin{array}{rl}
\texttt{interval}\,(28) = \lambda\_, & \texttt{let}\ (\texttt{seal}, \texttt{unseal}) = \texttt{makeseal}() \ \texttt{in} \\
& \texttt{let}\ \texttt{makeint} = \lambda\ z_1\ z_2, \ \ \texttt{let}\ x = \texttt{malloc}(2)\ \texttt{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad x \leftarrow \{min(z_1, z_2); max(z_1, z_2)\}; \\
& \qquad\qquad\qquad\qquad\qquad\quad \texttt{seal}(x) \\
& \texttt{in} \\
& \texttt{let}\ \texttt{imin} = \lambda\ i,\ \texttt{unseal}(i)[0]\ \texttt{in} \\
& \texttt{let}\ \texttt{imax} = \lambda\ i,\ \texttt{unseal}(i)[1]\ \texttt{in} \\
& (\texttt{makeint}, \texttt{imin}, \texttt{imax}) \\
\\
\texttt{client}\,(29) = & \texttt{let}\ (\texttt{makeint}, \texttt{imin}, \texttt{imax}) = \texttt{interval}()\ \texttt{in} \\
& \texttt{let}\ \texttt{checkint} = \lambda\ i, \texttt{assert}(\texttt{imin}(i) \leq \texttt{imax}(i))\ \texttt{in} \\
& (\texttt{checkint}, \texttt{makeint}, \texttt{imin}, \texttt{imax})
\end{array}
$$

Fig. 14. High-level pseudo-code for the implementation of the interval library and its client.

$i \leq j$) is protected using dynamic sealing. Finally, the authors use their verified integer library to establish *robust safety* of a simple client program checking integrity of intervals, establishing that an untrusted context cannot violate the internal invariants of the program and its underlying libraries.

We implement and verify low-level variants of the dynamic sealing OCP, interval library, and their robustly safe client. This represents a non-trivial amount of code: our implementation of those three components adds up to 632 machine instructions. Nevertheless, despite the fact that those libraries are implemented in low-level assembly code, we are able to give them specifications at a level of abstraction similar to their high-level counterparts.

For ease of reading, we will keep the explanations fairly high-level. We will first show high-level pseudo-code for the implementation of the interval library and its client, and informally discuss what kind of properties should be enforced. Then, we will present the key ideas for implementing dynamic sealing on a capability machine, and then for reasoning about it, in particular how to instantiate its specification to be able to verify the interval library.

8.1 Interval Library and Client

The interval library implements an abstract data type representing intervals. An interval has a lower and upper bound, which can be extracted via two functions; `imin` and `imax`. An interval is created via a function `makeint` that takes as input two integers, and chooses the smallest input as the lower bound, and the largest input as the upper bound. Crucially, the internal representation of an interval must stay hidden so as to guarantee its integrity.

We thus use *dynamic sealing* ([Sumii and Pierce 2004]) to dynamically enforce data abstraction for the intervals representation. We detail our implementation of seals in Section 8.2. For now, it suffices to know that a seal is a pair of functions, `seal` and `unseal`, where the former hides the internal representation of some value, such that only the latter can expose it.

An interval can be represented as an ordered pair of integers. On the capability machine, we implement such a pair as a dynamically allocated region of size two, storing the lower and upper bound of the interval. Then, an interval itself consists of a capability with read/write authority over the corresponding region of size two. In Figure 14, we depict the high-level implementation of our interval library. Note that the library implements closures around a fresh `seal`-`unseal` pair, used to seal the aforementioned internal representation of intervals. The low-level implementation that we

formally reason about can be thought of as the result of compiling the high-level implementation shown in Figure 14.

The same figure also depicts a client of the interval library. The client exposes four entry points to the environment: in addition to entries to `makeint`, `imin` and `imax` from a fresh instance of the interval library, the client also exposes an encapsulated `checkint` function that, given an interval, dynamically asserts that the expected representation invariant holds for the interval, that is, that the minimum of the interval is indeed smaller than or equal to the maximum of the interval.

