Reasoning About a Machine with Local Capabilities

Provably Safe Stack and Return Pointer Management

LAU SKORSTENGAARD, Aarhus University, Denmark
DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

Capability machines provide security guarantees at machine level which makes them an interesting target for secure compilation schemes that provably enforce properties such as control-flow correctness and encapsulation of local state. We provide a formalization of a representative capability machine with local capabilities and study a novel calling convention. We provide a logical relation that semantically captures the guarantees provided by the hardware (a form of capability safety) and use it to prove control-flow correctness and encapsulation of local state. The logical relation is not specific to our calling convention and can be used to reason about arbitrary programs.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: capability machines, CHERI, local capabilities, well-bracketed control flow, stack encapsulation, logical relations, secure compilation

ACM Reference Format:
Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. Reasoning About a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (2019), 53 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Compromising software security is often based on attacks that break programming language properties relied upon by software authors such as control-flow correctness, local-state encapsulation, etc. Commodity processors offer little support for defending against such attacks: they offer security primitives with only coarse-grained memory protection and limited compartmentalization scalability. As a result, defenses against attacks on control-flow correctness and local-state encapsulation are either limited to mitigation of only certain common forms of attacks (leading to an attack-defense arms race [Szekeres et al. 2013]) and/or rely on techniques like machine code rewriting [Abadi et al. 2005; Wahbe et al. 1993], machine code verification [Morrisett et al. 1999], virtual machines with a native stack [Lindholm et al. 2014] or randomization [Forrest et al. 1997]. The latter techniques essentially emulate protection techniques on existing hardware at the cost of performance, system complexity, and/or security.

Authors' addresses: Lau Skorstengaard, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark, lau@cs.au.dk; Dominique Devriese, Vakgroep Computerwetenschappen, Vrije Universiteit Brussel, Pleinlaan 2, Elsene, 1050, Belgium, dominique.devriese@vub.be; Lars Birkedal, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark, birkedal@cs.au.dk.

0164-0925/2019/0-ART0 $15.00
https://doi.org/0000001.0000001

Capability machines are a type of processors that remediate these limitations with a better security model at the hardware level. They are based on old ideas [Carter et al. 1994; Dennis and Van Horn 1966; Shapiro et al. 1999] that have recently received renewed interest. In particular, the CHERI project has proposed new ideas and ways of tackling practical challenges like backwards compatibility and realistic OS support [N. M. Watson et al. 2015; Woodruff et al. 2014]. Capability machines tag every word (in the register file and in memory) to enforce a strict separation between numbers and capabilities (a kind of pointers that carry authority). Memory capabilities carry the authority to read and/or write to a range of memory locations. There is also a form of *object capabilities*, which represent the authority to invoke a piece of code without exposing the code's encapsulated private state (e.g., the M-Machine's enter capabilities (described in Section 2) or CHERI's sealed code/data pairs).

Unlike commodity processors, capability machines lend themselves well to enforcing local-state encapsulation. Potentially, they will enable compilation schemes that enforce this property in an efficient but also 100% watertight way (ideally evidenced by a mathematical proof, guaranteeing that we do not end up in a new attack-defense arms race). However, a lot needs to happen before we get there. For example, it is far from trivial to devise a compilation scheme adapted to the details of a specific source language's notion of encapsulation (e.g., private member variables in OO languages often behave quite differently than private state in ML-like languages). And even if such a scheme were defined, a formal proof depends on a formalization of the encapsulation provided by the capability machine at hand.

A similar problem is the enforcement of control-flow correctness on capability machines. An interesting approach is taken in CheriBSD [N. M. Watson et al. 2015]: the standard contiguous C stack is split into a central, trusted stack and disjoint, private, per-compartment stacks. The trusted stack is managed by trusted call and return instructions. To prevent illegal use of stack references, the approach relies on *local capabilities*, a type of capabilities offered by CHERI to *temporarily* relinquish authority, namely for the duration of a function invocation whereafter the capability can be revoked. However, details are scarce (how does it work precisely? what features are supported?) and a lot remains to be investigated (e.g., combining disjoint stacks with cross-domain function pointers seems like it will scale poorly to large numbers of components). Finally, there is no argument that the approach is watertight and it is not even clear what security property is targeted exactly.

In this paper, we make two main contributions: (1) an alternative calling convention that uses local capabilities to enforce stack frame encapsulation and well-bracketed control flow, and (2) perhaps more importantly, we adapt and apply the well-studied techniques of step-indexed Kripke logical relations for reasoning about code on a representative capability machine with local capabilities in general and correctness and security of the calling convention in particular. More specifically, we make the following contributions:

- We formalize a simple but representative capability machine featuring local capabilities and its operational semantics (Section 2).
- We define a novel calling convention enforcing control-flow correctness and encapsulation of stack frames (Section 3). It relies solely on local capabilities and does not require OS support (like a trusted stack or call/return instructions). It supports higher-order cross-component calls (e.g., cross-component function pointers) and can be efficient assuming only one additional piece of processor support (w.r.t. CHERI): an efficient instruction for clearing a range of memory.
- We present a novel step-indexed Kripke logical relation for reasoning about programs on the capability machine. It is an untyped logical relation, inspired by previous work on object

capabilities [Devriese et al. 2016]. We prove an analogue of the standard fundamental theorem of logical relations — to the best of our knowledge, our theorem is the most general and powerful formulation of the formal guarantees offered by a capability machine (a form of capability safety [Devriese et al. 2016; Maffeis et al. 2010]), including the specific guarantees offered for local capabilities. It is very general and not tied to our calling convention or a specific way of using the system's capabilities. We are the first to apply these techniques for reasoning about capability machines and we believe they will prove useful for many other purposes than our calling convention.

- We introduce two novel technical ideas in the unary, step-indexed Kripke logical relation used to formulate the above theorem: the use of a single orthogonal closure (rather than the earlier used biorthogonal closure) and a variant of Dreyer et al. [2012]'s public and private future worlds [Dreyer et al. 2012] to express the special nature of local capabilities. The logical relation and the fundamental theorem expressing capability safety are presented in Section 4.
- We demonstrate our results by applying them to challenging examples, specifically constructed to demonstrate local-state encapsulation and control-flow correctness guarantees in the presence of cross-component function pointers (Section 8). The examples demonstrate both the power of our formulation of capability safety and our calling convention.

This paper is an extension of the published conference paper Skorstengaard et al. [2018]. We have made improvements to readability and completeness throughout the paper. For instance, we have added an introduction to Section Logical Relation that provides informal intuition about how the logical relation machinery comes into play on a capability machine. We highlight the following changes:

- We have added proof sketches for the Fundamental Theorem and for the correctness lemma for the awkward example (Lemma 8.4).
- We have added figures that illustrate central parts of the calling convention.
- A section about malloc has been added. Specifically, we provide the specification for the malloc used in the examples of the paper.
- A section about macro instructions with descriptions of all the macros and the implementation of `scall` has been added.
- We have added a section on reasoning about programs that run on a capability machine. This section explains how one reasons about common scenarios that arise in programs on a capability machine, and, in particular, how the logical relation is used. It also introduces a number of lemmas that prove recurring bits once and for all.
- We have expanded on the explanation of the awkward example.
- Details previously found only in the technical appendix [Skorstengaard et al. 2019a] have been moved to the paper.

We have written a technical appendix [Skorstengaard et al. 2019a] which contains additional details and proofs left out from this paper.

2 A CAPABILITY MACHINE WITH LOCAL CAPABILITIES

In this paper, we work with a formalization of a capability machine with all the characteristics of real capability machines as well as local capabilities much like CHERI's. Otherwise, it is kept as simple as possible. It is inspired by both the M-Machine [Carter et al. 1994] and CHERI [N. M. Watson et al. 2015]. For simplicity, we assume an infinite address space and unbounded integers (see Section 9 for a discussion of these assumptions).

We define the syntax of our capability machine in Figure 2. We assume an infinite set of addresses Addr and define machine words as either integers or capabilities of the form $((perm, g), base, end, a)$. Such a capability represents the authority to execute permissions *perm* on the memory range $[base, end]$, together with a current address a and a locality tag g indicating whether the capability is global or local. On a capability machine, there is no notion of pointers other than capabilities, so we will use the terms interchangeably. The available permissions are null permission (O), readonly (RO), read/write (RW), read/execute (RX), and read/write/execute (RWX) permissions. Additionally, there are three special permissions: read/write-local (RWL), read/write-local/execute (RWLX) and enter (E), which we will explain below. The orderings of permissions and locality are displayed in Figure 1. The write permission is subsumed by the write-local permission, i.e. a write-local capability can be used to store both local and global capabilities to memory which is why write-local is above write in the permission hierarchy. We denote the pairwise ordering of permission and locality with $\sqsubseteq$.

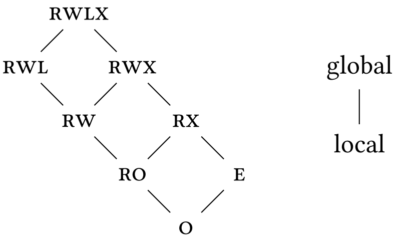


Fig. 1. Permission and locality hierarchy.

We assume a finite set of register names RegName. We define register files *reg* and memories *ms* as functions mapping register names resp. addresses to words. The state of the entire machine is represented as a configuration that is either a running state $\Phi \in$ ExecConf containing a memory and a register file or a failed or halted state where the latter is paired with the final state of memory.

The machine's instruction set is rather basic. Instructions *i* include relatively standard jump (`jmp`), conditional jump (`jnz`), and move (`move`, copies words between registers) instructions. Also familiar are load and store instructions for reading from and writing to memory (`load` and `store`) and arithmetic operators (`lt` (less than), `plus` and `minus`, operating only on numbers). There are three instructions for modifying capabilities: `lea` (modifies the current address), `restrict` (modifies the permission and local/global tag), and `subseg` (modifies the range of a capability). Importantly, these instructions take care that the resulting capability always carries less authority than the original (e.g. `restrict` will only weaken a permission according to the hierarchy in Figure 1). Finally, the instruction `isptr` tests whether a word is a capability or a number and instructions `getp`, `getl`, `getb`, `gete` and `geta` provide access to a capability's permissions, local/global tag, base, end and current address, respectively.

Figure 3 shows the operational semantics for a few instructions representative of the machine. Essentially, a configuration Φ either decodes and executes the instruction pointed to by Φ.reg(pc) if it is an executable capability with its address in the valid range; otherwise it fails. The table in the figure shows for instructions *i* the result of executing them in configuration Φ. The instructions `fail` and `halt` obviously fail and halt respectively. `move` simply modifies the register file as requested and updates the pc to the next instruction using the meta-function *updPc*.

