Efficient and Provable Local Capability Revocation using Uninitialized Capabilities

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
SANDER HUYGHEBAERT, Vrije Universiteit Brussel, Belgium
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

Capability machines are a special form of CPUs that offer fine-grained privilege separation using a form of authority-carrying values known as capabilities. The CHERI capability machine offers local capabilities, which could be used as a cheap but restricted form of capability revocation. Unfortunately, local capability revocation is unrealistic in practice because large amounts of stack memory need to be cleared as a security precaution.

In this paper, we address this shortcoming by introducing *uninitialized capabilities*: a new form of capabilities that represent read/write authority to a block of memory without exposing the memory's initial contents. We provide a mechanically verified program logic for reasoning about programs on a capability machine with the new feature and we formalize and prove capability safety in the form of a universal contract for untrusted code. We use uninitialized capabilities for making a previously-proposed secure calling convention efficient and prove its security using the program logic. Finally, we report on a proof-of-concept implementation of uninitialized capabilities on the CHERI capability machine.

CCS Concepts: • **Security and privacy → Logic and verification**; **Formal security models**; • **Theory of computation → Program verification**; **Program specifications**.

Additional Key Words and Phrases: capability machines, local capabilities, uninitialized capabilities, capability safety, universal contracts, program logic, capability revocation, CHERI

ACM Reference Format:
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, Article 6 (January 2021), 30 pages. https://doi.org/10.1145/3434287

1 INTRODUCTION

Capability machines are a type of CPUs with support for fine-grained privilege separation, dating back to the 1960s [Dennis and Van Horn 1966; Levy 1984; Watson et al. 2019]. In this paper, we

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

2475-1421/2021/1-ART6
https://doi.org/10.1145/3434287

will specifically focus on a recent family of capability machines called CHERI [Watson et al. 2019]. Capability machines provide native support for capabilities: values which represent a certain authority to interact with memory, the operating system or other isolated components in the system. Capabilities come in several forms. Memory capabilities represent the authority to access a certain region of memory with a certain permission (e.g. RW or RX). On many capability machines, including CHERI, memory capabilities are designed to directly replace pointers, thus adding native bounds and permission checks with almost zero runtime overhead.

Additionally, capability machines usually offer a form of object capabilities [Miller 2006]: a form of reified closures that represent the authority to invoke an isolated component without exposing its internal state and its private capabilities. Invoking such an object capability passes control to the other component and makes available its private capabilities and thus, its authority. As such, they offer a cheap form of context switches. On CHERI, object capabilities take the form of pairs of code and data capabilities, tied together by being sealed with a common seal [Watson et al. 2016, 2015]. Sealing is a primitive CHERI operation that renders capabilities opaque and unusable, except that the pair can be invoked with a special instruction `CCall`.

Local capabilities are a new feature of CHERI [Watson et al. 2015]. Conceptually, they are intended as a form of ephemeral capabilities that can be used directly but not stored for later use. More technically, they are a form of capabilities that can be kept in registers but not stored in memory. There is, however, an exception to the latter rule: local capabilities can be stored in memory through memory capabilities with special "write-local" permission. This exception is specifically intended for the stack capability, so that the stack can be used for spilling local capabilities from registers and function arguments.

In principle, local capabilities make it possible to pass a capability to an untrusted component temporarily, without allowing the component to store it for later use. In other words, if the component is invoked again, the local capability is effectively revoked: the component cannot have access to it anymore. As such, local capabilities can be seen as a restricted revocation primitive with little performance overhead.

Unfortunately, this potential is not realized in practice. While CheriBSD (an adaptation of FreeBSD which makes use of CHERI capabilities) does use local capabilities to represent stack pointers, they work with private per-compartment stacks, and local capabilities are never passed to untrusted code in other compartments [Watson et al. 2015]. Hence, the CheriBSD system does not actually rely on local capabilities for enforcing security properties but only to mitigate the impact of potential bugs; specifically, to prevent accidental leaks of stack pointers. The latest CHERI ISA reference document mentions two additional dimensions of locality (kernel vs. user-space memory, garbage-collected vs manually managed memory), but neither involves a form of revocation [Watson et al. 2019, §D.13].

The likely reason for this limited use of local capabilities as a revocation mechanism is that its guarantees only hold under an important restriction. If we want to revoke a local capability before a second invocation of untrusted adversarial code, we must make sure not to accidentally leak an old copy of the capability. While local capability rules ensure that such old copies can never end up in heap memory (because no write-local capabilities to heap memory exist), they may still be present in any location where the adversary may have previously stored them: capability registers, but also any region of memory which it had a write-local memory capability for. Practically, the only way accidental leaking can be avoided is by clearing unused registers and sweeping over this write-local memory to clear it entirely or at least erase local capabilities. For example, in a secure calling convention built on local capabilities, Skorstengaard et al. [2018] have to clear the entire unused part of the stack before any invocation of adversarial code. This requirement is very costly in practice, and also hard to avoid, since the stack must be made write-local if we want to allow invoked code to spill registers or store local capabilities away during sub-invocations. The

performance impact might be mitigated with special hardware support [Joannou et al. 2017], but it is unclear whether this is enough to make it realistic for practical use.

In this paper, we propose a way to redeem local capabilities as a restricted but efficient revocation primitive using *uninitialized capabilities*. This is a new form of capabilities that represents read-write access to a region of memory without access to its current contents. Regions of memory which the adversary has previously had write-local access to, specifically the stack, can be made available to the adversary through an uninitialized capability without the need to clear the memory beforehand. Technically, an uninitialized capability's range of authority is divided into two parts: the range *below* the address currently pointed to, say $[b, a)$, and the range *above* the current address, say $[a, e)$. The range below represents the initialized part of the capability, and the range above represents its uninitialized part. The capability grants read-write access to $[b, a)$, and write-only access to $[a, e)$. However, if the address a is written to, the boundary between the two parts is automatically changed to include the now-overwritten memory location, i.e., a is automatically incremented (pushing a value on the stack in the case of a stack capability). An uninitialized capability can be restricted by lowering the current address and thus "uninitializing" a range of memory (popping the stack), but its authority can only be increased by writing to it, thus overwriting its previous content. Additionally, regular capabilities can be made uninitialized and an uninitialized capability to $[b, e)$ can be restricted to a regular read-write capability to its initialized part $[b, a)$ which can be passed to existing code.

Although uninitialized capabilities are more generally useful, this paper focuses on how they redeem local capabilities as a revocation primitive. To this end, we formally establish the guarantees provided by local and uninitialized capabilities with a capability safety result based on the one by Skorstengaard et al. [2018]. Capability safety is expressed as a universal contract—or specification—that holds for arbitrary assembly code. The universal contract is defined using a logical relation which captures the authority represented by a capability, and guarantees that this authority is respected and monotonically preserved by arbitrary assembly code. To simplify the definition of the logical relation and avoid some tedious book-keeping related to step-indexing and shared logical state, we make use of a program logic for our capability machine model which we define using the Iris program logic framework [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a]. We have mechanized all of the technical development using the Iris implementation in Coq [Krebbers et al. 2018, 2017b].

Our program logic and logical relation are the most important technical contributions of this work. To allow reasoning about the pattern of local capability revocation, we use a novel combination of Iris' invariants and saved predicates with more traditional Kripke world-indexing. We use this Kripke world-indexing with public/private transitions [Dreyer et al. 2010; Skorstengaard et al. 2018] and a new idea of what we call frozen regions to support typical patterns of (temporary) local capability revocation.

To demonstrate both how uninitialized capabilities redeem local capabilities as a revocation primitive in practice and how our capability-safety result enables reasoning about programs using these features, we study a modification of Skorstengaard et al. [2018]'s calling convention that avoids the problematic clearing of large parts of the stack. The resulting calling convention is another contribution in its own right. We demonstrate how our program logic can be used to prove correctness of programs using the calling convention, specifically for the classic "awkward" example which relies on well-bracketed control flow and stack frame encapsulation. The mechanization is highly called for because of the low-level nature of capability machines, and the large amount of bookkeeping that is necessary for reasoning about example programs (arithmetic manipulation of addresses, restriction of all relevant capabilities, setup of activation records, etc.).

Finally, more practically, we provide evidence that uninitialized capabilities can be realistically added to the CHERI capability machine by implementing them in the CHERI-MIPS ISA and the definition of its operational semantics in SAIL [Armstrong et al. 2019]. Additionally, we add support for the new instructions to the Clang/LLVM assembler. The simulator that we thus obtain from SAIL and the modified assembler have been used to experiment with the new calling convention in manually modified assembly programs.

To summarize, our contributions are centered around the new uninitialized capabilities:

- We propose uninitialized capabilities: a new form of capabilities that represents read-write access to memory without exposing the memory's initial contents (Section 4).
- We explain how uninitialized capabilities redeem CHERI's local capabilities as a restricted but efficient revocation primitive (Section 4).
- We characterize the combined guarantees of the two features with a capability-safety result, mechanized in Coq, as a universal contract that holds for arbitrary assembly programs. It uses a logical relation and a novel combination of Iris features like guarded recursion and shared invariants, with Kripke world-indexing and public/private transitions for reasoning about local capability revocation (Sections 5 and 6).
- We define a modified version of the calling convention of Skorstengaard et al. [2018] which removes its performance problems. We provide evidence that it enforces well-bracketed control flow and local stack frame encapsulation by proving an implementation of the awkward example correct (Section 6.8).
- We implement uninitialized capabilities in the SAIL semantics of CHERI-MIPS and the Clang/LLVM assembler and use them to experiment with the modified calling convention (Section 7).

Finally we add that, to the best of our knowledge, our Iris-Coq mechanization of capability safety is the first mechanically verified account of key deep semantic properties (spanning several components, including unknown adversarial code) that are enforceable using capabilities. The Iris-Coq mechanization can be found at https://github.com/logsem/cerise-stack/releases/tag/POPL2021.

The idea and implementation of uninitialized capabilities has also been reported in the master thesis of one of the authors [Huyghebaert 2020], overlapping partly with Sections 4 and 7.

2 A CAPABILITY MACHINE WITH LOCAL CAPABILITIES

This section defines the operational semantics of our capability machine. Our machine model is defined along the same lines as the one from Skorstengaard et al. [2018], and hence transitively draws from CHERI [Watson et al. 2015] and the M-Machine [Carter et al. 1994]. In Section 2.1 we describe the operational semantics for a bare-bones capability machine (without local and uninitialized capabilities) as a starting point. Then, we add support for local capabilities in Section 2.2. The semantics for uninitialized capabilities will be treated later in Section 4, resulting in the full definition of the capability machine semantics we assume in the rest of the paper.

Figures 1 to 4 summarize the operational behavior of our capability machine, and will be referenced on multiple occasions. They are color-coded as follows: the bare-bones capability machine is defined in black; additions related to local capabilities are typeset in red. Finally blue additions, introduced on top of the red ones, account for uninitialized capabilities and will be discussed in Section 4.