When formally verifying the interval library and its client, we will need an invariant to keep track of each interval created by `makeint`. The invariant should capture the properties enforced by the implementation of the interval library. We can already list the internal properties of an interval intuitively. First and foremost, the lower bound of an interval must be less than or equal to its upper bound. A perhaps more subtle property is that intervals are immutable. Thus, we will need to define an invariant that represents each interval as a dynamically allocated region of size two, which stores the lower and upper bound, and is immutable. The `seal`-`unseal` pair encapsulated by the library will be used only to seal intervals that adhere to this representation (satisfy this invariant). Keeping this intuition in mind, let us now explore the technical implementation of seals.

8.2 Dynamic Sealing

Dynamic sealing makes it possible to support data abstraction dynamically. The function `makeseal` creates a pair of functions, `seal` and `unseal`, where `seal` is used to seal a word w into a fresh sealed word σ. We will also refer to σ as the key to w. The only way to extract the word w from σ is with `unseal`. The key point is that this `seal`-`unseal` pair supports data abstraction by *sealing away* or *hiding* the internal representation of some value, only known and available to the owner of the associated `unseal` function.

Although capability machines such as CHERI include seals as a language primitive, we show here how we can implement seals in software, as a low-level library. The library is implemented via a data structure that stores each word sealed through `seal`, associating each sealed word with a key. A key in itself does not reveal any details about the word it is hiding. However, it can provide access to that word, granted one has the proper authority to unseal it. Only a valid key should grant access to a sealed word. Keys, and the data structure that uses them, should intuitively satisfy two properties; (1) the unforgeable nature of keys and (2) the unique association between a key and the word it seals.

The `seal` and `unseal` subroutines respectively perform insertions and lookups in this underlying data structure. `seal` takes a word as input, generates a fresh key, and adds the key value association to the data structure. `unseal` takes a key as input, checks that the key is associated to a value in the data structure, in which case it returns the value.

8.2.1 Reasoning about dynamic sealing. A shared `seal`-`unseal` pair can be used to seal any word. In practice, one typically encapsulates a `seal`-`unseal` pair within a library, performing additional checks and thus ensuring that words that are sealed always satisfy a specific property. Then, whenever one successfully unseals a given key, one gets that the corresponding word satisfies the chosen property. For instance, the interval library enforces that each sealed word is a region of size 2, storing the ordered bounds of an interval.

When reasoning about code invoking the dynamic sealing library, one will need to pick, for each `seal`-`unseal` pair, an *representation invariant* $\Phi : \text{Word} \to \mathit{iProp}$ describing the values to be sealed/unsealed by the pair[7]. Then, each `seal`-`unseal` pair maintains an Iris invariant sealInv describing the state of the pair itself, namely the data structure storing the key-values for all sealed

[7] An analogous representation invariant is used in the [Swasey et al. 2017]

SEAL SPEC (30)

$$\left\{ \begin{array}{l} \\ (-, b_s, e_s, -); \\ \\ \end{array} \begin{array}{l} [b_s, e_s) \mapsto \texttt{seal} * \\ \text{sealInv } ds\ \Phi * \\ \mathtt{r}_1 \Mapsto v * \Phi(v) * \cdots \end{array} \right\} \rightsquigarrow \left\{ s\ k. \begin{array}{l} \lceil s = \text{Running} \rceil * \\ \quad \text{isSealedWord } k\ v * \\ \quad \mathtt{r}_1 \Mapsto\ k * \cdots \\ \vee \lceil s = \text{Failed} \rceil \end{array} \right\}$$

UNSEAL SPEC (31)

$$\left\{ \begin{array}{l} \\ (-, b_u, e_u, -); \\ \\ \end{array} \begin{array}{l} [b_u, e_u) \mapsto \texttt{unseal} * \\ \text{sealInv } ds\ \Phi * \\ \mathtt{r}_1 \Mapsto k * \cdots \end{array} \right\} \rightsquigarrow \left\{ s\ v. \begin{array}{l} \lceil s = \text{Running} \rceil * \\ \quad \text{isSealedWord } k\ v * \\ \quad \mathtt{r}_1 \Mapsto\ v * \Phi(v) * \cdots \\ \vee \lceil s = \text{Failed} \rceil \end{array} \right\}$$