The `load` instruction loads the contents of the requested memory location into a register, but only if the capability has appropriate authority (i.e. read permission and an appropriate range). The `restrict` instruction updates a capability's permissions and global/local tag in the register file, but only if the new permissions are weaker than the original according to the permission hierarchy in Figure 1. The `subseg` instruction reduces the range of authority of a capability. In order to represent the unbounded address space, we use -42 to represent infinity.

The `jmp` instruction updates the program counter to a requested location, but it is complicated by the presence of enter capabilities after the M-Machine's [Carter et al. 1994]. Enter capabilities cannot be used to read, write or execute and their address and range cannot be modified. They can only be used to jump to. When that happens, their permission changes to RX. They can be used to represent a kind of closures: an opaque package containing a piece of code together with local encapsulated state. Such a package can be built as an enter capability $c = ((\textsc{e}, g), b, e, a)$ where the range $[b, a-1]$ contains local state (data or capabilities) and $[a, e]$ contains instructions. The package is opaque to an adversary holding c. When c is jumped to however, the instructions can start executing and have access to the local data through the updated version of c, now in the pc-register.

The instruction `lea` manipulates the current address of non enter-capabilities. It is fine for a capability to have a current address outside its range of authority as long as it is not used with an instruction that requires a specific capability. The instruction `geta` queries the current address of a capability and stores it in a register.

Finally, the `store` instruction updates the memory to the argument value if the capability has write authority for the specified location. However, the instruction is complicated by the presence of local capabilities modeled after the ones in the CHERI processor [N. M. Watson et al. 2015]. At a high-level, local capabilities are special in that they can only be kept in registers, i.e. they cannot be stored to memory. This means that local capabilities can be temporarily given to an adversary, for the duration of an invocation. If we make sure to clear the capability from the register file after control is passed back to us, the adversary is unable to store the capability. However, there is one exception to the rule above: local capabilities can be stored to memory for which we have a capability with write-local authority (i.e. permission RWL or RWLX). This is intended to accommodate a stack where register contents can be stored, including local capabilities. As long as all capabilities with write-local authority are themselves local and the stack is cleared after control is passed back by the adversary, we will see that this does not break the intended behavior of local capabilities.

$$
\begin{array}{rrcl}
a \in & \text{Addr} & \overset{\text{def}}{=} & \mathbb{N} \\
perm \in & \text{Perm} & ::= & \textsc{o} \mid \textsc{ro} \mid \textsc{rw} \mid \textsc{rwl} \mid \textsc{rx} \mid \textsc{e} \mid \textsc{rwx} \mid \textsc{rwlx} \\
g \in & \text{Global} & ::= & \text{global} \mid \text{local} \\
c \in & \text{Cap} & \overset{\text{def}}{=} & \{((perm, g), b, e, a) \mid b, a \in \text{Addr}, e \in \text{Addr} \cup \{\infty\}\} \\
w \in & \text{Word} & \overset{\text{def}}{=} & \mathbb{Z} + \text{Cap} \\
r \in & \text{RegName} & ::= & \text{pc} \mid r_0 \mid r_1 \mid \ldots \\
reg \in & \text{Reg} & \overset{\text{def}}{=} & \text{RegName} \rightarrow \text{Word} \\
m \in & \text{Mem} & \overset{\text{def}}{=} & \text{Addr} \rightarrow \text{Word} \\
\Phi \in & \text{ExecConf} & \overset{\text{def}}{=} & \text{Reg} \times \text{Mem} \\
ms \in & \text{MemSeg} & \overset{\text{def}}{=} & \text{Addr} \rightharpoonup \text{Word} \\
 & \text{Conf} & \overset{\text{def}}{=} & \text{ExecConf} + \{\mathit{failed}\} + \{\mathit{halted}\} \times \text{Mem}
\end{array}
$$

$$
\begin{array}{rcl}
r & \in & \mathbb{Z} + \text{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ r \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{lt}\ r\ r\ r \mid \texttt{plus}\ r\ r\ r \mid \texttt{minus}\ r\ r\ r \mid \\
 & & \texttt{lea}\ r\ r \mid \texttt{restrict}\ r\ r \mid \texttt{subseg}\ r\ r\ r \mid \texttt{isptr}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getb}\ r\ r \mid \\
 & & \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Fig. 2. The syntax of our capability machine assembly language.

$$\Phi \rightarrow \begin{cases} [\![\mathit{decode}(n)]\!]\,(\Phi) & \text{if } \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), b, e, a) \text{ and } b \leq a \leq e \\ & \quad \text{and } \mathit{perm} \in \{\textsc{rx}, \textsc{rwx}, \textsc{rwlx}\} \text{ and } \Phi.\mathrm{mem}(a) = n \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$\mathit{updPc}(\Phi) = \begin{cases} \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \mathit{newPc}] & \text{if } \Phi.\mathrm{reg}(\mathrm{pc}) = ((\mathit{perm}, g), b, e, a) \\ & \quad \text{and } \mathit{newPc} = ((\mathit{perm}, g), b, e, a + 1) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

i	$[\![i]\!]\,(\Phi)$	Conditions
`fail`	*failed*	
`halt`	$(\mathit{halted}, \Phi.\mathrm{mem})$	
`move` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$r_2 \in \mathrm{Reg} \Rightarrow w = \Phi.\mathrm{reg}(r_2)$ and $r_2 \in \mathbb{Z} \Rightarrow w = r_2$
`load` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$\Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $w = \Phi.\mathrm{mem}(a)$ and $b \leq a \leq e$ and $\mathit{perm} \in \{\textsc{rwx}, \textsc{rwlx}, \textsc{rx}, \textsc{rw}, \textsc{rwl}, \textsc{ro}\}$
`restrict` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$\Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $(\mathit{perm}', g') = \mathit{decodePermPair}(\Phi.\mathrm{reg}(r_2))$ and $(\mathit{perm}', g') \sqsubseteq (\mathit{perm}, g)$ and $w = ((\mathit{perm}', g'), b, e, a)$
`subseg` r_1 r_2 r_3	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto w])$	$\Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), b, e, a)$ and for $i \in \{2, 3\}$ $n_i = \Phi.\mathrm{reg}(r_i)$ and $n_2 \in \mathbb{N}$ and $b \leq n_2$ and $n_3 \leq e$ where either $n_3 \in \mathbb{N}$ or ($n_3 = -42$ and $e = \infty$) and $\mathit{perm} \neq \textsc{e}$ and $w = ((\mathit{perm}, g), n_2, n_3, a)$
`jmp` r	$\Phi[\mathrm{reg}.\mathrm{pc} \mapsto \mathit{newPc}]$	if $\Phi.\mathrm{reg}(r) = ((\textsc{e}, g), b, e, a)$, then $\mathit{newPc} = ((\textsc{rx}, g), b, e, a)$ otherwise $\mathit{newPc} = \Phi.\mathrm{reg}(r)$
`lea` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto c])$	$\Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), b, e, a)$ and $n = \Phi.\mathrm{reg}(r_2)$ and $n \in \mathbb{Z}$ and $\mathit{perm} \neq \textsc{e}$ and $c = ((\mathit{perm}, g), b, e, a + n)$
`geta` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{reg}.r_1 \mapsto a])$	$\Phi.\mathrm{reg}(r_2) = ((\_, \_), \_, \_, a)$
`store` r_1 r_2	$\mathit{updPc}(\Phi[\mathrm{mem}.a \mapsto w])$	$\Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), b, e, a)$ and $\mathit{perm} \in \{\textsc{rwx}, \textsc{rwlx}, \textsc{rw}, \textsc{rwl}\}$ and $b \leq a \leq e$ and $w = \Phi.\mathrm{reg}(r_2)$ and if $w = ((\_, \mathrm{local}), \_, \_, \_)$, then $\mathit{perm} \in \{\textsc{rwlx}, \textsc{rwl}\}$
	…	
_	*failed*	otherwise

Fig. 3. An excerpt from the operational semantics.

We point out that our local capabilities capture only a part of the semantics of local capabilities in CHERI. Specifically in addition to the above, CHERI's default implementation of the CCall exception handler forbids local capabilities from being passed across module boundaries. Such a restriction fundamentally breaks our calling convention since we pass around local return pointers and stack capabilities. However, CHERI's CCall is not implemented in hardware but in software precisely to allow experimenting with alternative models like ours.

In order to have a reasonably realistic system, we use a simple model of linking where a program has access to a linking table that contains capabilities for other programs. We also assume malloc to be part of the trusted computing base satisfying a certain specification. Malloc and linking tables are described further in the next section. The specification of malloc is presented in Section 5 as it uses the semantic model we build in Section 4. For full details on the linking table, we refer to the technical appendix [Skorstengaard et al. 2019a].

3 STACK AND RETURN POINTER MANAGEMENT USING LOCAL CAPABILITIES

One of the contributions in this paper is a demonstration that local capabilities on a capability machine support a calling convention that enforces control-flow correctness in a way that is provably watertight, potentially efficient, does not rely on a trusted central stack manager, and supports higher-order interfaces to an adversary, where an adversary is just some unknown piece of code. In this section, we explain the high-level approach of this calling convention. We motivate each security measures with a situation we want to avoid (motivating each measure separately with a summary table at the end). After that, we define a number of reusable macro-instructions that can be used to conveniently apply the proposed convention in subsequent examples.

The basic idea of our approach is simple: we stick to a single, rather standard, C stack and register-passed stack and return pointers much like a standard C calling convention. However, to prevent various ways of misusing this basic scheme, we put local capabilities to work and take a number of not-always-obvious safety measures. The safety measures are presented in terms of what we need to do to protect ourselves against an adversary, but this is only for presentation purposes as our code assumes no special status on the machine. In fact, an adversary can apply the same safety measures to protect themselves against us. In the following paragraphs, we will explain the issues to be considered in all the relevant situations: when (1) starting our program, (2) returning to the adversary, (3) invoking the adversary, (4) returning from the adversary, (5) invoking an adversary callback, and (6) having a callback invoked by the adversary.