2.1 Bare-bones Capability Machine

Figure 1 defines the syntax we use in our capability machine. The set of addresses Addr is finite, to make our model more realistic, and described by the integer range $[0, \mathsf{AddrMax}]$. The address AddrMax is the top address and cannot be dereferenced.

$$
\begin{array}{llll}
a & \in \text{Addr} & \triangleq & [0, \text{AddrMax}] \\
p & \in \text{Perm} & ::= & \text{O} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
 & \mid \text{RWL} \mid \text{RWLX} \mid \text{URW} \mid \text{URWL} \mid \text{URWX} \mid \text{URWLX} \\
g & \in \text{Global} & ::= & \text{GLOBAL} \mid \text{LOCAL} \\
c & \in \text{Cap} & \triangleq & \{(p, g, b, e, a) \mid b, e, a \in \text{Addr}\} \\
w & \in \text{Word} & \triangleq & \mathbb{Z} + \text{Cap} \\
r & \in \text{RegName} & ::= & \text{pc} \mid \text{r}_0 \mid \text{r}_1 \mid \ldots
\end{array}
\qquad
\begin{array}{llll}
reg & \in \text{Reg} & \triangleq & \text{RegName} \to \text{Word} \\
m & \in \text{Mem} & \triangleq & \text{Addr} \to \text{Word} \\
\varphi & \in \text{ExecConf} & \triangleq & \text{Reg} \times \text{Mem} \\
\delta & \in \text{DoneState} & ::= & \\
 & \text{Standby} \mid \text{Halted} \mid \text{Failed} & & \\
\mu & \in \text{ExecMode} & ::= & \\
 & \text{SingleStep} \mid \text{Repeat } \mu \mid \text{Done } \delta & &
\end{array}
$$

$$
\begin{array}{lll}
\rho & \in & \mathbb{Z} + \text{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \\
 & & \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt} \mid \texttt{loadU}\ r\ r\ \rho \mid \texttt{storeU}\ r\ \rho\ \rho \mid \texttt{promoteU}\ r
\end{array}
$$

Fig. 1. Machine words, machine state and instructions.

A memory word $w \in \text{Word}$ is either an (unbounded) integer or a capability c. Capabilities are of the form (p, g, b, e, a) and allow exerting permissions p over the memory range $[b, e)$, while currently pointing to a. The permissions p and locality bit g appear in the permission and locality lattices of Figure 2, which induce a bottom-to-top partial order $\preccurlyeq$ on permissions, localities and pairs thereof. The locality bit g only plays a role in presence of local capabilities, and will be covered later in Section 2.2. The permission lattice, on the other hand, contains six different types of permissions; the null (O), read-only (RO), enter (E), read/write (RW), read/execute (RX) and read/write/execute (RWX) permissions. The sole non-standard permission, E, is inspired by the M-Machine [Carter et al. 1994]. Enter capabilities represent opaque closures, or object capabilities, encapsulating code and data, and hence cannot be read, written, executed or modified. They can only be jumped to, thereby loading them into the pc register and changing their permission from E to RX, effectively unsealing them. The operational semantics will further illustrate the use of enter capabilities.

The machine's instructions i either operate on register names r, or on sums ρ of registers and constants. We detail their semantics below.

The state of the machine is modeled by the semantics as a configuration φ, containing the state of the registers $\varphi.reg$ and the memory $\varphi.m$. A register file reg consists of a map from register names r to words, while the memory m maps addresses to words.

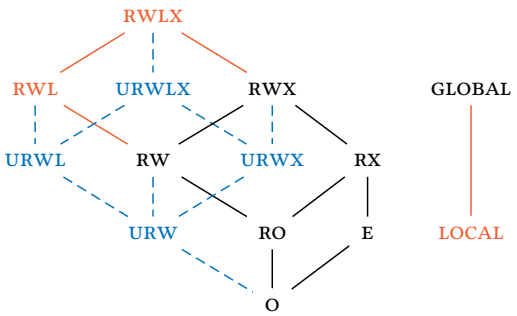


Fig. 2. Permission and locality hierarchy.

Figure 3 defines the small-step operational semantics for the capability machine. At each step, the machine's state is described by an execution mode μ and a configuration φ. The mode μ models the machine's instruction cycle, which loops infinitely (expressed by Repeat μ) until it reaches a successful done state Done Halted through RepeatHalt or a failed state Done Failed through RepeatFail. The RepeatSingle rule allows for the execution of single instructions through the ExecSingle rule. If the execution of the instruction is successful, i.e. execution in ExecSingle does not fail or halt and results in a Done SingleStep state, then RepeatStandby allows for another iteration of the processor's instruction cycle.

$$\textsc{RepeatSingle}\quad \frac{(\text{SingleStep}, \varphi) \to (\text{Done } \delta, \varphi')}{(\text{Repeat SingleStep}, \varphi) \to (\text{Repeat (Done } \delta), \varphi')}$$

$$\textsc{RepeatStandby}\quad (\text{Repeat (Done Standby)}, \varphi) \to (\text{Repeat SingleStep}, \varphi)$$

$$\textsc{RepeatHalt}\quad (\text{Repeat (Done Halted)}, \varphi) \to (\text{Done Halted}, \varphi)$$

$$\textsc{RepeatFail}\quad (\text{Repeat (Done Failed)}, \varphi) \to (\text{Done Failed}, \varphi)$$

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

Fig. 3. Operational semantics: reduction steps.

An execution step (ExecSingle) requires an executable and in-bounds capability in the pc register, failing otherwise. It reads the word z at the memory address a, decodes it and executes the result on the current state φ, denoted $[\![decode(z)]\!](\varphi)$. Figure 4 defines the operational behavior $[\![i]\!](\varphi)$ for a number of representative instructions i. The notation $\in$ is overloaded to deconstruct sum types, e.g. if $\rho \in \mathbb{Z} + \text{RegName}$, then the statement $\rho \in \mathbb{Z}$ will automatically unwrap ρ if it is of the form inl _ and fail otherwise. 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 e.g. $a + 1$ can result in illegal addresses, and *should* hence be represented as an option type. To avoid notational clutter, we assume this option type to be automatically unpacked through in the entire figure, resulting in failure in case of a None result. If an instruction operates on a value ρ, it either uses the constant value directly if $\rho \in \mathbb{Z}$, or it reads the value from the register if $\rho \in \text{RegName}$. In what follows, *the contents of* ρ will be used to signify the resulting value of either option.

We now describe the semantics of instructions, in particular those listed in Figure 4. The `fail` and `halt` instructions terminate execution in the Failed and Halted state respectively. `move` r ρ copies the contents of ρ into r. Memory is accessed using the `load` and `store` instructions: `load` r_1 r_2 reads the value pointed by the capability in r_2 provided it has the permission R and points within bounds, and `store` r ρ stores the contents of ρ through the capability in r provided it has the W permission and points within bounds. The `jmp` instruction jumps to a capability, by writing it into the pc register. In the case of an enter (E) capability, it unseals it into a RX capability first, allowing us to jump to opaque closures, as previously mentioned. Three instructions allow modifying capabilities. `restrict` r ρ allows restricting the permission and locality of a capability, by decoding the contents of ρ into a pair (p', g'), and provided it is less permissive than the current permission-locality-pair of r according to $\preccurlyeq$, restricts r accordingly. `subseg` r ρ_1 ρ_2 takes a subsegment of a capability range of authority. It uses the contents of ρ_1 and ρ_2 to restrict the range of authority of the capability in r, in case r is not an enter capability. Note that the inequality $0 \le z_2 < e$ suffices to guarantee monotonicity of authority, since if $z_2 \le z_1$, then the capability provides no authority over memory whatsoever. `lea` r ρ modifies the address of the capability in r, by adding to it the integer offset in ρ. As expected, `lea` fails for enter capabilities. A number of instructions allow inspecting capabilities. We show `geta` that retrieves the address field of a capability; `getp`, `getl`, `getb` and `gete` work similarly for the other fields. Not shown in Figure 4 are `jnz` (conditional jump), arithmetic instructions (`add`, `sub`, `lt`) and `isptr` which checks whether a word is a capability.

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

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

i	$[\![i]\!](\varphi)$	Conditions
`fail`	(Done Failed, φ)	
`halt`	(Done Halted, φ)	
`move` r ρ	updPC(φ[reg.$r \mapsto w$])	$w = \mathrm{getWord}(\varphi, \rho)$
`load` r_1 r_2	updPC(φ[reg.$r_1 \mapsto w$])	$\varphi.\text{reg}(r_2) = (p,g,b,e,a)$ and $w = \varphi.\text{mem}(a)$ and $b \le a < e$ and $p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}, \text{RWL}, \text{RWLX}\}$
`store` r ρ	updPC(φ[mem.$a \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and $b \le a < e$ and $p \in \{\text{RW}, \text{RWX}, \text{RWL}, \text{RWLX}\}$ and $w = \mathrm{getWord}(\varphi, \rho)$ and if $w = (\_, \text{LOCAL}, \_, \_, \_)$, then $p \in \{\text{RWLX}, \text{RWL}\}$
`jmp` r	(Done Standby, φ[reg.pc $\mapsto$ *newPc*])	if $\varphi.\text{reg}(r) = (\text{E}, g, b, e, a)$, then $newPc = (\text{RX}, g, b, e, a)$ otherwise $newPc = \varphi.\text{reg}(r)$
`restrict` r ρ	updPC(φ[reg.$r \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and $(p',g') = \mathrm{decodePermPair}(\mathrm{getWord}(\varphi, \rho))$ and $(p',g') \preccurlyeq (p,g)$ and $w = (p',g',b,e,a)$
`subseg` r ρ_1 ρ_2	updPC(φ[reg.$r \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and for $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \le z_1$ and $0 \le z_2 \le e$ and $p \neq \text{E}$ and $w = (p,g,z_1,z_2,a)$
`lea` r ρ	updPC(φ[reg.$r \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and $z = \mathrm{getWord}(\varphi, \rho)$ and $p \neq \text{E}$ and $w = (p,g,b,e,a+z)$ and if $p = \text{U-}$, then $z \le 0$
`geta` r_1 r_2	updPC(φ[reg.$r_1 \mapsto a$])	$\varphi.\text{reg}(r_2) = (\_,\_,\_,\_,a)$
`loadU` r_1 r_2 ρ	updPC(φ[reg.$r_1 \mapsto w$])	$\varphi.\text{reg}(r_2) = (p,g,b,e,a)$ and $p = \text{U-}$ and $off = \mathrm{getWord}(\varphi, \rho)$ and $b \le a + off < a \le e$ and $w = \varphi.\text{mem}(a + off)$
`storeU` r ρ_1 ρ_2	updPC(φ' [mem.$(a + off) \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and $p = \text{U-}$ and $off = \mathrm{getWord}(\varphi, \rho_1)$ and $w = \mathrm{getWord}(\varphi, \rho_2)$ and if $w = (\_, \text{LOCAL}, \_, \_, \_)$ then $p \in \{\text{URWLX}, \text{URWL}\}$ and $b \le a + off \le a < e$ and if $off \neq 0$ then $\varphi' = \varphi$ else $\varphi' = \varphi$[reg.$r \mapsto (p,g,b,e,a+1)$]
`promoteU` r	updPC(φ[reg.$r \mapsto w$])	$\varphi.\text{reg}(r) = (p,g,b,e,a)$ and $p = \text{U}\pi$ and $w = (\pi,g,b,\min(a,e),a)$
…		
_	(Done Failed, φ)	otherwise

Fig. 4. Operational semantics: instruction semantics.

Finally, if the capability checks for an instruction are not satisfied, the last row defines the resulting state as (Done Failed, φ).

2.2 Capability Machine with Local Capabilities

The red parts of Figures 1 to 4 add local capabilities to our bare-bones capability machine. The locality hierarchy in Figure 2 receives a second element, LOCAL. As evident from this hierarchy, the `restrict` instruction allows deriving local capabilities from global ones, but not vice versa.

Local capabilities can only be stored to memory through capabilities with a write-local permission, a stronger version of the W permission that we denote as WL. The permission hierarchy in Figure 2

(a) The stack capability Alice starts out with. (b) The stack capability Alice hands to Bob.

Fig. 5. Register state for Scenario 2, involving a write-local stack capability.

contains the two new write-local permissions RWL and RWLX at the top. The permission RWLX is a valid additional permission for the pc-register, as shown in Figure 3. The `restrict` instruction follows the order $\preccurlyeq$ and allows deriving writable capabilities from write-local ones.

The semantics of locality comes into play when interacting with memory, i.e. in the `load` and `store` instructions in Figure 4. Both `load` and `store` permit loading, respectively storing, using the two new permissions. Additionally, `store` only permits storing local values if the capability's permission allows local writes.