Fig. 15. Specifications of `seal` and `unseal`

entries. Additionally, this invariant stores the information that each sealed value satisfies Φ.

$$\text{sealInv } ds\ \Phi \text{ (32)} \triangleq \boxed{\begin{array}{l} \exists wvals, \text{dataStructure } ds\ wvals \\ * \mathbin{\ast\!\!\ast}_{(-,w) \in wvals}\ \Phi(w) \end{array}}$$

We require that Φ is persistent, since the representation invariant of a sealed word should always hold once sealed. The dataStructure predicate describes the state of the data structure internal to the seal library (see Section 8.2.2 for a formal definition). It asserts that *ds* can be used to access a data structure storing the key value pairs denoted by *wvals* (a sequence of pairs in Addr × Word). In other words, *wvals* is the complete list of all words that have been sealed so far, each paired with their associated key.

A sealed word is sealed forever. It is thus possible to persistently remember that a particular word is an element of *wvals*. The predicate isSealedWord $k\ v$ states that the key k is uniquely associated with the sealed word v. We present the formal definition of isSealedWord in Section 8.2.2.

The functional specifications of the `seal` and `unseal` subroutines depend on an instance of the seal invariant sealInv, for a specific user-provided predicate Φ. Then, `seal` can only be applied to words for which the representation predicate Φ holds. `unseal` can fail if a given key is not valid, or if it is not associated with any sealed word, however if it succeeds, it will return a word for which Φ holds. The specification of `makeseal` allocates a fresh sealInv instance, for any Φ chosen by the client of the library. Figure 15 shows specifications for `seal` and `unseal` (where we omits low-level administrative details).

8.2.2 Implementing a low level seal library. We now present the data structure used to implement the low-level seal library. We implement it as a linked associative list with a twist, next refered to as a *linked list dictionary*. The trick is to take advantage of the unforgeable nature of capabilities, and use the capability to (a subrange of) a list node as a key to that node; the corresponding value being then stored in the node.

Figure 16 shows the in-memory representation of a linked list dictionary storing three key-value pairs. Each node is implemented as a region of size three, where the bottom address acts as the key address. To avoid access to sealed values, it is important that a key does not provide authority over the other parts of a node (the value and the next pointer). For instance, the value v_1 is uniquely associated to the capability $(\text{RWX}, b_1, b_1 + 1, -)$.

The linked list dictionary library contains two subroutines, `findB` (33) and `append` (34). `findB` expects as input an integer b, searches the linked list for a node of the form $(\text{RWX}, b, b + 3, -)$ and returns the value that the associated node stores. It fails if no such node exists. `append` expects a

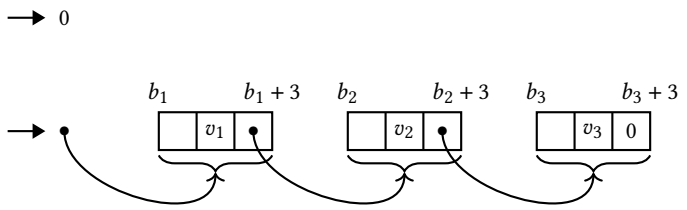


Fig. 16. In-memory representation of an empty dictionary linked list and a dictionary linked list with three values v_1, v_2 and v_3.

word as input, invokes `malloc` to dynamically allocate a new node of size three, stores the input word in the second position of that node, and then stores that node as the new tail of the linked list. Finally a key can then be derived from the newly created node; we now explain in more detail how that is done.

A fresh instance of a `seal`-`unseal` pair is created by calling the `makeseal` subroutine, which returns a pair of closures encapsulating a new empty linked list dictionary. Sealing a word w adds it to the dictionary, and returns a *restricted* capability representing the key to the linked list dictionary entry. Say for instance that the input word w is appended to the list in a fresh node $(\text{RWX}, b, b+3, b)$. The `seal` subroutine will then return the key $(\text{RWX}, b, b+1, -)$ (the address pointed to does not matter, and is here omitted for clarity).