Program start-up We assume that the language runtime initializes the memory as follows: a contiguous array of memory is reserved for the stack, for which we receive a stack pointer in the register r_{stk}. We stress that the stack is not built-in, but merely an abstraction we put on this piece of the memory. The stack pointer is local and has RWLX permission. Note that this means that we will be placing and executing instructions on the stack. Crucially, the stack is the only part of memory for which the runtime (including malloc, loading, linking) will ever provide RWLX or RWL capabilities. Additionally, our examples typically also assume some memory to store instructions or static data. Another part of memory (called the heap) is initially governed by malloc and at program start-up, no other code has capabilities for this memory. Malloc hands out RWX capabilities for allocated regions as requested (no RWLX or RWL permissions). For simplicity, we assume that memory allocated through malloc cannot be freed.[1]

Returning to the adversary Perhaps the simplest situation is returning to the adversary after they invoked our code (Figure 4a). In this case, we have received a return pointer from them, and we just need to jump to it as usual. An obvious security measure to take care of is properly clearing the non-return-value registers before we jump (since they may contain data or capabilities that the adversary should not get access to). Additionally, we may have used the stack for various purposes (register spilling, storing local state when invoking other functions etc.), so we also need to clear that data before returning to the adversary (Figure 4b and Figure 4c).

However, if we are returning from a function that has itself invoked adversary code (Figure 4d), then clearing the used part of the stack is not enough. The unused part of the stack may also contain data and capabilities, left there by the adversary, including local capabilities since the stack is write-local. As we will see later, we rely on the fact that the adversary cannot keep hold of local

[1] In more realistic settings, reusing freed memory on a capability machine can be made safe by checking or enforcing that there are no dangling pointers to the freed memory, as implemented, for example, in CHERI-JNI [Chisnall et al. 2017].

capabilities when they pass control to the trusted code and receive control back. In this case, the adversary could use the unused part of the stack to store local pointers and load them from there after they get control back (Figure 4e). To prevent this, we need to clear (i.e. overwrite with zeros) the entire part of the stack that the adversary has had access to; not just the parts that we have used ourselves (Figure 4f). Since we may be talking about a large part of memory, this requirement is the most problematic aspect of our calling convention for performance (see the discussion in Section 9 for how this might be mitigated).

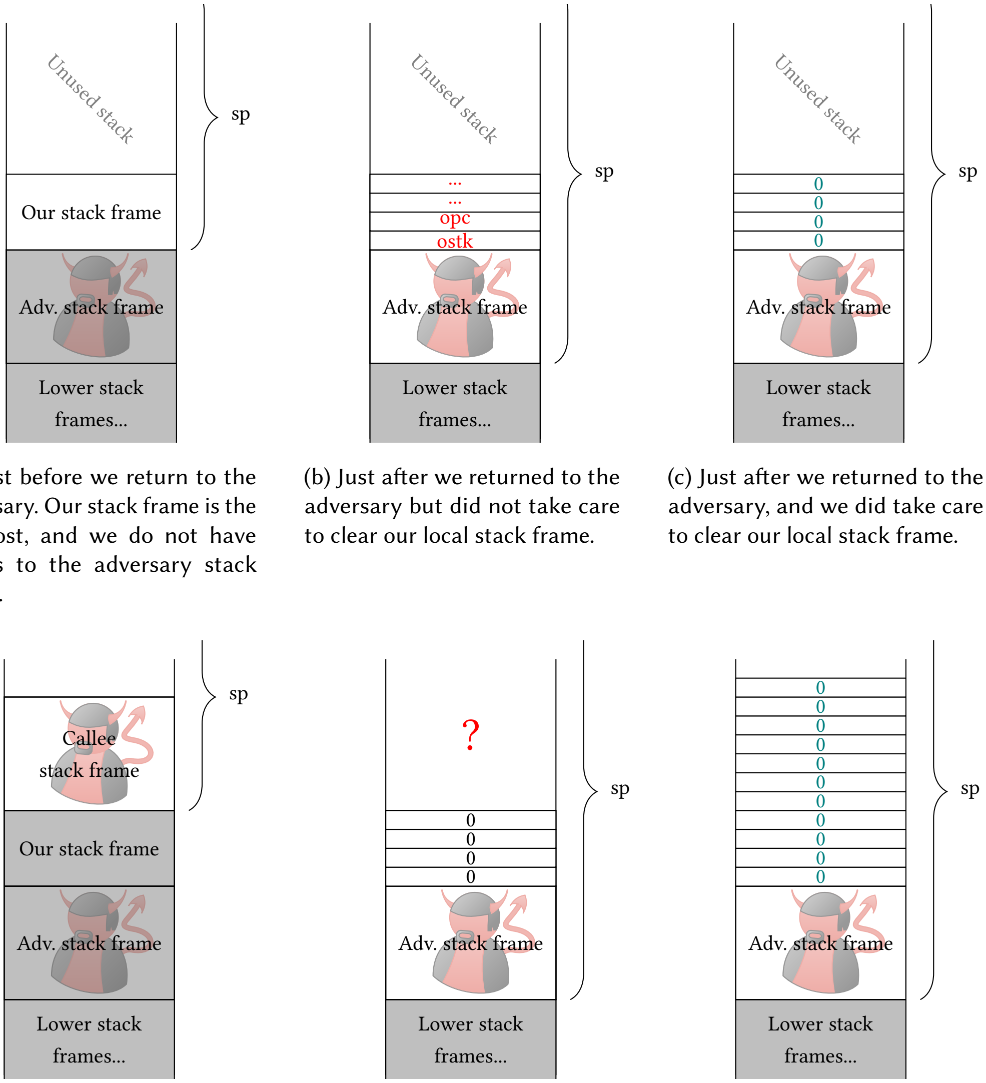
Invoking the adversary A slightly more complex case is invoking the adversary. As above, we clear all the non-argument registers, as well as the part of the stack that we are not using (because, as above, it may contain local capabilities from previously executed code that the adversary could exploit in the same way). We leave a copy of the stack pointer in r_{stk}, but only after we have used the `subseg` instruction to shrink its authority to the part that we are not using ourselves.

In one of the registers, we also provide a return pointer which must be a local capability. If it were global, the adversary would be able to store away the return pointer in a global data structure (i.e. there exists a global capability for it) and jump to it later in circumstances where this should not be possible. For example, they could store the return pointer, legally jump to it a first time, wait to be invoked again, and then jump to the old return pointer a second time instead of the new return pointer received for the second invocation. Similarly, they could store the return pointer, invoke a function in our code, wait for us to invoke them again, and then jump to the old return pointer rather than the new one received for the second invocation. By making the return pointer local, we prevent such attacks. The adversary can only store local capabilities with a write-local capability, and the only piece of memory governed by a write-local capability is the stack. Since the stack pointer itself is also local, it can also only be stored on the stack. There is no way for the adversary to recover either of these local capabilities because we clear the part of the stack that they had access to before we pass control back to them.

Note that storing stack pointers for use during future invocations would also be dangerous in itself, i.e. not just because it can be used to store return pointers. Imagine that the adversary stores their stack pointer (Figure 5a), invokes trusted code that uses part of the stack to store private data (Figure 5b) and then invokes the adversary again with a stack pointer restricted to exclude the part containing the private data (Figure 5c). If the adversary had a way of keeping hold of their old stack pointer, it could access the private data stored there by the trusted code and break local-state encapsulation.

Returning from the adversary Return pointers must be passed as local capabilities. But what should their permissions be, what memory should they point to and what should that memory (the activation record) contain? Let us answer the last question first by considering what should happen when the adversary jumps to a return pointer. In that case, the program counter should be restored to the instruction after the jump to the adversary, so the activation record should store this old program counter. Additionally, the stack pointer should also be restored to its original value. Since the adversary has a more restricted authority over the stack than the code making the call, we cannot hope to reconstruct the original stack pointer from the stack pointer owned by the adversary. Instead, it should be stored as part of the activation record.

Clearly, neither the old program counter nor the old stack pointer should be accessible by the adversary. In other words, the return pointer provided to the adversary must be a capability that they can jump to but not read from, i.e. an enter capability. To make this work, we construct the activation record as depicted in Figure 6. The E return pointer has authority over the entire activation record (containing the previous return and stack pointer) and its current address points to a number of restore instructions in the record. Upon invocation, the instructions in the activation record are executed and can load the old stack pointer and program counter back into the register file. As the



(a) Just before we return to the adversary. Our stack frame is the topmost, and we do not have access to the adversary stack frame.

(b) Just after we returned to the adversary but did not take care to clear our local stack frame.

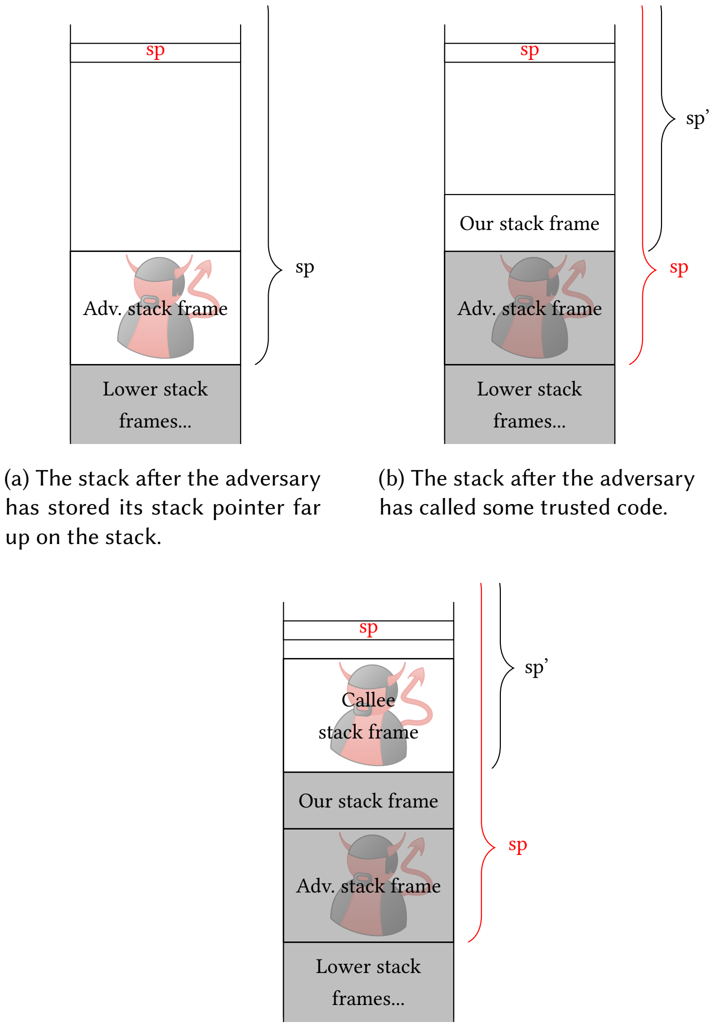
(c) Just after we returned to the adversary, and we did take care to clear our local stack frame.

(d) After an adversary has called us, and we have called the adversary.