3 REVOCATION USING LOCAL CAPABILITIES

We now discuss the use of local capabilities as a (flawed) revocation primitive. We use an incremental example consisting of **three scenarios** which build towards the secure calling convention of Skorstengaard et al. [2018]. It will become clear why local capability revocation and the calling convention incur inherent performance issues because of stack clearing.

3.1 Using Local Capabilities for Revocation

Consider the following scenario, which we will refer to as **Scenario 1**: a client, Alice, wishes to invoke an untrusted adversary, Bob, twice. Alice owns a capability, c, that she wishes to share with Bob, through a register r, but only for the duration of the first call. During the second call to Bob, he should not be able to access the capability any more. In other words, Alice wishes to revoke capability c before the second call. If c is a GLOBAL capability, i.e. $c = (p, \text{GLOBAL}, b, e, a)$, Bob can simply store c in any part of memory he has access to, during the first invocation, and retrieve it during the second, thwarting Alice's plans. This is where local capabilities come in. In case c is local, i.e. $c = (p, \text{LOCAL}, b, e, a)$, and we disregard write-local permissions for the moment, Bob cannot store the capability c to memory for later use, and can therefore not recover c during the second invocation, provided Alice cleared it from the registers before the second call. In other words, as soon as Bob returns to Alice, Bob's access to c is effectively *revoked*.

3.2 Write-local Memory and Stack Clearing

The situation changes when we consider the existence of write-local permissions in an extended **Scenario 2**. Specifically, we extend Scenario 1 to handle the stack explicitly, through a local, write-local stack capability c_{stk} stored in a register we call r_{stk}, as shown in Figure 5a. Concretely, c_{stk} carries RWLX permission. The c_{stk} capability is write-local, to allow spilling of local arguments and other capabilities onto the stack. It will become clear in Section 3.3 why c_{stk} needs an execute permission. Finally, we cannot allow c_{stk} to be GLOBAL, since Bob could then, during the first invocation, store c_{stk} to memory, write c into c_{stk}, and then, during the second invocation, read

c again after retrieving c_{stk}. Since GLOBAL, write-local capabilities clearly break any attempts at building a sensible revocation schema using local capabilities, we forbid their existence.

Figure 5a shows the initial contents of r_{stk}, when Alice starts executing. When Alice calls Bob, she will `restrict` the stack capability and pass the unused part of the stack in r_{stk}, as shown in Figure 5b. At the time of the first call, we set $c_{\text{stk}} = (\text{RWLX}, \text{LOCAL}, b', e', b')$. For simplicity, we assume that c_{stk} has the same value on the second call, i.e. client's stack frame does not change size in between calls. Notice that it is currently unclear how Alice obtains this c_{stk} capability for the second call, since c_{stk} itself is local and hence not easily stored in between the first and second call to Bob. We will clarify this point in Section 3.3. We also assume that c_{stk} is initially zeroed out.

Alice still wants to prohibit Bob from accessing c during the second call. But now, Bob *does* have a way of storing local capability c during the first invocation; he can store it anywhere in $[b', e')$ through the write-local capability c_{stk}. Therefore, Alice has to make sure that the region $[b', e')$ does not contain any copies of c before invoking Bob a second time. The solution is to clear $[b', e')$ before the second invocation, or more generally, *clear all write-local memory* that Bob had access to. This means a potentially large runtime overhead, since the region $[b', e')$ may be quite large in practice. Note that we assume (here and elsewhere in the paper) that the stack is the only memory region that has write-local capabilities pointing into it; otherwise, Alice would have to find and clear all other write-local regions that Bob might have had access to as well, to ensure that he did not store the capability c there.

Additionally, Alice wants to enforce *local state encapsulation*, i.e. ensure that Bob cannot gain access to her local stack frame, including e.g. the value 42 stored at address a_{42} in Figure 5b. This is currently trivially enforced by not passing Bob a reference to the full stack capability.

3.3 A Secure Calling Convention using Local Capabilities

Having discussed the core performance issue in the calling convention of Skorstengaard et al. [2018], namely the stack clearing caused by the use of local capabilities, we now extend our previous scenario to their full secure calling convention. Concretely, we need to make two additions.

First, the astute reader may have noticed that our scenario from Section 3.2 does not actually work. The problem is that after Bob returns, Alice has no capability to erase Bob's part of the stack $[b', e')$, or to access her old stack frame, since Alice's stack capability was itself local, and could only have been stored on the stack itself. We could require Bob to return his own stack capability, but Alice would still have no way of accessing her own stack frame after the first call to Bob. To remedy this, Skorstengaard et al. [2018] have Alice create a kind of return closure on the stack, and pass a capability for invoking it to Bob as a return capability c_{ret}. This capability is represented as an enter capability and points to restoration instructions pushed onto Alice's stack frame, along with her stack pointer, before invoking Bob. When executed, these restoration instructions reinstate Alice's old stack pointer and then resume execution by loading a previously pushed value for the pc register. The execution of these instructions on the stack is the reason we gave c_{stk} execute permission in Section 3.2. Since enter capabilities are opaque, Bob can only use c_{ret} as a jumping destination, and when he does, Alice's old return pointer is restored. Bob cannot simply store the capability c_{ret} for later use, since c_{ret} is itself local, as it was derived from the local capability c_{stk} using `restrict`.

Secondly, to ensure generality, we have to assume that Alice is called by a second untrusted party, Charlie, rather than being allowed to initiate execution. In this **Scenario 3**, the stack capability c_{stk} in Figure 5a that was previously assumed to be initially zeroed, is now passed to Alice by Charlie. Charlie has the option to protect his own stack frame by calling Alice in a fashion similar to Figure 5b. Alice again wishes to revoke Bob's access to c and respects local state encapsulation. With the introduction of a second adversary, Alice now also has the extra goal of enforcing *well-bracketed*

control flow, i.e. ensure that Bob cannot bypass Alice and return to Charlie directly. To achieve all three goals, Alice needs to make sure that c_{stk} does not contain any capabilities that Bob should not have access to when invoking him. Since Charlie has access to a larger stack capability than both Alice and Bob, and could have stored his stack or return pointer high up in the stack, Alice now has to additionally erase the entire memory region $[b', e')$ even before the first call to Bob.

When Alice returns to Charlie, Skorstengaard et al. [2018] originally proposed to erase the entire stack $[b', e')$ again, but as they later point out, it suffices for Alice to clear her own stack frame when returning to Charlie [Skorstengaard et al. 2019a]. This is because any stack capabilities that Bob might want to smuggle to Charlie through the stack, ultimately originate from Charlie in the first place, and are not of any added value to him. Sharing his return pointer with Charlie will do Bob no good either, since it will jump to an address within Charlie's own stack. The formalization of this previously informal observation is one of the novelties in our logical relation in Section 6.

4 UNINITIALIZED CAPABILITIES

Now let us introduce *uninitialized* capabilities in Section 4.1 and see how they can be used to solve the issue of stack clearing in Section 4.2.

4.1 Adding Uninitialized Capabilities to the Capability Machine

Uninitialized capabilities are a new form of capabilities that represent read write ability to a region of memory without access to its *current* contents. More specifically, they are represented as new permissions that are counterparts of the ones that have at least read write ability. The blue labels in Figure 2 represent the additions to our permission lattice.

An uninitialized capability ($\text{U}\pi$, g, b, e, a) has permission π on the range $[b, a)$ (the *initialized* part) and *write-only* permission on the range $[a, e)$ (the *uninitialized* part), assuming $b \leq a < e$ for simplicity. For instance, if π is RWX, then the capability can read, write, or execute anything in the initialized part of the capability, but can only write to the uninitialized part[1]. The initialized part of the capability can be extended by writing to the first uninitialized address, i.e. a.

Capabilities that have at least read-write permissions can be restricted to their uninitialized counterparts. Uninitialized capabilities can be further restricted w.r.t. the initialized part, e.g., an URWLX permission can be restricted to an URW permission. Since an uninitialized capability ($\text{U}\pi$, g, b, e, a) represents authority π on the initialized part $[b, a)$, we also allow converting it to a regular capability (π, g, b, a, a) with authority π on the initialized range $[b, a)$, using a new `promote` instruction. We will make use of this instruction to construct return capabilities in Section 4.2.

We now discuss the changes to the operational semantics, indicated in blue in Figure 4. Instead of modifying `load` and `store` to support uninitialized capabilities, we define two new instructions `loadU` and `storeU` that can only be used with uninitialized capabilities. `loadU` r_1 r_2 ρ first checks that r_1 contains a capability ($\text{U}\pi$, g, b, e, a), that $b \leq a + \mathit{off} < a < e$ (where off is the contents of ρ). If both checks succeed, the value at address $a + \mathit{off}$ will be loaded into register r_2. Similarly, `storeU` r ρ_1 ρ_2 checks that r contains a capability ($\text{U}\pi$, g, b, e, a) and $b \leq a + \mathit{off} \leq a < e$ (with off the contents of ρ_1). It will then store the value in ρ_2 into the address $a + \mathit{off}$. If $\mathit{off} = 0$, then the capability in r is incremented.

From a hardware implementation perspective, the new `loadU` and `storeU` instructions do perform more work than `load` and `store`. In particular, they additionally need to compute an addition and an extra bound check. Nevertheless, we expect that this should not drastically change the implementation complexity or the critical path for our new instructions. Woodruff et al. [2019] show

[1]Using an URWLX, URWX or URX capability to execute is actually only possible after first initializing (a part of) it and converting it to a regular capability using `promote`, as explained in the next paragraph.

that bound checks can typically be made efficient by running them in parallel with memory accesses: "any bounds check on the virtual address can be performed in parallel to [address] translation, making memory access a particularly convenient time to perform a bounds check". We believe the same optimisation could be applied to an implementation of `loadU` and `storeU`.

Finally, one instruction must be slightly modified: we cannot allow `lea` to increase the current address of an uninitialized capability, as this would increase its read authority. Therefore, when using `lea` to change the address of a capability ($\text{U}\pi, g, b, e, a$) to a', the machine checks that $a' \leq a$.

4.2 A New Calling Convention

Description of the calling convention. With uninitialized capabilities, we can now revisit the calling convention from Section 3.3 and use uninitialized capabilities to avoid the stack clearing requirement and fix its performance issues. Instead of using a RWLX stack capability, we give it permission URWLX. Let us consider again the example from Section 3.3, but let Alice pass the capability $c_{\text{stk}} = (\text{URWLX}, \text{LOCAL}, b, e, b)$ to Bob. Bob now cannot use c_{stk} to read the contents of $[b, e)$ without overwriting it first, so stack clearing is no longer needed.

Alice still needs to provide an enter capability c_{ret} as a return pointer to Bob. However, Alice must now first `promote` it back into a RWLX capability before she can use `restrict` c_{stk} to create the return capabilty. When Alice returns to Charlie, Charlie regains access to the entire stack, so Alice still needs to clear her own stack frame. This clearing requirement is very reasonable compared to the earlier case, as Alice only needs to clear the part of the stack she has actually used.

We recap the new calling convention formally:

At program start-up. A local URWLX capability stack pointer is in register r_{stk}.
When called by an adversary. Check that the received stack pointer has permission URWLX.
Before calling an adversary. Push activation record to the stack and create a local E-capability to use as return pointer. `Subseg` the stack capability to the unused part. Clear non-argument registers.
Before returning to an adversary. Clear non-return-value registers and the part of the stack we used.