Recall that in the enclosed linked list dictionary, w will be stored at address $b+1$, for which the returned sealed value, or key, does not have authority. This sealed value is *unforgeable*. The only way to create it would be to derive it from a capability $(\text{RWX}, b', e', \_)$ where $[b, b+1) \subseteq [b', e')$. However, this is impossible since the appended node is freshly allocated using a safe `malloc` subroutine, which is guaranteed to hand out fresh regions upon invocation. Only `seal` has access to such a capability, and thus sealed values cannot be forged.

In turn, the `unseal` subroutine expects a RWX capability of range 1 as input. It reads its lower bound, searches the enclosed linked list for a node with matching lower bound, and returns the associated word. Let us consider a continuation of the previous example. Say that `unseal` receives $(\text{RWX}, b, b+1, -)$ as input. It begins by authenticating the key by dynamically verifying its permission to be RWX, and its size to be 1. Upon validating its permission and range, it then runs `findB` on the enclosed linked list dictionary with the integer b, and returns the word stored within the node $(\text{RWX}, b, b+3, -)$ at address $b+1$, namely the previously sealed word w. The authentication guarantees that a key has the same unforgeable authority as when it was created.

In summary, the `seal` and `unseal` subroutines are implemented as follows:

- `seal`:
  (1) append the input to the enclosed linked list dictionary
  (2) restrict the range of the fresh node capability to bottom address of node
  (3) return resulting restricted capability
- `unseal`:
  (1) check that permission of input is RWX
  (2) check that the range of input is 1
  (3) get the lower bound of input
  (4) find the node in the linked list dictionary with same lower bound
  (5) return the stored word at that node (fail if no such node exists)

We now have enough ingredients to revisit the predicates used in the previous section to define the seal invariant. Recall that the dataStructure predicate represents the state of the data structure internal to the seal library (now defined to be a linked list dictionary), and that the isSealedWord predicate describes a persistently known association between a sealed word and its key.

$$\begin{aligned} \text{dataStructure } ds\ wvals &\triangleq \exists hd,\ ds \mapsto hd \\ &\quad * \text{ isList } hd\ wvals \\ &\quad * \text{ Exact } wvals \\ \text{isSealedWord } k\ v &\triangleq \exists wvals, \text{ Pref } wvals * \lceil (k, v) \in wvals \rceil\ ^{8} * \mathcal{V}(\text{RWX}, k, k+1, -) \end{aligned}$$

The head of the linked list dictionary is stored in location *ds*. isList corresponds to a standard inductive separation logic predicate for linked lists. Since the list monotonically grows, it is useful to persistently remember any prefix of the linked list dictionary. Exact *wvals* (the authoritative view of the list state) roughly states that *wvals* is the full state of the data structure. Pref *wvals* (the local fragment view) states that *wvals* is a prefix of the data structure. isSealedWord k v, a persistent predicate, states that the word v has been sealed with a key; a capability with with lower bound k. This key is safe to share, hence $\mathcal{V}(\text{RWX}, k, k+1, -)$ holds.

In the next section, we describe how we use the reasoning principles about `seal-unseal` to verify our interval library.

8.3 Verifying the Interval Library and its Client

The first key step is to formally define the representation invariant for an interval. Recall the intuitive description given in Section 8.1: an interval is a capability with authority over a region of size 2, storing the lower and upper bounds of an interval, and which is immutable.

A first thought might be that one can define the representation invariant using two points-to predicates for the region. However, this does not capture the immutability of intervals, nor is it persistent. Instead, we use *persistent* points-to predicates ([Vindum and Birkedal 2021]). A persistent points-to predicate $a \hookrightarrow w$ asserts that address a stores the word w. It can be used to read from address a, but not write to it, and as such, is a persistent resource. This is exactly what we need for our immutable invariants. We formally define the representation invariant isInterval as follows:

$$\begin{aligned} \text{isIntervalInt } z_1\ z_2\ w\ (35) &\triangleq \exists a, \lceil w = (\text{RWX}, a, a+2, a) \rceil * a \hookrightarrow z_1 * (a+1) \hookrightarrow z_2 * \lceil z_1 \leq z_2 \rceil \\ \text{isInterval } (36) &\triangleq \lambda w, \exists z_1\ z_2, \text{isIntervalInt } z_1\ z_2\ w \end{aligned}$$

(Note, in particular, that the invariant also captures the property that the lower bound is less than or equal to the upper bound.) Using properties of persistent points-to predicates, we can prove the following lemma:

Lemma 8 (37). $\text{isIntervalInt } z_1\ z_2\ w \rightarrow \text{isIntervalInt } z_3\ z_4\ w \rightarrow \lceil z_1 = z_3 \wedge z_2 = z_4 \rceil$.