(e) After the adversary has returned to us, and we have returned from the first adversary call. At this point, the adversary has access to the local state from the nested adversary call.

(f) After the adversary has returned to us, and we have returned from the first adversary call after we took care to clear the unused part of the stack.

Fig. 4. Depictions of the stack in relation to stack clearing. The greyed out areas are parts of the stack that are not accessible at the time. The sp capability is the stack pointer.



(a) The stack after the adversary has stored its stack pointer far up on the stack.

(b) The stack after the adversary has called some trusted code.

(c) The stack after the trusted code has called the adversary. The adversary's old stack pointer is still available on the stack.

Fig. 5. Illustration of the situations related to stack clearing when invoking an adversary. The greyed out areas are parts of the stack that are not accessible at the time.

return pointer is an enter pointer, the adversary cannot get hold of the activation record's contents. However after invocation, its permission is updated to RX, so the contents become available to the restore instructions.

The final question that remains is: where should we store this activation record? The attentive reader may already see that there is only one possibility: since the activation record contains the old stack pointer, which is local, the activation record can only be constructed in a part of memory where we have write-local access, i.e. on the stack. Note that this means we will be placing and executing instructions on the stack, i.e. it will not just contain code pointers and data. This means that our calling convention should be combined with protection against stack smashing attacks (i.e. buffer overflows on the stack overwriting activation records' contents). Luckily, the capability machine's fine-grained memory protection [Woodruff et al. 2014] should make it reasonably easy for a compiler to implement such protection by making sure that only appropriately bounded versions of the stack pointer are made available to source language code.

Invoking an adversary callback If we have a higher-order interface to the adversary, we may need to invoke an adversary callback. In this case, not so much changes with respect to the situation where we invoke static adversary code. The adversary can provide a callback as a capability for us to jump to, either an E-capability if they want to protect themselves from us or just an RX capability if they are not worried about that. However, there is one scenario that we need to prevent: if they construct the callback capability to point into the stack, it may contain local capabilities that they should not have access to upon invocation of the callback. As before, this includes return and stack pointers from previous stack frames that they may be trying to illegally use inside the callback.

To prevent this, we only accept callbacks from the adversary in the form of global capabilities, which we dynamically check before invoking them (and we fail otherwise). This should not be an overly strict requirement: our own callbacks do not contain local data themselves, so there should be no need for the adversary to construct callbacks on the stack.[2]

Having a callback invoked by the adversary The above leaves us with perhaps the hardest scenario: how to provide a callback to the adversary. The basic idea is that we allocate a block of memory using malloc that we fill with the capabilities and data that the callback needs, as well as some prelude instructions that load the data into registers and jumps to the right code. Note that this implies that no local capabilities can be stored as part of a closure. We can then provide the adversary with an enter-capability covering the allocated block and pointing to the contained prelude instructions. However, the question that remains in this setup is: from where do we get a stack pointer when the callback is invoked?

Our answer is that the adversary should provide it to us; just as we provide them with a stack pointer when we invoke their code. However, it is important that we do not just accept any capability as a stack pointer but check that it is safe to use. Specifically, we check that it is indeed an RWLX capability. Without this check, an adversary could potentially get control over our local stack frame during a subsequent callback by passing us a local RWX capability to a global data structure instead of a proper stack pointer and a global callback for our callback to invoke. If our local state contains no local capabilities, then, otherwise following our calling convention, the callback would not fail and the adversary could use a stored capability for the global data structure to access our local state. To prevent this from happening, we need to make sure the stack capability carries RWLX authority

[2] Note that it does prevent a legitimate but non-essential scenario where the adversary wants to give us temporary access to a callback not allocated on the stack.

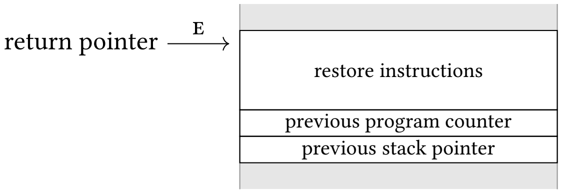


Fig. 6. Structure of an activation record

since the system wide assumption then tells us that the adversary cannot have global capabilities to our local stack.

Calling convention With the security measures introduced and motivated, let us summarize our proposed calling convention:

At program start-up A local RWLX stack pointer resides in register r_{stk}. No global write-local capabilities.

Before returning to the adversary Clear non-return-value registers. Clear the part of the stack we had access to (not just the part we used).

Before invoking the adversary Push activation record to the stack. Create return pointer as local E-capability to the instructions in the record. Restrict the stack capability to the unused part and clear it. Clear non-argument registers.

Before invoking an adversary callback Make sure callback is global.

When invoked by an adversary Make sure received stack pointer has permission RWLX.

Modularity The calling convention ensures well-bracketed calls and local-state encapsulation for the caller but not the callee. In the above presentation to make it easy to distinguish between the parties involved, we present the callee as some adversarial code that we do not trust. In reality, the callee could be well-behaved and wish to ensure well-bracketed calls and local-state encapsulation as well. The calling convention puts no restriction on the callee that the caller itself does not follow, so by following the calling convention, the callee can also obtain those guarantees. In other words, the calling convention is modular and scales to scenarios with multiple distrusting parties invoking each other.

4 LOGICAL RELATION

Now that we have defined our calling convention, how can we be sure that it works? More concretely, suppose that we have a program that uses the convention in its interaction with untrusted adversary code. Can we formally prove the program's correctness if it relies on well-bracketed control flow and private state encapsulation for the interaction with the adversary? Clearly, such a proof should depend on a formal expression of the guarantees provided by the capability machine, including the specific guarantees for local capabilities.

In this section, we construct such a formalization. We make use of some well-studied and powerful (but non-trivial) machinery from the literature. Specifically, we employ a unary step-indexed Kripke logical relation with recursive worlds and some additional special characteristics of our own. Step-indexing, Kripke logical relations, and recursive worlds are techniques that may be familiar from lambda calculus settings, but it may not be clear to the reader how they apply in this more low-level assembly language. Therefore, in the next section, we do not immediately dive into the details, but first we try to provide some informal intuition about how all of this machinery comes into play in our setting.

Note: even though the calling convention is the main application in this paper, the logical relation we construct is very general and can be seen as a formulation of capability safety; hence it should be regarded as an independent contribution.

4.1 Formalizing the guarantees of the capability machine

What differentiates a capability machine from a more standard assembly language is that we can bound the authority of an executing block of code based solely on the capabilities it has access to. Specifically, it does not matter which instructions are actually executed, i.e. the bound also applies to untrusted adversary code that has not been inspected or modified in any way.

Worlds. But what does a "bound on the authority" of an executing block of code mean? In our setting, there are no externally observable side-effects; the only primitive authority that code may hold is authority over memory. As such, the authority bounds we consider are related to memory, but in a form that is more fine-grained than standard read/write authority: a piece of code's authority can be bounded by arbitrary memory invariants that it is required to respect. Specifically, we will define worlds $W \in \text{World}$, which describe a set of memory invariants, and our results will express authority bounds on code as safety with respect to such a world, i.e. the fact that the code respects the invariants registered in the world.

Safe values. Suppose we have a world W expressing that the memory must contain value 42 at address 0, may contain arbitrary values at addresses 50-60, a RW capability for address 0 at address 73, and an integer at address 100 that may only increase over time[3]. Our main theorem will state that if the current register file only contains safe words (numbers or capabilities which preserve the invariants in W under any interaction), then an execution will necessarily also preserve the memory invariants (irrespective of the instructions being executed).

To make this more precise, we need to define the set $\mathcal{V}(W) \in \mathcal{P}(\text{Word})$ of words that are safe w.r.t. W. Essentially, the set should only include words that preserve W's invariants under any interaction, but should otherwise be as liberal as possible. Numbers are clearly always safe as they cannot be used to break invariants. Whether a capability is safe depends on the authority that it carries. In the above-described world, a read capability for address 0 is safe as it can only be used to read the value 42, which is itself safe. However, a write capability for address 0 is not safe: it can be used to overwrite the memory at that address with a value other than 42 breaking the invariant for that address.

Step-indexing. More generally, we want to define that a read capability for memory range $[b, e]$ is safe if the world guarantees that the words at those addresses are themselves safe. However, this definition is cyclic: suppose the world guarantees that the memory at address a will contain a read capability for address a? Then the definition says that a read capability for address a is safe if and only if the same read capability for address a is safe. This form of cyclic reasoning is related to similar challenges in languages with recursive types or higher-order ML-style references, and a standard solution is to use step-indexing [Appel and McAllester 2001]: essentially, the cycle is broken by defining safety up to a certain number of interaction steps. All words will be considered safe up to 0 steps (since if there is no interaction, nothing unsafe can happen), and, for example, a read capability will be safe up to n steps if the world guarantees that the words at the corresponding addresses are safe up to $n - 1$ steps. We can then prove that the above read capability for address a is safe up to any number of steps.

Future worlds. Worlds are defined as a set of invariants on the memory, but what if we allocate fresh memory through malloc? We may want to establish new invariants for this freshly allocated memory and be sure that the adversary will also respect those (if we don't provide them with capabilities through which the new invariants can be broken). To accommodate this, we allow worlds to evolve, for example by adding additional invariants for freshly allocated memory. Formally, we define valid ways for a world W to evolve into a new world W' through a future-world relation $W' \sqsupseteq W$ and we ensure that the set of safe words in world W must remain safe in any future world W'. Defining safety w.r.t. a notion of evolvable worlds makes our logical relation into a Kripke logical relation [Pitts and Stark 1998].

[3] Indeed, we will allow a notion of evolvable invariants, aka protocols, that can express such a temporal property.

Invariants and Recursive Worlds. Worlds group a set of memory invariants, but how are they actually defined formally? We represent each invariant by a region $r \in \text{Region}$. We will see later that regions contain state machines to support a notion of evolvable invariants. Every state of the state machine contains a predicate H that defines the valid memory segments. Unfortunately, it is not enough to just take $H \in \mathcal{P}(\text{MemSeg})$, because sometimes the invariant may itself be world-dependent. For example, we may want to express invariants like "the memory at address 50 contains a value that is safe in the current world". As explained, worlds may evolve, and the set of safe values may grow in future worlds. Therefore, we need to index H over worlds, i.e., take $H \in \text{World} \rightarrow \mathcal{P}(\text{MemSeg})$. The result is worlds containing regions with world-indexed predicates, i.e., the set of worlds must be recursively defined. We will see how such a recursive definition can be accommodated using techniques from the literature (essentially an advanced application of step-indexing).