While the changes may seem simple, there are some details to get right. Let's revisit **Scenario 3** from Section 3.3, assume Alice receives stack capability $c_{\text{stk}1} = (\text{URWLX}, \text{LOCAL}, b_1, e, b_1)$ from Charlie and uses range $[b_1, b_2)$ to store data. She now calls Bob with $c_{\text{stk}2} = (\text{URWLX}, \text{LOCAL}, b_2, e, b_2)$. Suppose that after this first call, Alice needs less stack space. She can instead provide $c_{\text{stk}3} = (\text{URWLX}, \text{LOCAL}, b_3, e, b_3)$ as stack capability to Bob with $b_1 \leq b_3 \leq b_2$ for the second call. Alice does not need to clear the range $[b_3, b_2)$ since Bob cannot possibly read it as it is uninitialized. However, when returning to Charlie, Alice must be careful to clear everything she has ever written to, i.e. the whole range $[b_1, b_2)$ and not just $[b_1, b_3)$. This is because Alice cannot be sure that Bob overwrote what is in $[b_3, b_2)$ and she must ensure that any capabilities she may have inadvertently left there are scrubbed before returning to Charlie.

Informal cost analysis. With these details in mind, let's make sure that we can indeed witness a gain in performance. At first glance, since the new calling convention still clears the local stack frame upon return, it could appear as if the new calling convention only provides a minor constant factor performance improvement. However, in the calling convention by Skorstengaard et al. [2018], the *full* stack space (even parts that will never be used) is cleared twice for each call. In our new proposed calling convention, each function only needs to clear its own stack frame once upon return.

Let us consider two concrete scenarios. First, assume that we are making n secure calls in sequence to untrusted components. Let us note m the size of the remaining unused stack space, and c the size of the stack frame that we currently use.

Skorstengaard et al.'s calling convention requires that we clear the whole unused stack space before each call, and finally, clear it again along with our own stack frame before returning. The cost of clearing is then:

$$n * m + m + c$$

In a typical scenario where c is small compared to m, the overall cost of clearing is quadratic: $O(mn)$ (m is typically comparable to the overall available stack space, which counts in megabytes). In the improved calling convention, we only need to clear *our* stack frame before returning. We therefore only need to pay the small price of clearing c memory cells.

Now assume that we are making n nested secure calls, each call using a stack frame of size c.

Skorstengaard et al.'s calling convention now requires that we clear m before the first call and $m + c$ when returning, $(m - c)$ before the second call and m when returning, etc, i.e.:

$$2nm - n(n-1)c + c$$

Again, in a typical scenario where the portion of the stack actually used (nc) is small compared to the available stack space m, the cost remains a quadratic $O(mn)$. With our calling convention, we only need to clear the individual stack frames, which amounts to an overall linear cost of nc. (Notice how this does not depend on the size of the available stack space.)

In summary, it seems like uninitialized capabilities solve the stack clearing requirement and associated performance issue of local capability revocation and the secure calling convention of Skorstengaard et al. [2018, 2019a]. But security of the result relies on subtle arguments and invariants. Fortunately, in the next section, we'll see that we can build on Skorstengaard et al.'s approach for reasoning about capability machines and the guarantees they provide and prove security of local capability revocation and the updated calling convention.

5 PROGRAM LOGIC

In order to reason about the behavior of programs running on the capability machine, we build a program logic on top of the machine operational semantics. The logic provides rules describing the execution of single machine instructions, that can then be used to establish a specification for a complete program running until the machine halts (or fails).

Specifications are written as separation logic triples, both in the case of manually written specifications for concrete programs (such as the macros of Section 6.8), and in the case of the "universal specification" that holds of arbitrary code by the Fundamental Theorem (see Section 6.7). Figure 6 shows specifications for some single machine instructions as well as for a program composed of several instructions (in this case, a simple macro). In a high level language, a separation logic triple $\{P\}\, e\, \{Q\}$ provides a precondition P and postcondition Q for the execution of the expression e. However, in our setting, there is no direct equivalent of e since code executed by the machine is laid out in memory as mere integers that are then decoded into instructions. Instead, we use triples of the form $\{P\}\, \mu\, \{Q\}$, where μ denotes an *execution mode* as defined in Figure 1. Treating execution modes as expressions in this way makes our assembly language fit well into the Iris framework, which is more usually used with lambda calculi. A triple using the SingleStep execution mode specifies the behavior of a single instruction (the one currently pointed to by the program counter). A triple using the Repeat SingleStep execution mode specifies a complete execution, starting from the instruction currently pointed to by the program counter, and continuing until the machine halts or fails.

We use Iris' standard definition of triples, which correspond to partial correctness: correctness does not entail termination. Finally, note that machine failure (e.g. failure to pass a capability check) is modeled explicitly. A failing program does not get stuck, instead, it reduces to a configuration

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}, p') \qquad \mathrm{ValidSubseg}(p, b, e, z_1, z_2) \qquad \mathrm{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2}{\begin{array}{l}\{\mathrm{pc} \mapsto (p_{pc}, g_{pc}, b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_{p'} n * r \mapsto (p, g, b, e, a)\} \\ \quad \text{SingleStep} \\ \{v, v = \text{Done Standby} * \mathrm{pc} \mapsto (p_{pc}, g_{pc}, b_{pc}, e_{pc}, a_{pc}+1) * a_{pc} \mapsto_{p'} n * r \mapsto (p, g, z_1, z_2, a)\}\end{array}}$$

$$\frac{\mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}, p') \qquad \mathrm{ValidStore}(p, b, e, a, p'', w) \qquad \mathrm{decode}(n) = \texttt{store}\ dst\ src}{\begin{array}{l}\{\mathrm{pc} \mapsto (p_{pc}, g_{pc}, b_{pc}, e_{pc}, a_{pc}) * a_{pc} \mapsto_{p'} n * dst \mapsto (p, g, b, e, a) * src \mapsto w * a \mapsto_{p''} -\} \\ \quad \text{SingleStep} \\ \left\{ \begin{array}{l} v, v = \text{Done Standby} * \mathrm{pc} \mapsto (p_{pc}, g_{pc}, b_{pc}, e_{pc}, a_{pc}+1) * a_{pc} \mapsto_{p'} n * dst \mapsto (p, g, b, e, a) \\ * src \mapsto w * a \mapsto_{p''} w \end{array} \right\}\end{array}}$$

$$\frac{\forall i \in [0, n),\ \mathrm{ValidPC}(p, b, e, a_i, p') \qquad n = |\mathrm{rclear\_instrs}\ l|}{\begin{array}{l}\left\{ \begin{array}{l} \mathrm{pc} \mapsto (p, g, b, e, a_0) * \ast\!\!\ast_{r \in l}\ r \mapsto - * \ast\!\!\ast_{i \in [0,n)}\ a_i \mapsto_{p'} (\mathrm{rclear\_instrs}\ l)[i] * \\ \left(\begin{array}{l} \mathrm{pc} \mapsto (p, g, b, e, a_n) * \ast\!\!\ast_{r \in l}\ r \mapsto 0 * \ast\!\!\ast_{i \in [0,n)}\ a_i \mapsto_{p'} (\mathrm{rclear\_instrs}\ l)[i] \mathrel{-\!\!*} \\ \mathrm{wp}\ \text{Repeat SingleStep}\ \{Q\} \end{array} \right) \end{array} \right\} \\ \quad \text{Repeat SingleStep} \\ \{Q\}\end{array}}$$

$$\begin{array}{lcl} \mathrm{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}, p') & \triangleq & p_{pc} \preccurlyeq p' \wedge p' \in \{\textsc{rx}, \textsc{rwx}, \textsc{rwlx}\} \wedge b_{pc} \le a_{pc} < e_{pc} \\ \mathrm{ValidSubseg}(p, b, e, z_1, z_2) & \triangleq & b \le z_1 \wedge 0 \le z_2 \le e \\ \mathrm{ValidStore}(p, b, e, a, p'', w) & \triangleq & \textsc{rw} \preccurlyeq p \preccurlyeq p'' \wedge b \le a < e \wedge \\ & & \text{if } w = (\_, \textsc{local}, \_, \_, \_) \text{ then } p \in \{\textsc{rwlx}, \textsc{rwl}\} \\ \mathrm{rclear\_instrs}\ l & \triangleq & \mathrm{map}\ (\lambda r.\ \mathrm{encode}(\texttt{move}\ r\ 0))\ l \end{array}$$

Fig. 6. Separation Logic specifications for the machine instructions subseg 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.

with the Done Failed execution mode. A postcondition binds the execution mode at the end of the execution, allowing specifications to talk explicitly about failure or success.

As an additional subtlety, note that separation logic triples are not a primitive concept in Iris. Instead, they are a defined as sugar on top of a weakest-precondition combinator

$$\{P\}\,\mu\,\{Q\} \triangleq \Box(P \mathrel{-\!\!*} \mathrm{wp}\ \mu\ \{Q\})$$

The triple $\{P\}\,\mu\,\{Q\}$ specifies that owning the resource P is sufficient to run the machine with mode μ and eventually obtain the postcondition Q. Furthermore, this fact is required to hold not only at the current point, but also to remain true indefinitely, using the Iris modality $\Box$ [see, e.g., Birkedal and Bizjak 2017; Jung et al. 2018]. This "persistent" modality $\Box$ expresses that the proof of a triple may not rely on assumptions that hold now but may cease to hold in the future ("ephemeral assertions"). Instead, it must only rely on assumptions that remain true at any point in the execution of the system ("persistent assertions"), because we may want to invoke this specification at any later point.

Access to registers and memory is described using two separate points-to assertions. The assertion "$r \mapsto w$" asserts that register r currently contains the machine word w, and provides exclusive

ownership over that register. The assertion "$a \mapsto_p w$" asserts that the memory location at address a currently contains the machine word w and provides ownership over that location. Furthermore, access to the location is restricted with permission p: for instance, if p is RO then it is not possible to modify the value stored at that location. More generally, when accessing a memory location with permission p using a capability with permission p', the permission of the capability must be included in the permission for the location, i.e. $p' \preccurlyeq p$.

The first two rules of Figure 6 show specifications for the `subseg` and `store` instructions. Their respective preconditions describe the subset of the machine state accessed by the instruction, and the postconditions describe the updated state after executing the instruction. For both specifications, the postcondition asserts that the execution mode after executing the instruction is Done Standby, meaning that the machine instruction always succeeds under the premises of the specification. The first rule 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, if the register r contains a capability, and assuming that the program counter is valid (ValidPC(...)) and that z_1 and z_2 are valid new bounds (ValidSubseg(...)), then the machine successively increments the program counter and restricts the capability held in register r with new bounds z_1 and z_2. Similarly, the second rule states that successfully executing the `store` instruction reads a word from the *src* register and writes it into the memory location pointed to by the capability in the *dst* register.

The specifications that appear in Figure 6 for `subseg` and `store` are in fact not the most general specifications for these instructions. They assume that some side-conditions hold and establish that the execution succeeds, making them useful for reasoning about the correctness of a concrete program. However, there are many ways in which instructions can fail: because of capability checks, but also, for example, because incrementing the program counter or performing address arithmetic can fail since we have finite memory. Our program logic thus also provides rules (not reproduced here) to reason about cases where executing an instruction fails. Furthermore, "most general" specifications covering all cases are also provided, that are useful not only as a proxy for deriving more specific rules, but also directly in the proof of the Fundamental Theorem (Theorem 6.1), for characterizing the behavior of arbitrary instructions that might or might not fail.

Our machine code does not have primitive mechanisms for structured control flow. Similarly, our program logic does not make assumptions about program control flow. Instead, programs composed of several instructions are specified in continuation-passing style: one proves a specification for a complete execution of the machine, starting at the beginning of the program, by assuming a specification for the continuation of the program, which is reached either through sequential instruction fetch, or through a `jmp` instruction.

The last rule of Figure 6 exemplifies such a specification for a program composed of several instructions; the `rclear` macro. This macro clears a number of registers by setting their contents 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 as a triple using the Repeat SingleStep execution mode, meaning that the specification covers a full execution of the machine, and prove that starting before the execution of `rclear`, to reach any postcondition Q (describing the state of the machine at the very end of the execution) it is enough to prove that one can reach Q from the *continuation*, i.e. after `rclear` as been executed. In other words, the postcondition of `rclear` is given as the precondition of its continuation.