Because isInterval is persistent, we can use it as the representation predicate for a `seal-unseal` pair, which will thus operate over the following invariant:

$$\text{sealInv } ll \text{ isInterval}$$

This seal invariant is allocated by the specification for `makeseal`, which is invoked during the creation of an interval library closure.

When sealing a new interval using `makeint`, we must establish isInterval for the newly created interval. This requires us to transform the regular points-to predicates handed out by the `malloc` specification into persistent points-to predicates, and assert that indeed $min(z_1, z_2) \leq max(z_1, z_2)$.

[8] In the Coq mechanization, *wvals* associates the word w to $k+1$ rather than k, for technical reasons. This small discrepancy has otherwise no impact on the rest of the proof.

Specifications for `imin` and `imax` return the respective lower and upper bound of a sealed interval. The seal invariant guarantees that the sealed word is an interval according to the representation invariant isInterval. In other words, if `imin` or `imax` succeeds for some word w, we know that w is the key to some associated capability pointing to the bounds of an interval $[l, r]$; specifically that isIntervalInt l r w holds.

During the verification of `checkint`, the specification for `imin` gives us some value l and predicate isIntervalInt l r w. Similarly, the specification for `imax` gives us some value r' and predicate isIntervalInt l' r' w. Notice that the bounds may be different, but the sealed word w is the same in each instance. We can thus apply Lemma 8 on the two given instances of isIntervalInt, and use the definition of isInterval to conclude that the given `assert` statement succeeds, namely that $l \leq r$.

Finally, all that remains is to apply adequacy and prove the following final end-to-end theorem:

Theorem 7 (End-to-end theorem: the interval client does not trigger an assertion failure (38)). *Starting from an initial state of the machine* (reg, mem) *in which regions reserved for the interval library, the seal library,* `malloc`*, the assert flag, the client and the adversary are all disjoint, and initialized as expected, we have that, for any* reg', mem', *if* $(reg, mem) \longrightarrow^* (reg', mem')$ *then* $mem'(a_{flag}) = 0$.

9 DISCUSSION AND PERSPECTIVES

In this paper we have introduced Cerise, a program logic for reasoning about a low-level capability machine. Moreoever, we have shown how Cerise can be used to define a logical relation for reasoning about unknown code. Thanks to the logical relation and the fundamental theorem from Section 5, Cerise can be used for *robust verification* [Sammler et al. 2020; Swasey et al. 2017], i.e., to verify correctness of software that interacts with unverified components. The Cerise program logic is the culmination of ideas used in a sequence of earlier papers [Georges et al. 2021; Skorstengaard et al. 2018, 2019a; Van Strydonck et al. 2021] and this paper is intended to give an accessible and didactic introduction to Cerise and the application of Cerise to program verification in the presence of untrusted code, accompanied with new results on a heap-based calling convention and implementations of sophisticated object-capability patterns.

Throughout the paper we have introduced increasingly complex examples, which demonstrate how fine-grained abstractions can be implemented on a capability machine and reasoned about using Cerise. Our examples from Section 7 and Section 8 are modeled after examples from a paper about a high-level object capability language [Swasey et al. 2017]. Because of the more low-level nature of our capability machine, we had to implement some abstractions ourselves (such as the calling convention in Section 7.3) but we think it is otherwise fair to say that our examples faithfully represent the examples used by Swasey et al., using the same granularity of encapsulation and attacker interaction. As such, this paper demonstrates that the low-level security primitives offered by our capability machine are expressive enough to implement high-level language abstractions, despite the stronger attacker model of a low-level adversary. At the same time, the examples show that Cerise is expressive enough to reason about these abstractions.