Local capabilities. When we invoke an untrusted piece of code and provide it with certain global capabilities, it may have stored those capabilities in memory. In this case, we will only be able to reinvoke the code if we can guarantee that those values are still valid. Formally, worlds represent the invariants that global capabilities' safety relies on and the reinvocation is only safe in future worlds where the invariants are respected.

However, if we provide the adversary with local capabilities in that first invocation, then the situation is a bit different. The adversary has no way to store these local capabilities, so if we make sure that there are also no old local capabilities in the register file for the second invocation (including the capability being invoked), then the adversary cannot use them any more.[4] Therefore, we can allow the second invocation to happen in any private future worlds ($W' \sqsupseteq^{priv} W$) in which safe global capabilities remain safe but local capabilities do not. This private future world relation is more liberal than the standard public one ($W' \sqsupseteq^{pub} W$, in which all safe capabilities remain safe). Concretely, worlds may contain temporary regions representing invariants that only local capabilities may rely on for their safety and that may be revoked (disabled) in private future worlds.

Interestingly, this idea is a variant of a notion of public/private future worlds that has been previously used in the literature (see Section 9). However, temporary regions are new in our setting and there is an interesting interplay with the recursiveness of the worlds: for a temporary region, the predicate $H \in \text{World} \rightarrow \mathcal{P}(\text{MemSeg})$ (which defines the safe memory segments in the current world) is only required to be monotone w.r.t. public future worlds (i.e. safe memory segments remain safe in public future worlds). On the other hand for permanent regions, the world-indexed predicate must be monotone w.r.t. private future worlds. As a consequence, the memory for a permanent region may not contain local capabilities (as their safety would be broken in private future worlds), which in turn implies that only local capabilities may have write-local permission (a general sanity requirement when using local capabilities[5])[6].

4.2 Worlds

A world is a semantic model of the memory that carves out memories with a particular shape from all the possible memories. In our correctness proofs, worlds allow us to rely on the memory having a particular shape, but sometimes we will also have to guarantee that the memory has a certain

[4] We ignore write-local capabilities in this discussion. If the adversary does have access to write-local capabilities in the first and second invocation, then the memory they address must be cleared entirely before the second invocation in order for the reinvocation to remain safe.

[5] In fact, local capabilities become useless as soon as the adversary has access to a single global, write-local capability.

[6] For explanation purposes, this discussion ignores certain ways to allow for local capabilities in a permanent region, for example, by not requiring that they are valid or requiring that they are local versions of valid global capabilities.

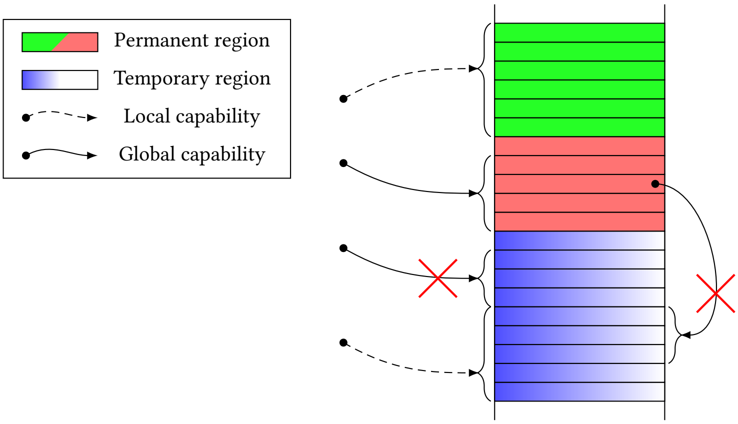


Fig. 7. The relation between local/global capabilities and temporary/permanent regions. The colored fields are regions governing parts of memory. Global capabilities cannot depend on temporary regions.

shape. Essentially, a world is a collection of invariants. The memory satisfies the world when part of the memory satisfy each of the invariants.

Worlds are represented as a finite map from region names, modeled as natural numbers, to regions that each correspond to an invariant on part of the memory. We have three types of regions: permanent, temporary, and revoked. Each permanent and temporary region contains a state transition system with public and private transitions that describe how the invariant is allowed to change over time. In other words, they are protocols for the region's memory. Protocols imposed by permanent regions stay in place indefinitely. Any capability, local or global, can depend on these protocols. Protocols imposed by temporary regions can be revoked in private future worlds. Doing this may break the safety of local capabilities but not global ones. This means that local capabilities can safely depend on the protocols imposed by temporary regions, but global capabilities cannot since a global capability may outlive a temporary region that is revoked. This is illustrated in Figure 7.

We need the future world relation to be extensional, so we do not actually remove a revoked temporary region from the world, but we turn it into a special revoked region that exists for this purpose. Such a revoked region contains no state transition system and puts no requirements on the memory. It simply serves as a mask for a revoked temporary region. Masking a region like this goes back to earlier work of Ahmed [2004] and was also used by Thamsborg and Birkedal [2011].

Regions are used to define safe memory segments, but this set may itself be world-dependent. In other words, our worlds are defined recursively. Recursive worlds are common in Kripke models and the following theorem uses the method of Birkedal and Bizjak [2014]; Birkedal et al. [2011] for constructing them. The formulation of the lemma is technical, so we recommend that non-expert readers ignore the technicalities and accept that there exists a set of worlds Wor and two relations $\sqsupseteq^{priv}$ and $\sqsupseteq^{pub}$ satisfying the (recursive) equations in the theorem (where the ▶ operator can be safely ignored[7]). The solution Wor is some c.o.f.e. that we do not know much about that is, essentially, isomorphic to the our worlds. The isomorphism ξ allows us to move between the solution and the worlds we can work with and respects the future world relations.

Theorem 4.1. *There exists a c.o.f.e. (complete ordered family of equivalences)* Wor *and preorders* $\sqsupseteq^{priv}$ *and* $\sqsupseteq^{pub}$ *such that* $(\mathrm{Wor}, \sqsupseteq^{priv})$ *and* $(\mathrm{Wor}, \sqsupseteq^{pub})$ *are preordered c.o.f.e.'s, and there exists an isomorphism* ξ *such that*

$$\begin{aligned}
\xi\ &: \mathrm{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} \mathrm{Region})\\
\mathrm{Region} &= \{\mathrm{revoked}\} \uplus\\
&\qquad \{\mathrm{temp}\} \times \mathrm{RState} \times \mathrm{Rels} \times (\mathrm{RState} \to (\mathrm{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \mathrm{UPred}(\mathrm{MemSeg}))) \uplus\\
&\qquad \{\mathrm{perm}\} \times \mathrm{RState} \times \mathrm{Rels} \times (\mathrm{RState} \to (\mathrm{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\, ne} \mathrm{UPred}(\mathrm{MemSeg})))
\end{aligned}$$

and for $W, W' \in \mathrm{Wor}$.
$$\begin{aligned}
W' \sqsupseteq^{priv} W &\Leftrightarrow \xi(W') \sqsupseteq^{priv} \xi(W)\\
W' \sqsupseteq^{pub} W &\Leftrightarrow \xi(W') \sqsupseteq^{pub} \xi(W)
\end{aligned}$$

In the above theorem, RState × Rels corresponds to the aforementioned state transition system where Rels contains pairs of relations corresponding to the public and private transitions, and RState is a set of world states that we assume to at least contain the states we use in this paper. The last part of the temporary and permanent regions is a state interpretation function that determines what memory segments the region permits in each state of the state transition system. The different monotonicity requirements in the two interpretation functions reflect how permanent regions rely only on permanent protocols whereas temporary regions can rely on both temporary and permanent protocols. UPred(MemSeg) is the set of step-indexed, downwards closed predicates on memory segments: $\mathrm{UPred}(\mathrm{MemSeg}) = \{A \subseteq \mathbb{N} \times \mathrm{MemSeg} \mid \forall (n, ms) \in A. \forall m \leq n. (m, ms) \in A\}$.

With the recursive domain equation solved, we could take Wor as our notion of worlds, but it is technically more convenient to work with the following definition instead:

$$\mathrm{World} = \mathbb{N} \xrightarrow{fin} \mathrm{Region}$$

4.2.1 Future Worlds. The future world relations model how memory may evolve over time. We have a public and a private future world relation that, respectively, model the memory changes any capability can rely on and the memory changes only local capabilities can rely on. As local capabilities fall within the category of all capabilities, the public future relation is subsumed in the private future relation.

The public future world $W' \sqsupseteq^{pub} W$ requires that $\mathrm{dom}(W') \supseteq \mathrm{dom}(W)$ and $\forall r \in \mathrm{dom}(W).\ W'(r) \sqsupseteq^{pub} W(r)$. That is in a public future world, new regions may have been allocated, and existing regions may have evolved according to the public future region relation (defined below). The private future world relation $W' \sqsupseteq^{priv} W$ is defined similarly, using a private future region relation. The public future region relation is the simplest. It satisfies the following properties:

$$\frac{(s, s') \in \phi_{pub}}{(v, s', \phi_{pub}, \phi, H) \sqsupseteq^{pub} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{(\mathrm{temp}, s, \phi_{pub}, \phi, H) \in \mathrm{Region}}{(\mathrm{temp}, s, \phi_{pub}, \phi, H) \sqsupseteq^{pub} \mathrm{revoked}}$$

$$\frac{}{\mathrm{revoked} \sqsupseteq^{pub} \mathrm{revoked}}$$

[7] The interested reader can find a brief coverage of c.o.f.e.'s and ▶ in Appendix A.2.

In public future worlds, both temporary and permanent regions are only allowed to transition according to the public part of their transition system. Additionally, revoked regions must either remain revoked or be replaced by a temporary region. This means that the public future world relations allows us to reinstate a region that has been revoked earlier. The private future region relation satisfies:

$$\frac{(s, s') \in \phi}{(v, s', \phi_{pub}, \phi, H) \sqsupseteq^{priv} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{r \in \text{Region}}{r \sqsupseteq^{priv} (\text{temp}, s, \phi_{pub}, \phi, H)} \qquad \frac{r \in \text{Region}}{r \sqsupseteq^{priv} \text{revoked}}$$

Here, revocation of temporary regions is allowed. In fact, temporary regions can be replaced by an arbitrary region not just the special revoked. Conversely, revoked regions may also be replaced by any other region. On the other hand, permanent regions cannot be masked away. They are only allowed to transition according to the private part of the transition system.

Intuitively, the future world relation specifies how memory protocols may change over time. Rather than deleting regions in future worlds, we follow the approach in Ahmed [2004]; Thamsborg and Birkedal [2011] and use masks to signal which regions are active. This approach implies that the future world relation is a preorder and hence we can use the method in Birkedal and Bizjak [2014]; Birkedal et al. [2011] to solve the recursive world equation.