Concretely, the specification of `rclear` assumes 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 instructions, then all the registers in l have been cleared and now contain 0. Importantly, notice that the specification for the continuation of `rclear` is given not as a separation logic triple, but directly

in terms of the weakest-precondition combinator. Unlike triples, this specification is not required to be persistent (note the absence of □). Indeed, it only makes sense to call to this specification once, at the point of the execution where the continuation is reached (i.e. when pc reaches (p, g, b, e, a_n)).

6 LOGICAL RELATION MODEL

Now that we have this program logic, we can explain the most important contribution of this paper: the formalization and proof of *capability safety*. This is the set of guarantees that the capability machine provides for untrusted code and it includes both general capability safety guarantees and guarantees that are specific to local and uninitialized capabilities.

While our program logic (Section 5) provides rules for concrete machine instructions which are useful to verify known concrete code, capability safety provides a universal contract that holds for unknown, arbitrary code. Thanks to the capability checks implemented by the capability machine, an arbitrary piece of code cannot behave completely arbitrarily: it is limited by the set of capabilities it has access to. Our logical relation model thus captures how we can reason about the interaction of known and unknown code, and in particular which guarantees one exactly gets from the revocation mechanism enabled by local and uninitialized capabilities.

For readability, we introduce the required machinery gradually, starting with a simple formulation of capability safety without support for revocation (Section 6.1). Next, we provide some intuitions on what needs to change for supporting revocation in Section 6.2. This motivates the need for a form of Kripke worlds with public/private transitions, and standard and custom resources, which we explain and apply in Sections 6.3 to 6.5. In Section 6.6, we provide more technical details on how we combine Iris invariants and saved predicates with more traditional Kripke world-indexing. The Fundamental Theorem, which establishes that our machine indeed satisfies the capability safety formalized by the logical relation, is discussed in Section 6.7. Finally, we demonstrate reasoning about examples with revocation, by outlining a proof of the classic awkward example in Section 6.8.

6.1 A Version of the LR without Kripke Worlds/Local Capabilities

Following Skorstengaard et al. [2018], we formulate the guarantees provided by the capability machine as a logical relation, capturing an informal property known as capability safety [Miller 2006]. Intuitively, the idea is to define the authority represented by a capability. The guarantees provided by the machine then amount to the fact that arbitrary code can never exceed the authority of the capabilities it has access to, or create capabilities with larger authority.

To formalize these intuitions, we define a maximum bound on the authority of a capability using a notion of safety with respect to a set of registered invariants. A capability will be considered safe if it cannot be used in any way to break those invariants. This intuition is instantiated differently for different types of capabilities. For example, memory capabilities are safe if they only grant access to memory that is guaranteed to contain safe values by an invariant. Updating the memory with safe values must not break registered invariants. If the capability is executable, jumping to it with safe words in the registers must respect invariants and produce safe result values.

The reader may notice that this intuitive definition of safety is problematically circular. This is commonly referred to as the world circularity problem [Ahmed 2004; Birkedal et al. 2011]. Skorstengaard et al. [2018] resolve it for their model using step-indexed Kripke logical relations [Ahmed 2004; Birkedal et al. 2011]. We define our logical relations model in Iris so that we can use (1) its built-in support for guarded recursion, which we can use to replace manual bookkeeping of step-indexes, and invariants for reasoning about shared state, and (2) its implementation in Coq and the associated interactive proof mode [Krebbers et al. 2017b].

Formally, we define in Figure 7 three mutually recursive logical relations. The value relation $\mathcal{V} : \text{Word} \to \mathit{iProp}$ defines what it means for a word to be safe, the expression relation $\mathcal{E}$:

$$\boxed{\mathcal{E}(v)} \quad \triangleq\ \forall reg,\ \mathcal{R}(reg) * \mathrm{pc} \mapsto v * \text{✻}_{(r,w)\in reg, r\neq \mathrm{pc}}\, r \mapsto w \mathrel{-\!\!*} \mathrm{wp\ Repeat\ SingleStep}\left\{v, v = \mathrm{Done\ Halted} \Rightarrow \exists reg', \text{✻}_{(r,w)\in reg'}\, r \mapsto w\right\}$$

$$\boxed{\mathcal{R}(reg)} \quad \triangleq\ \text{✻}_{(r,w)\in reg, r\neq \mathrm{pc}}\, \mathcal{V}(w)$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z), \mathcal{V}(\mathrm{O}, -) & \triangleq \top \\ \mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \square \triangleright \mathcal{E}(\mathrm{RX}, b, e, a) \\ \mathcal{V}(p, b, e, a) & \triangleq \text{✻}_{a'\in[b,e)}\ \exists p', p \preccurlyeq p' \wedge \boxed{\exists w, a' \mapsto_{p'} w * \mathcal{V}(w)} \end{cases}$$

Fig. 7. Safety without Revocation.

Word → *iProp* expresses what it means for a program counter to be safe to execute and the relation $\mathcal{R}$: (RegName → Word) → *iProp* expresses that a register file is safe if all register values are safe.

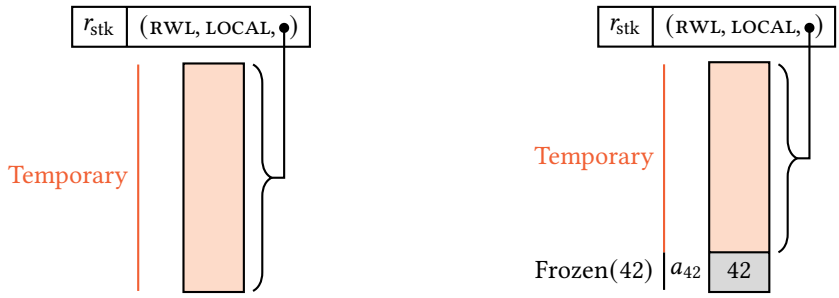
We define safety of words $\mathcal{V}$ as a guarded fixed point: each recursive occurrence of $\mathcal{V}$ is either guarded by the so-called "later" modality ▷ or appears inside an Iris invariant, indicated by the boxed assertion, and thus Iris guarantees that $\mathcal{V}$ is well-defined. For space reasons, we will not explain the later modality or Iris invariants technically; readers who are unfamiliar with them may interpret ▷P to mean that P holds after one step of execution and think of an Iris invariant as a property that remains valid at every step of execution.

We define the expression relation $\mathcal{E}$ as a program specification, expressed using the weakest-precondition combinator. Conceptually, the body of $\mathcal{E}$ can be read as a Hoare-triple (see Section 5), except that it is not required to be persistent. A word v is in the expression relation—i.e. it is safe to execute—if one can run the machine with v in the pc register, and safe values in the other registers, provided we temporarily give up ownership of the registers but we get it back afterwards. Note that we do not specify what happens if the machine runs into an error, but only the case where the machine halts gracefully. Now, since we are not requiring any interesting property to hold in the postcondition, it might seem like the definition of $\mathcal{E}$ is trivial and always true! This is not the case, however. For a weakest-precondition assertion to hold within Iris, one additionally needs to prove that all Iris invariants are preserved at every step of the execution. This includes the Iris invariants mentioned in the definition of $\mathcal{V}$, as detailed next.

The value relation, $\mathcal{V}(w)$, defines what it means for a word to be safe. Intuitively, the definition expresses that a word is safe when it cannot be used to violate invariants. There are two modes of usage to consider: (1) read/write authority over an address, and (2) authority to jump to an enter capability. A capability (p, b, e, a) with a permission p other than E or O grants read/write authority over each address a within its range of authority. It is in the value relation, if for each a within $[b, e)$, there exists an Iris invariant (indicated by the boxed assertion) which owns the memory location and guarantees that it will always contain a safe value. Note that the invariant is allowed to hold a stronger permission $p' \succcurlyeq p$ (so that we can easily downgrade capabilities' permissions).

A capability with an enter (E) permission is a special case: it cannot be used directly to read values from memory, so we do not require safety of the values it points to. Instead, its safety only requires that the capability is safe to execute (by the expression relation) after changing its permission to RX (as happens when invoking an enter capability). Since this capability may be jumped to at any point of the execution, this fact needs to hold persistently, hence the "box" modality.

Interestingly, the safety of executable capabilities (RX or RWX) does not require any additional conditions. As we will see in Section 6.7, this is because we are formalizing capability safety: a property that holds for arbitrary code. As such, we could in principle allow the adversary to execute any capability it has read access to and in fact, all executable permissions in the lattice of Figure 2 also have read permission. In fact, even if we give an adversary read but not execute permission



(a) The stack Alice receives from Charlie.

(b) The stack Alice passes to Bob.

Fig. 8. A scenario where a stack capability is passed in a register r_{stk} between different parties.

over some memory, we already cannot prevent them from executing the instructions anyway: as soon as they have writable and executable access to any block of memory $[b, e)$ elsewhere, they can simply copy the instructions into the range $[b, e)$ and jump to them there.

6.2 Reasoning about Revocation

The logical relation from the previous section is relatively easy to understand, but only captures a basic form of capability safety. In the following sections, we extend it to support local and uninitialized capabilities as well as revocation.

To understand what needs to change, we first take another look at scenario 3 from Section 3.3, using the illustrations in Figure 8. In this scenario, Alice receives a stack capability c_{stk} from Charlie in some register r_{stk}, as shown in Figure 8a. Alice knows that Charlie only has access to safe capabilities, so every address a in the range of c_{stk} must be owned by an invariant $\boxed{\exists w, a \mapsto_{p'} w * \mathcal{V}(w)}$. These invariants are depicted as Temporary in Figure 8, a term that we will explain in the next sections. This invariant means that any component in the system is allowed to change the content of the memory cell at a to any safe value w.

However, when Alice invokes Bob, the situation is different. Alice has now stored the value 42 in location a_{42} and expects Bob to not be able to change this value (see Figure 8b). To this end, Alice uses local capabilities to revoke Bob's read/write access to part of the stack and only allow him to modify the other parts. In other words, the invariant $\boxed{\exists w, a \mapsto_{p'} w * \mathcal{V}(w)}$ that used to govern the memory location a_{42} should no longer be active. Instead, it should be replaced by a new invariant expressing Alice's intention: the memory location should now be frozen: it should not be modifiable and only be allowed to contain 42, as shown in Figure 8b.

Replacing this old invariant with such a frozen invariant also means that capabilities that used to be safe are not safe any more. Specifically, a read/write memory capability c whose range includes a (e.g. Charlie's stack capability) will no longer be safe as the required invariant has been replaced. This observation makes a lot of sense: in this scenario, such a capability is really not safe anymore to pass to Bob, as he could use it to break the new frozen invariant.

In other words, reasoning about local capabilities and revocation requires two things that are impossible in the logical relation from Section 6.1. First, general Iris invariants cannot be deactivated (except temporarily during a single atomic step, but that's not what we need). Once they are defined, they remain active during the rest of the execution of the system. Second, the logical relation does not allow a capability to be safe at one moment but become unsafe later (when certain invariants have been revoked): the value relation is simply a predicate on words and if it is true, it remains true forever.

Moreover, we also need to ensure that an adversary does not deactivate an invariant, without reinstating it when they return. In other words, we need a more refined model, where invariants can be in different states and where safety can depend on these states. Moreover, we must be able to track precisely how these states evolve (to ensure that invariants are properly reinstated when necessary). This final point means that to define the refined model it is not enough simply to replace the general Iris invariants with so-called cancellable invariants. Instead we will parameterize our logical relation by an explicit notion of *world*, which will allow fine-grained control over invariant states.

6.3 Kripke Worlds to Track the State of Invariants

We change the signature of the value relation as follows: the safety of a word can now depend on a *world* representing the currently active invariants: $\mathcal{V} : \text{WORLD} \to \text{Word} \to \mathit{iProp}$.