Cerise is the first instantiation of the Iris framework to such a low-level language and thus this work also demonstrates that the key features of Iris (such as guarded recursion, ghost state, and invariants) are equally applicable in this low-level setting as in the high-level settings they were originally intended for.

Of course, while we implement and reason about our examples directly in the capability machine assembly language, we are not proposing that real software should all be developed in that way. On the contrary, we think this is only realistic for low-level code in compiler back-ends [Georges et al. 2021; Skorstengaard et al. 2019a], operating systems and low-level security measures [Van Strydonck

et al. 2021]. Other software should be developed and reasoned about in a more abstract setting, which suggests the need for a secure compiler that preserves high-level security guarantees in a low-level environment. In the context of capability machines, such compilers have been investigated already, both formally [El-Korashy et al. 2020; Van Strydonck et al. 2019], and practically [Chisnall et al. 2017; Richardson 2020]. While we in this work have shown how to implement and reason about some high-level programming patterns at a low level, much interesting work remains to be done to further explore the design of a high-level language whose security abstractions map well to those offered by a capability machine.

An important aspect of the universal contract provided by our logical relation and fundamental theorem is that it formalizes the security guarantee of our capability machine without overspecifying implementations of the ISA. The contract specifies an authority bound that suffices to reason about adversarial code, but does not overly constrain future extensions or optimized implementations of the ISA. This is similar to how the ISA itself is designed to specify expected behavior that is sufficient for software authors to reason about their code without preventing CPU designers from constructing optimized or extended implementations. In fact, we believe universal contracts offer a general and powerful approach for formalizing ISA security guarantees. Such security guarantees are informally stated in informal ISA specifications but they have not yet been incorporated in formal definitions of ISAs [Armstrong et al. 2019; Bourgeat et al. 2021]. As such, a promising application of universal contracts like the one from Section 5 is to incorporate them into the ISA definition to formalize intended ISA security guarantees.

Finally, it is worth acknowledging that in this paper, we only describe a minimal capability machine that lacks many features from realistic capability machine ISAs. Our approach has been extended to support some additional features in the literature (e.g., local and uninitialized capabilities [Georges et al. 2021], and MMIO [Van Strydonck et al. 2021]), but other features are still missing for now (e.g. sealing, interrupts, virtual memory, etc.). In terms of reasoning, the unary model we have described only supports reasoning about integrity properties. However, we have implemented a binary model in our Coq development which can be used to reason about relational properties (e.g., confidentiality).

10 RELATED WORK

We now discuss several lines of work related to ours. First, we discuss earlier variants of Cerise by the authors and colleagues. Then, we discuss work on verifying object capability patterns in *high-level* languages, verification of ISA properties *in CHERI*, and other applications of *universal contracts* in the literature.

10.1 Earlier variants of Cerise

Earlier variants of Cerise focused on showing how capabilities can be used to implement a *secure, stack-based calling convention* [Georges et al. 2021; Skorstengaard et al. 2019a,b] and *nested security wrappers* [Van Strydonck et al. 2021].

[Skorstengaard et al. 2019a] were the first to show that capabilities can be used to implement a secure stack-based calling convention, i.e., a calling convention where the security guarantees of function calls at the machine code level are faithful to the high-level notion of a function call. They employed an additional kind of "local" capabilities and stack clearing to achieve security. Their work follows a similar methodology as the one described here, that is, they define a logical relation which characterizes a notion of safety. However, their proofs were not mechanized and the logical relation was defined using a non-trivial concrete model; in contrast we use the Cerise program logic to define and prove properties about our logical relation, which means that our development is done at a higher-level of abstraction and thus we, e.g., do not have to solve any recursive domain equations. In

follow-up non-mechanized work, [Skorstengaard et al. 2019b] achieved similar security guarantees with a novel calling convention based on so-called "linear" capabilities; capabilities that can never be duplicated. Although this calling convention avoids the stack clearing required in the previous work, linear capabilities come with certain architectural restrictions [see e.g. Skorstengaard et al. 2019b, §6.2]. An efficient implementation of linear capabilities has so far not been demonstrated.