4.2.2 World Satisfaction. A memory satisfies a world, written $ms :_n W$, if it can be partitioned into disjoint parts such that each part is accepted by an active (permanent or temporary) region. Revoked regions are not taken into account as their memory protocols are no longer in effect.

$$ms :_n W \text{ iff } \begin{cases} \exists P : active(W) \rightarrow \text{MemSeg}.\ ms = \biguplus_{r \in active(W)} P(r) \text{ and} \\ \quad \forall r \in active(W). \\ \qquad \exists H, s.\ W(r) = (\_, s, \_, \_, H) \text{ and } (n, P(r)) \in H(s)(\xi^{-1}(W)) \end{cases}$$

4.3 Logical Relation

The logical relation defines semantically when values, program counters, and configurations are capability safe. The logical relation is defined in Figures 8 and 9, and we provide some explanations in the following paragraphs. For space reasons, we omit some definitions and explain them only verbally, but precise definitions can be found in the appendix. The logical relation is recursively defined, so we encourage first time readers to just read the section in its entirety and do a second read afterwards.

First, the observation relation $\mathcal{O}$ defines what configurations we consider safe. A configuration is safe with respect to a world, when the execution of said configuration does not break the memory protocols of the world. Roughly speaking, this means that when the execution of a configuration halts, then there is a private future world that the resulting memory satisfies. Notice that failing is considered safe behavior. In fact, the machine often resorts to failing when an unauthorized access is attempted such as loading from a capability without read permission. This is similar to Devriese et al. [2016]'s logical relation for an untyped language but unlike typical logical relations for typed languages that require programs to not fail.

The register-file relation $\mathcal{R}$ defines safe register-files as those that contain safe words (i.e. words in the $\mathcal{V}$-relation defined below) in all registers but the pc-register. The expression relation $\mathcal{E}$ defines what words are safe to use as a program counter. A word is safe to use as a program counter when it can be used to form a safe configuration by plugging it into the pc-register of a safe register file (i.e. a register file in $\mathcal{R}$) and pairing it with a memory satisfying the world. Note that integers and non-executable capabilities (e.g. RO and E capabilities) are considered safe program counters

$$\mathcal{O} \;:\; \text{World} \xrightarrow{ne} \text{UPred}(\text{Reg} \times \text{MemSeg})$$

$$\mathcal{O}(W) \overset{def}{=} \left\{ (n, (reg, ms)) \;\middle|\; \begin{array}{l} \forall ms_f, mem', i \leq n.\, (reg, ms \uplus ms_f) \rightarrow_i (halted, mem') \Rightarrow \\ \quad \exists W' \sqsupseteq^{priv} W, ms_r, ms'. \\ \qquad mem' = ms' \uplus ms_r \uplus ms_f \text{ and } ms' :_{n-i} W' \end{array} \right\}$$

$$\mathcal{R} \;:\; \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Reg})$$

$$\mathcal{R}(W) \overset{def}{=} \{(n, reg) \mid \forall r \in \text{RegName} \setminus \{\text{pc}\}.\, (n, reg(r)) \in \mathcal{V}(W)\}$$

$$\mathcal{E} \;:\; \text{World} \xrightarrow{ne} \text{UPred}(\text{Word})$$

$$\mathcal{E}(W) \overset{def}{=} \left\{ (n, pc) \;\middle|\; \begin{array}{r} \forall n' \leq n, (n', reg) \in \mathcal{R}(W), ms :_{n'} W. \\ (n', (reg[\text{pc} \mapsto pc], ms)) \in \mathcal{O}(W) \end{array} \right\}$$

$$\mathcal{V} \;:\; \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Word})$$

$$\begin{aligned} \mathcal{V}(W) \overset{def}{=} {} & \{(n, i) \mid i \in \mathbb{Z}\} \cup \\ & \{(n, ((\textsc{o}, g), b, e, a))\} \cup \\ & \{(n, ((\textsc{ro}, g), b, e, a)) \mid (n, (b, e)) \in readCond(g)(W)\} \cup \\ & \left\{ (n, ((\textsc{rw}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{nwl}, g)(W) \end{array} \right\} \cup \\ & \left\{ (n, ((\textsc{rwl}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{pwl}, g)(W) \end{array} \right\} \cup \\ & \left\{ (n, ((\textsc{rx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (\{\textsc{rx}\}, b, e)) \in execCond(g)(W) \end{array} \right\} \cup \\ & \{(n, ((\textsc{e}, g), b, e, a)) \mid (n, (b, e, a)) \in enterCond(g)(W)\} \cup \\ & \left\{ (n, ((\textsc{rwx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{nwl}, g)(W) \text{ and} \\ (n, (\{\textsc{rwx}, \textsc{rx}\}, b, e)) \in execCond(g)(W) \end{array} \right\} \cup \\ & \left\{ (n, ((\textsc{rwlx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{pwl}, g)(W) \text{ and} \\ (n, (\{\textsc{rwlx}, \textsc{rwx}, \textsc{rx}\}, b, e)) \in execCond(g)(W) \end{array} \right\} \end{aligned}$$

Fig. 8. The logical relation.

because when they are plugged into a register file and paired with a memory, the execution will immediately fail, which is safe.

The value relation $\mathcal{V}$ defines when words are safe. We make the value relation as liberal as possible by defining it based on the principle "what is the most we can allow an adversary to use a capability for without breaking the memory protocols."

Non-capability data is always safe because it provides no authority. Capabilities give the authority to manipulate memory and potentially break memory protocols, so they need to satisfy certain conditions to be safe. In Figure 9, we define such a condition for each kind of permission a capability can have.

$$
\begin{aligned}
\mathit{readCond}(g)(W) &= \left\{ (n, (b, e)) \;\middle|\; \begin{array}{l} \exists r \in \mathit{localityReg}(g, W). \\ \quad \exists [b', e'] \supseteq [b, e].\, W(r) \overset{n}{\underset{\sim}{\subset}} \iota^{pwl}_{b',e'} \end{array} \right\} \\
\mathit{writeCond}(\iota, g)(W) &= \left\{ (n, (b, e)) \;\middle|\; \begin{array}{l} \exists r \in \mathit{localityReg}(g, W). \\ W(r) \text{ is address-stratified and} \\ \exists [b', e'] \supseteq [b, e].\, W(r) \overset{n-1}{\underset{\sim}{\supset}} \iota_{b',e'} \end{array} \right\} \\
\mathit{execCond}(g)(W) &= \left\{ (n, (P, b, e)) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq W, a \in [b', e'] \subseteq [b, e], \mathit{perm} \in P. \\ \quad (n', ((\mathit{perm}, \mathrm{g}), b', e', a)) \in \mathcal{E}(W') \end{array} \right\} \\
\mathit{enterCond}(g)(W) &= \left\{ (n, (b, e, a)) \;\middle|\; \begin{array}{l} \forall n' < n. \forall W' \sqsupseteq W. \\ \quad (n', ((\textsc{rx}, g), b, e, a)) \in \mathcal{E}(W') \end{array} \right\} \\
&\text{where } g = \text{local} \Rightarrow \sqsupseteq \,=\, \sqsupseteq^{pub} \text{ and } g = \text{global} \Rightarrow \sqsupseteq \,=\, \sqsupseteq^{priv}
\end{aligned}
$$

Fig. 9. Permission-based conditions

$$
\begin{aligned}
\iota^{pwl}_A, \iota^{nwl}_A &: \text{Region} \\
\iota^{pwl}_A &\overset{def}{=} (\text{temp}, 1, =, =, H^{pwl}_A) \\
\iota^{nwl}_A &\overset{def}{=} (\text{temp}, 1, =, =, H^{nwl}_A) \\
\iota^{nwl,p}_A &\overset{def}{=} (\text{perm}, 1, =, =, H^{nwl}_A) \\
H^{pwl}_A &: \text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred(MemSeg)}) \\
H^{pwl}_A(s)(\hat{W}) &\overset{def}{=} \left\{ (n, ms) \;\middle|\; \begin{array}{l} \text{dom}(ms) = A \text{ and} \\ \forall a \in A.\, (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\} \\
H^{nwl}_A &: \text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\, ne} \text{UPred(MemSeg)}) \\
H^{nwl}_A(s)(\hat{W}) &\overset{def}{=} \left\{ (n, ms) \;\middle|\; \begin{array}{l} \text{dom}(ms) = A \text{ and} \\ \forall a \in A. \\ \quad ms(a) \text{ is non-local and} \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\}
\end{aligned}
$$

Fig. 10. The standard permit write-local and no write-local regions.

Capabilities with read permission cannot directly break memory protocols because they cannot make changes to the memory. However, a read capability could be used to read a capability that can break memory protocols. For this reason, a capability with read permissions is only safe when it can be used only to read safe capabilities. This is captured by the *readCond*-condition. More precisely, the condition requires the memory addressed by the capability to actually be governed by some region $W(r)$. This region should at least require all the memories accepted by the state interpretation function to contain safe words but stricter requirements may be imposed on the memories. This is expressed in terms of an upper bound on the possibly permitted memories defined in terms of a so called "standard region". In this case, it is a permit-write-local standard region $\iota^{pwl}_{[b,e]}$ (defined in Figure 10). The permit-write-local standard region only accepts memories with address range $[b, e]$. More importantly, the permit-write-local standard region's state interpretation function only accepts memory segments that only contain safe words. The relation between the actual region in the world and the upper bound is $W(r) \overset{n}{\underset{\sim}{\subseteq}} \iota^{pwl}_{[b,e]}$ (defined in Appendix A.1). Essentially, the relation requires the two regions state transition systems to be the same. Further in the current state of region $W(r)$ and in any world, the state interpretation function should only allow memories that are also allowed in the standard region.

The locality of capabilities play a role in whether or not they are safe. Generally speaking, global capabilities should only depend on permanent regions. This is expressed in *readCond* with *localityReg*(g, W) which projects out all the regions the capability can depend on based on its locality. Specifically, if the capability is local, then all active regions (non-revoked) are projected. On the other hand, if the capability is global, then only the permanent regions are projected.

Capabilities with write permission can be used to write to memory, so to define when a capability with write permission is safe we ask ourselves what an adversary should be allowed to do with it. An adversary should at least be able to write safe words to memory. Safe words cannot be used to break memory invariants, so we will permit any safe word to be written. A capability with write permission[8] only allows you to write to memory not read from it, so we can allow anything to be written to memory as long as it cannot be read back and used to break memory protocols. For this reason, the condition on capabilities with write-permission is defined as a lower bound on what can be written where the lower bound is "any safe word".