Some readers may notice that our value relation now has the same signature as a step-indexed Kripke logical relation, but we hasten to point out that our worlds are much simpler than is typical in such settings. In earlier work, e.g., [Ahmed et al. 2009; Dreyer et al. 2010; Skorstengaard et al. 2018, 2019b], worlds track both invariant states and associated predicates (which are also world-indexed) on memory and are therefore recursively defined. Here instead, worlds track only the states of invariants and in Section 6.6, we will discuss how the associated predicates on memory are tracked using an Iris mechanism called saved predicates[Jung et al. 2016, 2018].

Before we move on to the definition of worlds, there is a final important observation to make in the revocation scenario we discussed. As discussed, Alice revokes the old invariant for location a_{42} before invoking Bob and as discussed, this will break the safety of some capabilities. However, not all invariants can be revoked in this way and also, not all capabilities will be made unsafe by revoking an invariant. To understand this, consider that it is easy to control the local capabilities that an adversary has access to: they must reside in the registers or in memory that the adversary receives write-local access to. However, the same is not true for global capabilities: the adversary might have previously stored those in arbitrary memory and we have no way to revoke them. Since we can't revoke an adversary's access to global capabilities, it should not be possible to revoke invariants which their safety depends on. Conversely, global capabilities' safety should be able to survive the revocation of invariants like the one for a_{42}.

What this means is that we need to distinguish two kinds of invariants: (1) non-revocable ones, which global capabilities' safety may depend on, and (2) revocable ones, which global capabilities' safety must not depend on. Revoking the latter may affect the safety of local capabilities but not global capabilities. To formalize this, we follow previous work [Dreyer et al. 2010; Skorstengaard et al. 2019a] and distinguish public and private world updates. The former are those which cannot break safety of any capabilities (e.g. adding new invariants for previously unused memory) while the latter are updates which may break safety of local capabilities but not global capabilities (e.g. revoking invariants, adding new invariants for unused memory). If a world W' can be reached from W using public transitions alone, we call it a public future world ($W' \sqsupseteq^{pub} W$) and similarly for private transitions and private future worlds ($W' \sqsupseteq^{priv} W$).

Our worlds assign to memory locations a logical state belonging to a small protocol tailored to talk about revocation. This "standard" protocol uses four possible states. A location can be either in the Temporary, Frozen, Permanent or Revoked state: The first two are revocable (consequently, global capabilities may not depend on them), the third is not (consequently, global capabilities can depend on them).

- The Temporary state represents the invariant that a location may only contain safe words, including local capabilities. This type of invariant is intended to cover memory locations in the stack, which are allowed to contain local capabilities.

- The Permanent state represents the invariant that a location contains safe words, but only those whose safety will survive private updates, i.e., no local capabilities.
- The Revoked state corresponds to the result of revocation: a location that was previously Temporary or Frozen but got revoked. This means we know nothing about the contents of the memory at this location: conceptually, someone has taken control over the location, and needs to do some work to reinstantiate the invariants and restore safety of capabilities for it.
- Finally, the Frozen state asserts that we know the exact (not-necessarily-safe) value stored at the location, and it is not allowed to change. Frozen states are used for two purposes: (1) to keep a local stack frame frozen during a call to an adversary and (2) to freeze the uninitialized part of a capability. Indeed, locations in the uninitialized part of a U- capability will point to the same word right until they are written to. Whenever an uninitialized capability is purposely uninitialized (when passing it to an adversary), the Frozen state will allow us to remember the old, now unsafe value it still contains. If the word is never overwritten, then that knowledge can be used to reinstate the address to its previous Temporary state.

We call these states the *standard* states, StdStates.

Invariants represented by these standard states are collected in W^{std}, the first component of a world W. It is a partial map from addresses to standard states: $W^{std} : Addr \hookrightarrow \mathrm{Ex}(\mathrm{StdStates})$. Here Ex refers to the Iris notion of an exclusive resource algebra—readers who are unfamiliar with Iris can ignore it. This map only tracks the states of shared resources, i.e. those that safe capabilities can range over. The shared resources are exactly those that are associated to the standard behaviour. In Section 6.5, we will explain a second component of W, which collects other, custom invariants. Such custom invariants are never directly addressable by safe capabilities, but they are necessary for modeling advanced examples (closures with non-trivial local state), see, e.g., Section 6.8.

6.4 The Logical Relation with Support for Revocation

Let us now take a look at Figure 9 and see how the logical relation is updated to use these worlds. The differences with the LR from Section 6.1 are highlighted in blue. Apart from the addition of world parameters W, the changes are concentrated around the validity of a read-write capability. Instead of requiring the presence of an Iris invariant, that condition now formalizes the intuitive idea mentioned above: $rel(a, p', \mathcal{V})$ associates a memory invariant (namely $\mathcal{V}$ itself) to address a using saved predicates, whereas $\mathcal{S}$ and $\mathcal{S}^u$ associate the address a to its state. More precisely, the state relation $\mathcal{S}(W)(a, g, p)$ looks at the locality g and the permission p, and requires W to contain the appropriate state in W^{std}. The uninitialized state relation $\mathcal{S}^u(W)(a, g, p, mid)$ does the same but for U- permissions, for which the required state also depends on the boundary mid between the initialized and uninitialized part (i.e. the current address a of the uninitialized capability). The resource $rel(a, p', \mathcal{V})$ will be discussed later in Section 6.6.

We highlight what the states are for some interesting cases of safe capabilities:

- A capability with a RWLX permission (which must be itself local) is in $\mathcal{V}(W)$ if each address within its range of authority is in a Temporary state of W^{std} (so the address can be used to store local capabilities).
- A capability in $\mathcal{V}(W)$ with a URWLX permission and LOCAL locality, currently pointing to the address mid, has all addresses $a < mid$ in a Temporary state, whereas it has all addresses $a \geq mid$ either in a Temporary state or Frozen at some hidden word w.
- Global capabilities in $\mathcal{V}(W)$ have all addresses in their range of authority in a Permanent state, regardless of their permission.

In addition, the value relation for enter capabilities now quantify over future worlds $W' \sqsupseteq^g W$. For g local resp. global, this means that the execution of the capability must hold in arbitrary public

$$\boxed{\mathcal{E}(W)(v)} \triangleq \forall reg,\ \mathcal{R}(W)(reg) * sharedResources(W) * stsCollection(W) * \mathrm{pc} \mapsto v * \circledast_{(r,w)\in reg/\mathrm{pc}}\ r \mapsto w \twoheadrightarrow \text{wp Repeat SingleStep} \left\{ \begin{array}{l} v, v = \text{Done Halted} \rightarrow \exists W'\ reg',\ W' \sqsupseteq^{priv} W \\ * sharedResources(W') * stsCollection(W') \\ * \circledast_{(r,w)\in reg'}\ r \mapsto w \end{array} \right\}$$

$$\boxed{\mathcal{R}(W)(reg)} \triangleq \circledast_{(r,w)\in reg/\mathrm{pc}}\ \mathcal{V}(W)(w)$$

$$\boxed{\mathcal{V}(W)(w)} \left\{ \begin{array}{ll} \mathcal{V}(W)(z), \mathcal{V}(W)(\mathrm{O}, -) & \triangleq \top \\ \mathcal{V}(W)(\mathrm{E}, g, b, e, a) & \triangleq \square\, \forall W' \sqsupseteq^{g} W, \triangleright\ \mathcal{E}(W')(\mathrm{RX}, g, b, e, a) \\ \mathcal{V}(W)(p, g, b, e, a) & \triangleq \circledast_{a'\in[b,e)} \exists p', p \preccurlyeq p' \wedge rel(a', p', \mathcal{V}) \\ & \quad \wedge \begin{cases} \mathcal{S}^{u}(W)(a', g, p, a) & \text{if } p = \mathrm{U}\text{-} \\ \mathcal{S}(W)(a', g, p) & \text{otherwise} \end{cases} \end{array} \right.$$

State relation

$$\mathcal{S}(W)(a, g, p) \triangleq \begin{cases} W^{std}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \wedge g = \mathrm{LOCAL} \\ W^{std}(a) = \text{Temporary} & \text{if write-local}(p) \wedge g = \mathrm{LOCAL} \\ W^{std}(a) = \text{Permanent} & \text{if } g = \mathrm{GLOBAL} \end{cases}$$

$$\mathcal{S}^{u}(W)(a, g, p, mid) \triangleq \begin{cases} \mathcal{S}(W)(a, g, p) \vee \exists w, W^{std}(a) = \text{Frozen}\{[a := w]\} & \text{if } a \geq mid \\ & \wedge\ g = \mathrm{LOCAL} \\ \mathcal{S}(W)(a, g, p) & \text{otherwise} \end{cases}$$

Fig. 9. Safety with Revocation. Differences with Figure 7 are highlighted in blue.

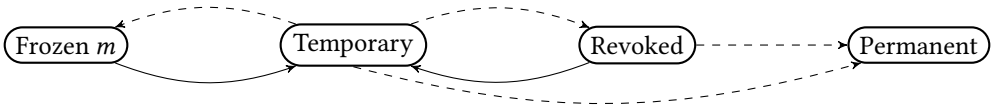


Fig. 10. Standard State Transition System. Full lines indicate public transitions, dashed lines indicate private transitions. Public transitions are also private.

resp. private future worlds. This quantification makes sure that global enter capabilities remain safe when temporary invariants are revoked, and enforces that invariants are properly reinstated.

Finally, the new *sharedResources* and *stsCollection* assertions in the figure are used to ensure that shared memory actually satisfies the memory invariants, which have registered using saved predicates, during execution. This aspect of the LR is a bit technical and will be discussed further in Section 6.6.

6.5 World Updates and Monotonicity

Now let us reconsider our worlds and future world relations in more detail. As already mentioned, temporary invariants may be revoked to obtain a private future world and fresh invariants over unused memory may be added to obtain a public future world. Actually, those are not the only types of updates allowed; Figure 10 depicts the allowed transitions between standard states. Dashed lines in the figure indicate private updates and full lines indicate public ones. One can observe that making a frozen or revoked location temporary is a public update: indeed, doing so can never make safe capabilities unsafe, only the reverse. In contrast, changing the state of a temporary location is a private update, because it may break safety of capabilities depending on it.

$$\text{monoReq}(W, \phi, v, \sqsupseteq) \triangleq \square\, \forall W', W' \sqsupseteq W \to \phi(W, v) \mathrel{-\!\!*} \phi(W', v)$$

$$\text{permR}(a, p, W, \phi) \triangleq \exists v, a \mapsto_p v * \triangleright\ \phi(W, v) * \text{monoReq}(W, \phi, v, \sqsupseteq^{\text{priv}})$$

$$\text{tempR}(a, p, W, \phi) \triangleq \exists v, a \mapsto_p v * \triangleright\ \phi(W, v) * \begin{cases} \text{monoReq}(W, \phi, v, \sqsupseteq^{\text{pub}}) & \text{if write-local}(p) \\ \text{monoReq}(W, \phi, v, \sqsupseteq^{\text{priv}}) & \text{o/w} \end{cases}$$

$$\text{frozenR}(a, p, m, M^{std}) \triangleq a \mapsto_p m(a) * \forall a \in dom(m), M^{std}(a) = \text{Frozen m}$$

Fig. 11. Standard Resources.

We can now give the full definition of WORLD. In addition to the component W^{std} which we have already seen, it contains a second component W^{cus}. This component contains custom state transition systems, whose states can be associated with arbitrary Iris predicates. Such custom invariants are often needed for examples that involve closures with some private state evolving according to a certain ad hoc protocol, like in the example presented in Section 6.8. We remark that the definition of the value relation does not depend on the custom states, but through its quantification over future worlds, the value relation enforces that custom states evolve according to their public and private future world relations.