The subsequent work by [Georges et al. 2021] introduced a new type of capabilities (called "uninitialized") to avoid most of the stack clearing from Skorstengaard et al.'s first calling convention, thereby improving runtime efficiency. Importantly, uninitialized capabilities do not come with the same architectural hurdles as linear capabilities. As a second contribution, Georges et al. used Iris to formulate safety as a logical relation and mechanized their proofs of security.

The aforementioned logical relations of both Skorstengaard et al. and Georges et al. are more expressive and therefore significantly more complicated than the one presented here: they permit reasoning about revocation of local/linear/uninitialized capabilities and well-bracketedness properties of machine-code "function calls", on top of local-state encapsulation. In our present work, object capabilities ensure local state encapsulation, but we do not enforce calls and returns to be well-bracketed. In particular, we do not prevent an adversary from invoking a return pointer several times, or storing return pointers for later use. In other words, our calling convention implements the kind of function calls one has in a high-level language with control operators (e.g., call/cc), where calls and returns are not necessarily well-bracketed. (It is well-known that models of well-bracketed function calls are more involved than models of not-necessarily-well-bracketed function calls, see, e.g., [Abramsky et al. 1998; Dreyer et al. 2012], and here we opted for the latter, to present a more accessible model, which suffices for a heap-based calling convention and for studying low-level implementations of object-capability patterns.)

In a different line of work, Van Strydonck et al. [2021] employed a capability machine and logical relations model similar to the one presented here, but with additional support for MMIO, to verify safety properties for small, nestable wrappers around security-critical devices on a capability architecture. As part of the verification effort, multiple end-to-end security theorems were proven, which state that safety predicates of interest hold over the trace of IO events admitted by the machine. Here we have instead focused on demonstrating how a core model (without MMIO support) can be used to reason about low-level implementations of object-capability patterns.

10.2 Verifying object capability patterns in high-level languages

A number of high-level programming languages allow for programming patterns similar to object capabilities, that enable preserving local state while interacting with unknown code. Examples are closures, and high-level objects in capability safe languages.

[Devriese et al. 2016] pioneered the use of a logical relation to give a semantic characterization of capability safety (earlier work used a more conservative syntactic approach based on whether or not objects contain references to each other and ignored the behaviour of objects). [Devriese et al. 2016] focused on capability safety for a core calculus of Javascript, including a notion of observable effects, and used an explicit construction of their logical relation (not a program logic), which was the inspiration for the capability model by [Skorstengaard et al. 2019a] mentioned above and for the work by [Swasey et al. 2017], who presented a program logic which allows reasoning modularly about object capability patterns in a high-level language. The methodology of [Swasey et al. 2017] is close to the one presented here, but in contrast to [Swasey et al. 2017] we reason about object capabilities on a low-level machine. For instance, Swasey et al. define two predicates to describe a reference: a predicate for "high integrity" locations ($\ell \hookrightarrow v$), and one for "low integrity" locations (lowloc ℓ). The first predicate grants exclusive access to the corresponding reference, and is therefore not safely shareable with an adversary. The second is shareable with an adversary, but

can only be used to read and write "low integrity" values. In our setting, "high integrity" directly corresponds to the predicate $a \mapsto w$ for a memory location, and "low integrity" corresponds to the invariant used in the definition of $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Correspondingly, our definitions satisfy similar reasoning rules to the ones established by Swasey et al.. In particular, we believe that the various object capability patterns they verify can be implemented and verified in a similar way in the setting of a capability machine, using the principles presented in this paper. We demonstrated one such implementation by adapting their dynamic sealing example in Section 8. Additionally, the robust safety theorem of [Swasey et al. 2017] is related to our template adequacy theorem with malloc and assert (Theorem 5); our assert flag plays a role similar to the OK flag in [Swasey et al. 2017].