The condition on write capabilities is complicated by the fact that we have two flavours of write permission: write and write-local. A write-local capability can be used to write both local and global capabilities, so it is (at least) allowed to write any safe capabilities. Write capabilities, on the other hand, cannot write local capabilities, so we set the lower-bound as non-local safe words.

The requirements on write-capabilities are captured by the *writeCond*-condition, and it is defined in a similar fashion to the *readCond*-condition. The regions the write-capability may depend on are projected from the world with the *localityReg*-function, and $\overset{n-1}{\underset{\sim}{\supseteq}}$ is used to relate the actual region to the lower bound. We need a different lower-bound depending on whether the permission of the capability is write or write-local, so *writeCond* is parameterized with the region it actually uses. If the capability has write-local permission, then we use the permit-write-local standard region. If capability only has write permission, then we use a no-write-local standard region $\iota^{nwl}_{[b,e]}$ (defined in Figure 10). In addition to the requirements of a permit-write-local standard region, the no-write-local standard region requires all words in the accepted memory segments to be non-local.

Finally, there is a technical requirement that the region must be address-stratified. Intuitively, this means that if a region accepts two memory segments, then it must also accept every memory segment "in between" in the sense that it should be possible to come from one memory segment to the other. Our capability machine can only update one memory address at a time, so it should be accepted to construct a memory that is on its way to become the one we want to accept. Specifically, if two memory segments are accepted by a region, then the region must also accept every memory segment where each address contains a value from one of the two accepted memory segments.

Due to the permission combinations on the capability machine we consider, a *writeCond* always comes with a *readCond* which creates a somewhat tight bound for the possible regions that make a capability safe. Further, we could have made the *writeCond* based on the fact that a writer permission

[8] On the capability machine we consider, write permission always comes with read permission, so we could have made a stricter condition for write capabilities.

always comes with a read permission, but we opted to make the logical relation as general as possible, so it can be reused in a setting with a richer permission hierarchy.

The conditions *enterCond* and *execCond* are very similar. Both require that the capability can be safely jumped to. However, executable capabilities can be updated to point anywhere in their range, so they must be safe as a program counter (in the $\mathcal{E}$-relation) no matter the current address. The range of an executable capability can also be reduced, so they must also be safe as program counter no matter what their range of authority is reduced to. In contrast, enter capabilities are opaque and can only be used to jump to the address they point to. This is why *enterCond* depends on the current address of the capability unlike for other types of capabilities. They also change permission when jumped to, so we require them to be safe as a program counter after the permission is changed to RX. Because the capabilities are not necessarily invoked immediately, this must be true in any future world, but it depends on the capability's locality which future worlds we consider. If it is global, then we require safety as a program counter in private future worlds (where temporary regions may be revoked). For local capabilities, it suffices to be safe in public future worlds where temporary regions are still present.

In the technical appendix, we prove that safety of all values is preserved in public future worlds, and that safety of global values is also preserved in private future worlds:

Lemma 4.2 (Double monotonicity of value relation).

- *If* $W' \sqsupseteq^{pub} W$ *and* $(n, w) \in \mathcal{V}(W)$*, then* $(n, w) \in \mathcal{V}(W')$.
- *If* $W' \sqsupseteq^{priv} W$ *and* $(n, w) \in \mathcal{V}(W)$ *and* $w = ((perm, \text{global}), b, e, a)$ *(i.e. w is a global capability), then* $(n, w) \in \mathcal{V}(W')$.

In Section 3, we require capabilities with write-local permission to be local. This indicates that our logical relation should imply the same, namely that capabilities with write-local permission are local.

Lemma 4.3 (Stack capabilities are local).

- *If* $ms :_n W$ *and* $(n, ((perm, g), b, e, a)) \in \mathcal{V}(W)$ *and* $b \leq e$ *and* $perm \in \{\textsc{rwlx}, \textsc{rwl}\}$*, then* $g = \text{local}$.

In our definition of worlds, nothing prevents a world from having regions that are overlapping. This may seem like an issue with the $\mathcal{V}$-relation as it allows different permission-based requirements to be satisfied by different regions. In practice, it is not an issue as we will always have a memory satisfaction assumption which doubles as a well-formedness condition on the world as it prevents the regions from overlapping. The memory satisfaction assumption in Lemma 4.3 is there to ensure a well-formed world.

4.4 Capability Machine Safety

With the logical relation defined, we can now state the fundamental theorem of our logical relation: a strong theorem that formalizes the guarantees offered by the capability machine. Essentially, it says a capability that only grants safe authority is capability safe as a program counter.

Theorem 4.4 (Fundamental Theorem). *If one of the following holds:*

- $perm = \textsc{rx}$ *and* $(n, (b, e)) \in readCond(g)(W)$
- $perm = \textsc{rwx}$ *and* $(n, (b, e)) \in readCond(g)(W)$ *and* $(n, (b, e)) \in writeCond(\iota^{nwl}, g)(W)$
- $perm = \textsc{rwlx}$ *and* $(n, (b, e)) \in readCond(g)(W)$ *and* $(n, (b, e)) \in writeCond(\iota^{pwl}, g)(W)$,

then $(n, ((perm, g), b, e, a)) \in \mathcal{E}(W)$

Proof sketch. Induction over n. By definition of $\mathcal{E}(W)$, show

$$(n', (reg[\mathrm{pc} \mapsto ((perm, g), b, e, a)], ms)) \in \mathcal{O}(W)$$

assuming $n' \leq n$, $(n', reg) \in \mathcal{R}(W)$, and $ms :_{n'} W$. By definition of $\mathcal{O}$ let ms_f, mem', and $i \leq n'$ be given and for $\Phi = (reg[\mathrm{pc} \mapsto ((perm, g), b, e, a)], ms \uplus ms_f)$ assume $\Phi \rightarrow_i (halted, mem')$ and show there exists $W' \sqsupseteq^{priv} W$ that part of mem' satisfies. First observe that $i \neq 0$ as Φ is a non-halted configuration, so Φ takes at least one step, i.e. $\Phi \rightarrow \Phi' \rightarrow_{i-1} (halted, mem')$.

The rest of the proof considers the different possibilities for the step $\Phi \rightarrow \Phi'$, i.e. it considers each of the instructions that could have been executed. For each of these cases, we argue that (1) Φ' is consistent with the world in the sense that the register-file and memory still respect the world and that (2) the rest of the execution respects the world. Depending on where the pc in Φ' comes from, the second result is proven in one of two ways. If the step to Φ' was a jump, then the new pc is one of the safe values in Φ's registers, and the value relation is used to argue that the jump is safe. On the other hand, if the pc was just incremented, then we can apply the induction hypothesis.

In order to argue (1), we look at the possible states for Φ according to the operational semantics. If we consider the memory in Φ', then it either (a) remains unchanged or (b) one address has been updated and the register-file contains an appropriate capability for writing. The latter occurs when the executed instruction is `store`. Otherwise we are in the former case. In case (a), the memory hasn't changed, so we conclude that the memory still respects the world simply by downwards closure of memory satisfaction. In case (b), we use an auxiliary lemma that uses the safety of the write capability used by the `store` instruction to show that the updated memory satisfies the world. To show that the updated register-file is safe, we consider the changes made to it by all instructions in separate lemmas and show that they all preserve safety of the register file.

The complete proof can be found in the technical appendix [Skorstengaard et al. 2019a]. □

The permission-based conditions of Theorem 4.4 make sure that the capability only provides safe authority in which case the capability must be in the $\mathcal{E}$ relation, i.e. it can safely be used as a program counter in an otherwise safe register-file.

The Fundamental Theorem can be understood as a general expression of the guarantees offered by the capability machine which is an instance of a general property called capability safety [Devriese et al. 2016; Maffeis et al. 2010]. The theorem says that an arbitrary capability $((perm, g), b, e, a)$ is safe as a program counter without making any assumptions about what program it points to (the only assumptions we have are about the read or write authority that it carries). As such, the theorem expresses the capability safety of the machine which guarantees that any instruction is safe to execute and will not be able to go beyond the authority of the values it has access to. We demonstrate this in Section 8 where Theorem 4.4 is used to reason about capabilities that point to arbitrary instructions. The relation between Theorem 4.4 and local-state encapsulation and control-flow correctness will also be shown by example in Section 8 as the examples depend on these properties for correctness.

5 MALLOC

In the examples presented in Section 8, we will assume the existence of a trusted malloc routine, so that both the trusted code and the adversary are able to allocate new memory. Malloc is considered part of the trusted computing base as mentioned in Section 2. This is unavoidable: if we cannot trust malloc, then we cannot use the memory it allocates as we have no idea whether it is aliased by some untrusted program.

Our semantic model is not specific to a particular implementation of malloc, so rather than providing the implementation we provide a malloc specification. The specification expresses standard expectations about the behaviour of malloc, but making the specification realistic requires some of the technical machinery from the logical relation. As such, this section is a bit technical and can safely be skipped on first read. Definition 5.1 is the malloc specification. Following the definition, we provide an informal description of the definition with references to each of the items in the definition.

Definition 5.1 (Malloc Specification). c_{malloc} satisfies the specification for malloc iff the following conditions hold:

(1) $c_{malloc} = ((\textsc{e}, \text{global}), \_, \_, \_)$

(2) There exists a $\iota_{malloc,0}$ such that

(a) $\iota_{malloc,0}.v = \text{perm}$

(b) For all $\iota' \sqsupseteq^{priv} \iota_{malloc,0}, W, i$ with $W(i) = \iota'$, we have that

$$\iota'.H(\iota'.s)(\xi^{-1}(W)) = \iota'.H(\iota'.s)(\xi^{-1}([i \mapsto W(i)]))$$