W^{cus} refers to a map from region names $rn \in \text{RNames}$ to custom state transition systems with private and public transitions, and their current state. A WORLD is simply a pair of W^{std} and W^{cus}, and $stsCollection(W)$ denotes the full ownership (aka the authoritative view) of W.

We can now also define the rules for public and future worlds, which we've already seen above. We call W' a public future world of W, $W' \sqsupseteq^{pub} W$, if each state in W' is either fresh, or publicly reachable from its state in W. A state is publicly reachable by a sequence of public transitions. Conversely, we call W' a private future world of W, $W' \sqsupseteq^{priv} W$, if each state in W' is either fresh, or privately reachable from its state in W. A state is privately reachable by a (possibly interwoven) sequence of private and public transitions.

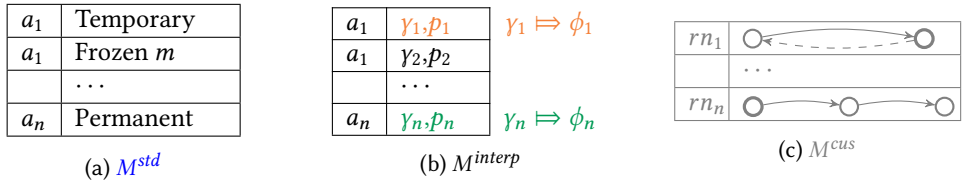
6.6 Linking Worlds to Memory

So far, we've defined worlds, explained how the logical relation depends on the invariants in a world and how worlds are allowed to evolve over time. What is still missing in the story is mapping invariants to requirements on memory contents and ensuring that those requirements are satisfied at runtime. This part of the logical relation is a bit technical and makes use of Iris machinery like stored predicates and certain resource algebras. You may wish to skip it on first reading.

First, in Figure 11, we define the resource interpretation of each standard state: standard shared resource invariants. The role of each interpretation is to map an address to the requirement on the location's contents that the associated invariant in its current state represents.

A *permanent* resource invariant for some address a and permission p contains the ownership of a points-to predicate for the address a. It states that some predicate ϕ holds at the current state of a, say v, and some world W. Crucially, this ϕ holds *invariantly*, that is in any private future world of W (when applied to that same v). On the other hand, local capabilities are allowed to depend on revocable invariants, so a *temporary* invariant only requires ϕ to be monotone with regards to public future worlds and is not required to be able to survive private world updates.

Finally, a *frozen* resource invariant Frozen m is parametrized by a memory segment: a partial map $m : Addr \hookrightarrow Word$ from addresses to specific words. Note that these words do not need to satisfy any invariant ϕ or be themselves safe in any way. The *Frozen* state imposes two requirements for addresses $a \in \text{dom}(m)$: that they point to the associated word, i.e. they cannot change until the



(a) M^{std} (b) M^{interp} (c) M^{cus}

Fig. 12. Abstract Machine.

state changes from *Frozen* to *Temporary*, and that each address in the map m are also frozen to the same map m (see below for more details on M^{std}). This additional requirement ensures that if one of the invariants for an address in $\mathrm{dom}(m)$ is revoked, all other ones must be revoked along and it allows us to think of the addresses in m as being frozen as a block, rather than individually.

So, now we have defined maps that keep track of the standard state of shared resources, the region state of custom resources, and their associated state transition systems. We have also defined the meaning of temporary, permanent, and frozen resource invariants as requirements on memory. What remains is to connect the two: mapping a given world to the requirement on memory that its invariants represent. This connection is made using a few non-trivial pieces of Iris machinery that we cannot explain in detail due to space constraints, so we restrict ourselves to a rough sketch of what is going on.

Technically, this connection is made in the predicates $rel(a, p, \phi)$, $sharedResources(W)$ and $stsCollection(W)$ that appear in Figure 9. These three predicates are all defined as requirements on what we can think of as an *instrumented machine state*. This instrumented machine state consists of three parts, depicted in Figure 12. The most important part is M^{interp}, which associates each address a with a predicate ϕ and a permission p. To simplify the definition of M^{interp}, we do not associate a directly to ϕ, but indirectly through an Iris saved predicate $\gamma_n \Longmapsto \phi_n$, but this is a technical detail that can be ignored. The permission represents the permission of the first allocated capability with authority over a, in other words an upper bound on the permission of all capabilities that contain a in their range of authority. The predicate $\phi : \mathsf{WORLD} \times \mathit{Word} \to \mathit{iProp}$ represents the predicate that is currently enforced on values stored in memory at address a. Without elaborating on its definition, the predicate $rel(a, p, \mathcal{V})$ that was used in Figure 9 requires that p and $\mathcal{V}$ are registered in M^{interp} as the permission and predicate for address a.

Additionally, the instrumented machine state contains two other pieces of logical state M^{std} and M^{cus}, containing an authoritative copy of the current world W and its two parts W^{std} and W^{cus}. Again we won't provide full details about the definition, but the two remaining predicates $stsCollection(W)$ and $sharedResources(W)$ from Figure 9 impose requirements on this authoritative world. Essentially, $stsCollection(W)$ requires that this authoritative copy of the world corresponds exactly to W. Finally, $sharedResources(W)$ makes the connection for every address a between three things: the actual word w in the capability machine's memory at a, the predicate ϕ registered for a in M^{interp} and the standard state S for a in M^{std}. It requires that ϕ satisfies the word w at the world W, in the appropriate way as defined in Figure 11 for the state S.

6.7 Fundamental Theorem

With the definition of our logical relation in place, we can now state the fundamental theorem of our logical relation (FTLR). In broad terms, the FTLR states that if a range $[b, e)$ is safe to read, then it is safe to execute. The permission of the capability must itself be executable, and in particular if the capability is RWLX, then its locality must be LOCAL.

Theorem 6.1 (FTLR). *Assume that* $p = \textsc{rx}$, $p = \textsc{rwx}$ *or* $(p = \textsc{rwlx} \wedge g = \textsc{local})$. *Assume also that* $\mathcal{V}(W)(p, g, b, e, a)$. *Then we have that* $\mathcal{E}(W)(p, g, b, e, a)$.

proof sketch. We prove the FTLR by Löb induction, i.e. by assuming that the theorem holds later (after one step), we prove that it holds now. In order to take a step in the program, we consider the different possible instructions pointed to by the program counter. For each instruction, we look at all the possible cases: for example we need to distinguish between moving a constant into a register and moving from one register to another. If the instruction fails, we are done since we know the postcondition of the expression relation holds for a failed configuration. If the instruction succeeds, we prove safety of the resulting machine state and the updated program counter capability, and apply the induction hypothesis. Each instruction has many cases, especially when one considers all the possible ways an instruction can fail. To avoid a tedious blow-up of case distinctions, we use a general form of the program logic rules that separate the (interesting) success case from all the (uninteresting) possible failure cases. The `store` and `load` instructions also require us to access the memory invariants of the source and destination addresses. This is done using the *sharedResources*(W) predicate, knowing that each address is accessed using a safe capability, which means we know its exact standard type in W. A particularly interesting case is that of the `storeU` instruction: if we store to an uninitialized capability at offset 0, the current address of that capability is increased by one. As a result, if we need to show that the resulting register state is safe, we will need to show that this updated capability is safe. This means we might have to change the state of that address from Frozen to Temporary. Since such a change is public, we can monotonically update the *sharedResources*(W) predicate to *sharedResources*$(W[a := \text{Temporary}])$. □

We use the fundamental theorem whenever we want to reason about unknown adversary code. For instance, if we go back to the third scenario, when Alice returns to Charlie, we can finish the execution simply by knowing that the return capability is safe: if it was an enter capability, we directly apply the execute condition, and if it was an executable capability, we know that its range is safe to read, and thus by the fundamental theorem, safe to execute. Let us see in slightly more detail what kind of properties we can prove about example programs.

6.8 A Concrete Scenario: the "Awkward Example"

We demonstrate the use of our logical relation model by verifying the correctness of a tricky example program, in a scenario where known (verified) code calls to and is called by unknown (possibly adversarial) code. The example is a low-level version of the "awkward example" [Dreyer et al. 2010]:

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

The correctness of this program—the assert never fails—relies on local state encapsulation (the adversary cannot modify private location x) and well-bracketed control flow (the adversary must return to where he was last invoked). Exploiting these properties when they are built into the language is already quite challenging: Dreyer et al. deploy a step-indexed Kripke logical relation with public and private transitions to achieve that task. In subsequent work, Skorstengaard et al. [2018] verify (on paper) a low-level version of this example adapted to a capability machine with local capabilities. In that setting, local state encapsulation and well-bracketed control flow are not properties of the language (they do not make sense at the machine-code level), but are instead consequences of the secure calling convention implemented in the example.

In this work, we adapt the machine-code example from Skorstengaard et al. [2018] to use our improved calling convention with uninitialized capabilities, and prove the updated code correct using Iris and our logical relation mechanized in Coq. Our code appears in Figure 13, with differences

```
g1: malloc r2 1
    store r2 0
    move r3 pc
    lea r3 offset
    crtcls [(x, r2)] r3
    jmp r0
f1: reqglob r_adv
    (continues in next column)
```

```
(continued from previous column)
prepstack r_stk
store r_env 0
scallU r_adv ([], [r0, r_adv, r_env])
store r_env 1
scallU r_adv ([], [r0, r_env])
load r_adv r_env
assert r_adv 1
(continues in next column)
```

```
(continued from previous column)
getb r1 r_stk
add r2 r1 10
subseg r_stk r1 r2
mclear r_stk
rclear RegName\{pc, r0}
jmp r0
```

Fig. 13. The awkward example using our new calling convention. It relies on local state encapsulation and well-bracketedness as provided by `scall`. `g1` is the entry-point of the program; when executed, it creates a closure (as an E capability) whose body executes `f1`. *offset* is the offset to `f1`. Changes following our new calling convention are highlighted in blue.

highlighted in blue. There are two main changes: first, secure function calls are made through a new `scallU` macro that implements the stack discipline described in Section 4.2; second, we now only clear our own stack frame before returning to the adversary instead of the whole stack (here, this means clearing ten memory cells instead of possibly thousands or millions).

We carry out the proof in two main steps. In a first step (Lemma 6.2), we show that the program entry-point `g1` is safe according to the expression relation $\mathcal{E}$.

Lemma 6.2. *For any world W, assuming that the memory has been properly initialized*[3] *in region $[b_{\text{awk}}, e_{\text{awk}})$ with the code of the program and a pointer to the* `malloc` *and* `assert` *subroutines, we have:*

$$\mathcal{E}(W)(\mathit{RX}, \mathit{GLOBAL}, b_{\text{awk}}, e_{\text{awk}}, \texttt{g1}).$$

The bulk of the work consists in proving this lemma: the proof requires allocating a custom state transition system for the encapsulated reference, stepping through the code of the program using the program logic rules, and using the FTLR (Theorem 6.1) to reason about calls to unknown code (made by `scallU` and the final `jmp` to an unknown return pointer).

In a second step (Theorem 6.3), we use the standard adequacy theorem of Iris, and derive a closed statement for the correctness of our program against the operational semantics of the machine[2].

Theorem 6.3. *(Correctness of the awkward example) Let $\mathit{reg} \in \text{Reg}$, $m \in \text{Mem}$ and*

$$c_{\text{awk}} \triangleq (\mathit{RX}, \mathit{GLOBAL}, \ldots) \quad c_{\text{stk}} \triangleq (\mathit{URWLX}, \mathit{LOCAL}, \ldots) \quad c_{\text{adv}} \triangleq (\mathit{RWX}, \mathit{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer[3]*. Furthermore, assume that:*

- *m has been initialized with the code of the program and subroutines (pointed to by c_{awk}), an uninitialized stack (pointed to by c_{stk}), and unknown adversarial code (pointed to by c_{adv});*
- *$\mathit{reg}(pc) = c_{\text{awk}}$, $\mathit{reg}(r_{\text{stk}}) = c_{\text{stk}}$, $\mathit{reg}(r_0) = c_{\text{adv}}$ and $\mathit{reg}(r) \in \mathbb{Z}$ otherwise;*
- flag *denotes the memory address set to* 1 *by the* `assert` *subroutine in case of failure;*
- $m(\text{flag}) = 0$.