10.3 Verifying ISA properties in CHERI

[Nienhuis et al. 2020] formally verify a number of "architectural" properties of CHERI capability machines. This constitutes a significant mechanization effort: the authors tackle the full generality of a realistic operational semantics for CHERI, which is significantly more complex than the minimal machine we consider here. The approach followed by Nienhuis et al. is different from ours: they state the properties they establish as trace properties, over a trace of "abstract actions" describing the various capabilities transiting through the machine during the execution. This approach makes it possible to state the desired properties in a very explicit and concrete fashion. For instance, the authors state and prove a property of "capability monotonicity": during the execution, the authority of available capabilities cannot increase (in other words, the machine does not allow forging new authority). Intuitively, this seems like a very reasonable property, required for proper operation of the capability machine. However, in practice it is more subtle: calls between components (in our case, jumping to an E-capability) do allow for some restricted form of non-monotonicity. The property proved by Nienhuis et al. is thus restricted to trace fragments that do not include calls to a different component. Our methodology is less explicit, but more expressive. In our setting, the fundamental theorem can be understood as expressing that "the machine works well". Its very extensional statement is admittedly harder to understand in terms of the operational semantics of the machine, but it enables deriving correctness statements in terms of the operational semantics that do apply to a full execution of the machine, including calls between an arbitrary number of components.

10.4 Other applications of universal contracts

As mentioned, our fundamental theorem constitutes a universal contract for arbitrary code, i.e., it allows deriving the guarantee that *any* adversarial capability is safe to execute, given validity of said capability. This safety is typically obtained by syntactically restricting the adversarial capability; e.g., requiring that the adressed memory only contains integers.[9] Similar notions of universal contracts have been used for high-level languages (explicitly or implicitly) in the literature. The aforementioned work of Skorstengaard et al. [2019a,b], and Swasey et al. [2017] all used a version of universal contracts, and placed varying syntactic restrictions on adversaries. The semantic type systems of Jung et al. [2017] and Sammler et al. [2020] permit similar reasoning about untrusted code based on a syntactic well-typedness restriction. The back-translation in the full-abstraction proof by Van Strydonck et al. [2019] involved an explicit, universal separation logic contract for a C-like language with capabilities. Generally, whenever a semantic model is used to describe semantic guarantees satisfied by arbitrary code (possibly subject to syntactic restrictions), and

[9]Note that instructions are encoded in memory as integers.

when these guarantees are used in the manual verification of other code, this can be regarded as an application of a universal contract.

Acknowledgements. Thanks to Léon Gondelman and Pierre Pradic for feedback on earlier drafts of this document.

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation; by the Research Foundation - Flanders (FWO); and by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU). Thomas Van Strydonck holds a Research Fellowship of the Research Foundation - Flanders (FWO). Amin Timany was postdoctoral fellow of the Flemish Research Foundation (FWO) during parts of this project.

REFERENCES

Samson Abramsky, Kohei Honda, and Guy McCusker. 1998. A Fully Abstract Game Semantics for General References. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*. IEEE Computer Society, 334–344. https://doi.org/10.1109/LICS.1998.705669

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 71:1–71:31. https://doi.org/10.1145/3290384

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. 2021. A Multipurpose Formal RISC-V Specification. *arXiv:2104.00762 [cs]* (April 2021). arXiv:2104.00762 [cs] http://arxiv.org/abs/2104.00762

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 319–327. https://doi.org/10.1145/195473.195579

David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 569–583. https://doi.org/10.1145/3037697.3037725

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities Using Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/EuroSP.2016.22

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2020. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. (May 2020). arXiv:2005.05944

Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*. IEEE.

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 66:1–66:34 pages. https://doi.org/10.1145/3158154

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

Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press. https://homes.cs.washington.edu/~levy/capabook/

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. 2020. Rigorous

engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*.

Alexander Richardson. 2020. *Complete Spatial Safety for C and C++ Using CHERI Capabilities*. Ph.D. Dissertation. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-949.html

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer International Publishing, 475–501.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems* 42, 1 (Dec. 2019), 5:1–5:53. https://doi.org/10.1145/3363519

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019b. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 19:1–19:28. https://doi.org/10.1145/3290332

Eijiro Sumii and Benjamin C. Pierce. 2004. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 161–172. https://doi.org/10.1145/964001.964015

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM. https://people.mpi-sws.org/~swasey/papers/ocpl/ocpl-20170418.pdf

Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2021. Proving Full-System Security Properties under Multiple Attacker Models on Capability Machines. (2021).

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* ICFP (2019).

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 76–90. https://doi.org/10.1145/3437992.3439930

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. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-951

Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Computers* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358288