If $(\text{Repeat SingleStep}, (\mathit{reg}, m)) \rightarrow^* (\mu, (\mathit{reg}', m'))$ *then* $m'(\text{flag}) = 0$.

Theorem 6.3 states that, starting from a properly initialized machine state, the in-memory flag set by the `assert` routine remains set to 0 at every step of the execution—meaning that the call to `assert` never fails. Obtaining Theorem 6.3 from Lemma 6.2 is mostly mechanical: this highlights

[2]We have also instantiated Theorem 6.3 with a simple adversarial code that invokes the awkward example with $f = (\lambda().\ ())$ and additionally proved that, in that setting, the whole machine runs and gracefully halts.

one of the benefits of using Iris, whose built-in soundness theorem can be leveraged to obtain a program specification stated directly against the operational semantics of the machine.

7 IMPLEMENTATION

We have implemented uninitialized capabilities in the CHERI-MIPS ISA for the 256-bit capability format (we believe that the implementation should be possible for other capability formats as well[4]). In CHERI-MIPS the stack grows downwards (from higher memory adresses to lower memory addresses) and the implementation of uninitialized capabilities is inverted to reflect the stack growth. Concretely, uninitialized capabilities only allow reading from the range $[a, e]$ and a moves downwards on writes below the current a, just like the stack. Capabilities now have a bit indicating if they are uninitialized or not. Some existing CHERI-MIPS instructions are modified to take the uninitialized permission into account: the load instructions and those that modify a capability's cursor. For experimentation purposes, we have opted to add separate store instructions for uninitialized capabilities, leaving the old store instructions intact. Additionally, we add an instruction to make a regular capability uninitialized and a new variant of `CSetBounds` (the CHERI version of `subseg`) that is needed for technical reasons.

These modifications result in a CHERI-MIPS simulator that supports uninitialized capabilities. We have also added support for the new instructions to the Clang/LLVM assembler for CHERI-MIPS[4]. This allows us to write assembly programs with the new instructions and run them on the simulator. With the simulator and assembler in place, we were able to experiment with the new calling convention by manually modifying assembly programs. The calling convention of Section 4.2 is slightly modified for CHERI-MIPS because CHERI uses pairs of sealed capabilities (a code capability and data capability) instead of enter capabilities. This means we do not need to store return closures on the stack (like for `StkTokens` [Skorstengaard et al. 2019b]), but otherwise makes little difference.

Although more investigation is needed, our results suggest that uninitialized capabilities and the calling convention from Section 4.2 can be adapted and applied in a CHERI setting.

8 RELATED WORK

We already discussed some related work in the introduction, which we briefly recall now. We follow an existing line of work on capability machines [Carter et al. 1994; Levy 1984; Watson et al. 2019, 2015], and in particular the CHERI family featuring local capabilities [Watson et al. 2019, 2015] that provide a form of revocable capabilities. To our knowledge, uninitialized capabilities and the idea of using them to reduce the cost of local capability revocation are both new.

Other forms of revocation have been proposed in capability machine contexts. A line of work of the CHERI project (CHERI-JNI [Chisnall et al. 2017], CHERIvoke [Xia et al. 2019], Cornucopia [Filardo et al. 2020]) presents a general revocation mechanism for memory managed through a dedicated memory allocator. In that setting, revocation happens by sweeping through the whole memory and clearing obsolete (revoked) pointers. This GC-like approach to revocation is somewhat orthogonal to our stack-based revocation mechanism. The authors mostly focus on practical feasibility, and do not formally state or prove the guarantees provided by their revocation procedure.

Linear capabilities [Watson et al. 2019] have also been proposed as a lightweight revocation mechanism, both for implementing a secure calling convention [Skorstengaard et al. 2019b] providing similar guarantees as ours, and as a secure compilation target for separation logic verified

[3] These assumptions are kept intentionally vague for brevity. Full statements can be found in the Coq formalization.
[4] Available at https://zenodo.org/record/4067949.

code [Van Strydonck et al. 2019]. However, there are concerns as to whether the atomic store-and-clear operation required by linear capabilities can be realistically implemented in hardware without an important performance penalty and whether they would be easy to support in existing compilers [Skorstengaard 2019, §3.6.2]. We expect uninitialized capabilities to be a more benign extension from a micro-architectural and compiler perspective.

Another category of related work is on formalizing capability safety, i.e. characterizing the guarantees provided by a capability machine or language runtime. In the context of high-level languages with object capabilities, Maffeis et al. [2010] define a syntactic notion of capability safety based on reachability between objects. This kind of criterion is however of limited expressive power as it is not directly defined with respect to the actual behaviour of objects. Drossopoulou et al. [2020] formalize a form of capability safety in their Chainmail specification language. It can be used to capture properties of object-oriented programs like "An account's balance can be changed only if a client has access to that particular account".

More closely to our current work, Devriese et al. [2016] propose a more expressive, semantic definition of capability-safety for object capabilities, based on a Kripke logical relation with public and private transitions which is not unlike ours. Swasey et al. [2017] extend this line of work by showing that a similar logical relation can be used to give compositional specifications for the robustness of object capabilities patterns, and formalize their work in Coq using Iris.

Other related work has considered capability safety of (low-level) capability machines. Nienhuis et al. [2020] build a formal model of the CHERI ISA, and formally verify a number of architectural security properties using Isabelle/HOL. A key security property they prove is capability monotonicity, meaning that the machine does not allow creating new capabilities out of thin air, and therefore, that an unknown code component can only modify parts of memory it has access to through its reachable capabilities. This is a somewhat syntactic property in nature, and it has an important limitation: it only holds until the code jumps to an enter capability (or sealed capability in the case of CHERI), which purposely gives access to new capabilities in a non-monotonic way. Therefore, their security properties only hold within a single "component". Our definition of capability safety, although more involved, allows reasoning about a complete machine execution, with arbitrary calls between different security domains and dynamic evolution of invariants and boundaries. Akram El-Korashy [2016] has studied a formal model of the CHERI capability machine and proved some properties of it. Their main capability safety property captures a whole-system form of capability monotonicity that appears unsuitable for reasoning.

The work by Skorstengaard et al. [2018, 2019a] is probably the most closely related to our own. As discussed before, they define capability safety for a capability machine with local capabilities as a logical relation, and propose a secure calling convention based on local capabilities. Our contributions are the introduction of a more efficient calling convention using uninitialized capabilities and a more expressive model (with the introduction of Frozen regions), as well as our formalization of our work in Coq using Iris. In subsequent work, Skorstengaard et al. [2019b] verify a secure calling convention based on linear capabilities. They phrase their result as a fully-abstract compilation theorem, rather than by verifying challenging examples, as they did in their previous work, and as we do here. This is an interesting perspective for future work: we believe that we could alternatively prove a similar theorem to characterize the correctness of our secure calling convention.

There are a number of previous work on using logical relations with public/private transitions to account for well-bracketed state changes [Devriese et al. 2016; Dreyer et al. 2010; Skorstengaard et al. 2018], as well as using Iris to mechanize logical relations using higher-level constructs [Giarrusso et al. 2020; Timany and Birkedal 2019; Timany et al. 2017]. Our combination of the two is novel: we use lightweight Kripke worlds and Iris saved predicates to allow for precisely tracking the

relationship between intermediate logical states (which would be impossible using Iris invariants), but we avoid solving recursive domain equations or working explicitly with step indices.

A final category of related work is on program logics for low-level code. Our program logic deals with code stored in memory as data, uses continuations to specify sequences of instructions, in combination with step-indexing to deal with unstructured control flow, and uses separation logic to model the resources associated to registers and memory. These features can often be found in a number of previous works [Cai et al. 2007; Chlipala 2011; Jensen et al. 2013; Myreen and Gordon 2007; Ni and Shao 2006]. The distinguishing feature of our program logic is that it is built on top of an existing general purpose logic. Consequently, we can (and we do) exploit the powerful features of Iris to reason about the low-level programs that we consider.

9 CONCLUSION

Local capabilities potentially provide an efficient but restricted revocation primitive in capability machines, with many possible applications. We have demonstrated how uninitialized capabilities can make them actually live up to this potential by solving an important performance problem. Moreover, using our novel formalized model of capability safety, we have demonstrated that the combination of local and uninitialized capabilities lends itself to machine-checked reasoning. In particular, we have verified an implementation of a classical example from the literature, which makes advanced use of local capability revocation through our modified calling convention. The example is, by the way, longer than it looks (400 instructions after unfolding macros). This shows the power of local capability revocation using uninitialized capabilities as well as the expressiveness of our reasoning infrastructure. Finally, our initial results suggest that uninitialized capabilities and our new calling convention can be practically applied in a more realistic setting like CHERI. We believe these different results combined make a strong case for the addition of uninitialized capabilities in CHERI and other capability machines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for valuable comments and suggestions. 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) under grant number G0G0519N; 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 a postdoctoral fellow of the Flemish research fund (FWO) during parts of this project.

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. https://doi.org/10.1145/1480881.1480925

Akram El-Korashy. 2016. *A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI.* Master Thesis. Saarland University.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Jon French, Kathryn E. Gray, Gabriel Kerneis, Neel Krishnaswami, Prashanth Mundkur, Robert Norton-Wright, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. 2013–2019. The Sail Instruction-Set Architecture (ISA) specification language.

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Logic. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf. (2017).

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed kripke models over recursive worlds. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, Thomas Ball and Mooly Sagiv (Eds.). ACM, 119–132. https://doi.org/10.1145/1926385.1926401

Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified Self-Modifying Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/1250734.1250743

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

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 234–245. https://doi.org/10.1145/1993498.1993526

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 with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 147–162. https://doi.org/10.1109/EuroSP.2016.22

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. https://doi.org/10.1145/1863543.1863566

Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Heike Wehrheim and Jordi Cabot (Eds.), Vol. 12076. Springer, 420–440. https://doi.org/10.1007/978-3-030-45234-6_21

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.

Paolo Giarrusso, Leo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step — Soudness for DOT with Step-indexed Logical Relations in Iris. *Proc. ACM Program. Lang.* ICFP (2020).

Sander Huyghebaert. 2020. *A Secure Calling Convention with Uninitialized Capabilities*. Master's thesis. Vrije Universiteit Brussel. https://doi.org/10.5281/zenodo.4073111

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-Level Separation Logic for Low-Level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/2429069.2429105

A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. 2017. Efficient Tagged Memory. In *IEEE International Conference on Computer Design (ICCD)*. IEEE. https://doi.org/10.1109/ICCD.2017.112

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

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

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

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Principles of Programming Languages (POPL)*.

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

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 125–140. https://doi.org/10.1109/SP.2010.16

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University.

Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare Logic for Realistically Modelled Machine Code. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 568–582.

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. *SIGPLAN Not.* 41, 1 (Jan. 2006), 320–333. https://doi.org/10.1145/1111320.1111066

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)*.

Lau Skorstengaard. 2019. *Formal Reasoning about Capability Machines*. Ph.D. Dissertation. Aarhus University.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 475–501. https://doi.org/10.1007/978-3-319-89884-1_17

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, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

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

Amin Timany and Lars Birkedal. 2019. Mechanized Relational Verification of Concurrent Programs with Continuations. *Proc. ACM Program. Lang.* 3, ICFP, Article 105 (July 2019), 28 pages. https://doi.org/10.1145/3341709

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

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).

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. 2019. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7)*. Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html

R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (Sept. 2016), 38–49. https://doi.org/10.1109/MM.2016.84

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

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