(c) For all $\Phi \in \text{ExecConf}$, $ms_{footprint}, ms_{frame} \in \text{MemSeg}$, $i, n, size \in \mathbb{N}$, $w_{ret} \in \text{Word}$, $\iota_{malloc} \sqsupseteq^{priv} \iota_{malloc,0}$, we have that

$$\begin{array}{l}
\text{If } \Phi.\text{mem} = ms_{footprint} \uplus ms_{frame} \land ms_{footprint} :_n [i \mapsto \iota_{malloc}] \land \\
\quad \Phi.\text{reg}(r_1) = size \land size \geq 0 \land \Phi.\text{reg}(r_0) = w_{ret} \land \\
\quad \Phi.\text{reg}(\text{pc}) = updPcPerm(c_{malloc}) \\
\text{Then,} \\
\quad \exists \Phi' \in \text{ExecConf}.\, \exists ms'_{footprint}, ms_{alloc} \in \text{MemSeg}. \\
\quad\quad \exists j \in \mathbb{N}.\, j > 0 \land \exists b', e' \in \text{Addr}.\, \exists \iota'_{malloc} \in \text{Region}. \\
\quad\quad\quad \Phi \rightarrow_j \Phi' \land \\
\quad\quad\quad \Phi'.\text{mem} = ms'_{footprint} \uplus ms_{alloc} \uplus ms_{frame} \land \\
\quad\quad\quad \iota'_{malloc} \sqsupseteq^{pub} \iota_{malloc} \land \\
\quad\quad\quad ms'_{footprint} :_{n-j} [i \mapsto \iota'_{malloc}] \land \\
\quad\quad\quad \text{dom}(ms_{alloc}) = [b', e'] \land \forall a \in [b', e'].\, ms_{alloc}(a) = 0 \land \\
\quad\quad\quad \Phi'.\text{reg} = \Phi.\text{reg}[\text{pc} \mapsto updPcPerm(w_{ret})] \\
\quad\quad\quad\quad\quad\quad\quad [r_1 \mapsto ((\textsc{rwx}, \text{global}), b', e', b')] \\
\quad\quad\quad\quad\quad\quad\quad [r_{t1}, r_{t2}, r_{t3} \mapsto 0, 0, 0] \land \\
\quad\quad\quad size - 1 = e' - b'
\end{array}$$

(d) For all $\Phi \in \text{ExecConf}$,

$$\begin{array}{l}
\text{If } (\Phi.\text{reg}(r_1) \notin \mathbb{Z} \lor \Phi.\text{reg}(r_1) < 0) \land \Phi.\text{reg}(\text{pc}) = updPcPerm(c_{malloc}) \\
\text{Then } \exists j \in \mathbb{N}.\, \Phi \rightarrow_j failed
\end{array}$$

We require a global capability for invoking malloc (because if the capability were local, then a program with access to malloc would have to give up this access when invoking untrusted code, 1). The capability is assumed to have enter permission (1), so malloc can protect its internal state even when the capability is shared with untrusted code.

In addition to these syntactic requirements, we also specify standard behavioural expectations of malloc. Intuitively, we require that when malloc is invoked with a length argument, then it will return a capability for a fresh piece of memory of that size. It should be fresh in the sense that malloc has not already allocated any of that memory and will not do so in the future. Also, when invoked with a nonsensical length argument such as a negative integer or a capability, malloc should simply fail. However, formulating these requirements is harder than one might expect. The problem is that a realistic implementation of malloc needs to rely on internal state (the free memory is part of malloc's internal state) that changes after every invocation and relies on invariants on that state. We can only expect that malloc behaves according to its specification if its internal state satisfies the current state of the invariants in an executing system. We express this in terms of the semantic model from Section 4.

To allow malloc implementations to rely on internal state and invariants for that state, we assume an initial region for malloc (2). The region is assumed to be permanent (since safety of the global malloc capability will depend on its presence, 2a). Furthermore, we want to express that malloc does not depend on any other memory than its own internal state. This is expressed by a restriction on the malloc region's state interpretation function which (as explained in Section 4.2) defines what memory segments it permits in a given world. We require that the accepted memory segments only depend on the presence of the malloc region itself, i.e. in any world the same memory segments are accepted if we remove all regions except the malloc region. This property should continue to hold throughout execution, so it must hold true for any private future region of the initial malloc region (2b).

The malloc specification also dictates what malloc should do when invoked in a memory with its internal state valid according to the malloc region (some future evolution of the initial malloc region). If malloc is invoked with an invalid length argument (that is, a negative integer or a capability), then we simply require malloc to fail (2d). This part of the specification does not actually rely on the malloc region: for simplicity, we assume malloc does not need its internal state to check the argument. When malloc is invoked with a valid length (2c), then it should return a fresh memory segment of the correct length. This segment is required to come from the footprint of malloc, i.e. the memory owned by the malloc region before the call. After malloc returns, we require the malloc region to have evolved (according to the public future world relation) to a new state where the new footprint is disjoint from the allocated memory. This implies that future invocations of malloc can never return previously-allocated memory.

For convenience, we also require that malloc returns the non-argument registers (except registers for malloc internal computations) of the register-file unchanged after the call. This allows the caller to keep private capabilities in the register file, without having to protect them by storing them in a private stack frame.

As described previously, the specification of malloc ensures that malloc has no capabilities pointing *out* of malloc. It does not, however, say anything about capabilities that point *in* to malloc. If we want to be able to trust malloc, we obviously cannot allow arbitrary capabilities to point in to it. We have chosen to keep the malloc specification simple and let this assumption be in the lemmas that use malloc. It is sufficient for these lemmas to require that there are no outside capabilities for malloc in the initial configuration as capabilities cannot appear out of thin air, and the malloc specification makes sure that capabilities are not leaked.

Malloc should not just be available to trusted programs but also to possibly malicious programs. This is safe as it follows from the specification that the malloc capability is always safe in a world with the malloc region:

Lemma 5.2 (Malloc is safe to pass to adversary). *For all c_{malloc} that satisfies the specification for malloc with region $\iota_{malloc,0}$, if $W(r) \sqsupseteq^{priv} \iota_{malloc,0}$, then $(n, c_{malloc}) \in \mathcal{V}(W)$ for all n.*

The reason we allow trusted code and adversary to invoke malloc is just to make our work more realistic, but we are otherwise not interested in its details. As such, we do not give a malloc implementation. We are, however, confident that it is possible to make an implementation of malloc that satisfies the malloc specification in Definition 5.1. There are in fact two simplifications in our system that makes things easier: First, we do not consider deallocation of memory which means that the data structure malloc uses to keep track of free memory does not have to handle reclaimed memory. Second, the malloc specification does not permit malloc to run out of memory and thus refuse allocation. This is possible on our simple capability machine because it has an infinite address space. An initial capability with an infinite range of authority would of course need to be part of malloc, but it could also double as the data structure that keeps track of free memory.

6 REUSABLE MACRO INSTRUCTIONS

With the calling convention and logical relation defined, we would like to show its usefulness by proving the correctness of a series of examples that rely on well-bracketedness and local-state encapsulation and use the calling convention to enforce these properties. However, the programs that run on our capability machine are assembly programs. Program examples that would be small in a high-level language become big and unintelligible at this low level. Thus to make our program examples intelligible, we introduce a series of low-level abstractions in the form of macros. We define a number of reusable macros capturing the calling convention as well as other conveniences. The macros that utilize the stack assume that it is available in register r_{stk}.

The macro `scall` $r(\overline{r_{args}}, \overline{r_{priv}})$ captures the parts of the calling convention related to actually transferring control to adversarial code. Specifically, it pushes the contents of the private registers, $\overline{r_{priv}}$, to the stack and pushes the “restoration” code to the stack. The restoration code is executed as the first thing upon return; it restores the stack pointer and the old program counter. After the restoration code is pushed to the stack, `scall` $r(\overline{r_{args}}, \overline{r_{priv}})$ adjusts the pc to point to the first instruction after the jump and pushes it to the stack. The `scall`macro constructs a protected return pointer from a stack pointer by adjusting it to point to the first instruction of the return code and encapsulating it by restricting its permission to E. Next, the `scall` macro reduces the range of authority of the stack pointer to the unused part of the stack and clears it (as discussed in Section 3). Finally, the `scall` macro clears non-argument registers and jumps to r. Upon return after the restoration code on the stack has been executed, the `scall` macro pops the restoration code from the stack and restores the private state by popping it from the stack to the appropriate registers. Figure 11 displays the implementation of `scall` and the restoration code used by `scall`. The implementation of `scall` uses some of the macros we present next.

The macro `mclear` r clears all the memory addresses that the capability in register r has authority over. It is used by `scall` to clear the unused part of the stack before control is transferred. It should also be used to clear the stack before returning to adversarial code. Similarly, the macro `rclear` R clears all the registers in the set R. It is also used by `scall`, and it should also be used before returning to adversarial code.

The macros `prepstack` r and `reqglob` r are the last macros related to the calling convention. The former, `prepstack` should be used when one receives a stack from an unknown source. The `prepstack` macro first ensures the stack capability has permission read/write-local/execute. Then, it adjusts the stack capability, so it follows the convention for the stack[9]. The other macro, `reqglob`, ensures that the capability in register r is global. This macro should be used on callbacks received from unknown sources to ensure that they cannot be derived from the stack pointer.

The remaining macros are not directly related to the calling convention, but they help making the examples in Section 8 intelligible. The macros `push` r and `pop` r respectively add and remove elements from the stack. The macro `fetch` r *name* fetches the capability related to *name* from the linking table and stores it in register r. The macro `malloc` r n invokes malloc with size argument n. The `malloc` macro assumes that a capability for malloc resides in the linking table and is basically

[9] The stack capability should always point to the topmost word on the stack. A stack received from an unknown source can be treated as empty, so the stack capability should point just outside its range of authority.

```
// Push the private registers to the stack.
  push $r_{priv,1}$
  ...
  push $r_{priv,n}$
// Push the restoration code to the stack.
  push $encode(i_1)$
  push $encode(i_2)$
  push $encode(i_3)$
  push $encode(i_4)$
// Push the old pc to the stack.
  move r_t1 pc
  lea r_t1 $off$
  push r_t1
// Push the stack pointer to the stack.
  push r_stk
// Set up the protected return pointer.
  move r_0 r_stk
  lea r_0 $off_{rec}$
  restrict r_0 $encodePermPair((\text{local}, \text{e}))$
// Restrict the stack capability to the unused part of the stack.
  geta r_t1 r_stk
  plus r_t1 r_t1 1
  getb r_t2 r_stk
  subseg r_stk r_t1 r_t2
// Clear the unused part of the stack.
  mclear r_stk
// Clear non-argument registers.
  rclear $R$
  jmp r
return:
// Pop the restore code.
  pop r_t1
  pop r_t1
  pop r_t1
  pop r_t1
// Pop the private state into approriate registers.
  pop $r_{priv,1}$
  ...
  pop $r_{priv,n}$
```

```
$i_1$ = move r_t1 pc
$i_2$ = lea r_t1 $off_{stk}$
$i_3$ = load r_stk r_t1
$i_4$ = pop pc
```

The restoration code used in scall.

Fig. 11. Implementation of scall $r(r_{args,1}, \ldots, r_{args,m}, r_{priv,1}, \ldots, r_{priv,n})$. The restoration code is presented in the top right corner. The variable off_{ret} is the offset to the label return, and $off_{rec} = -5$ which is the offset to the first instruction of the activation record. The set $R = \text{RegName} \setminus \{\text{pc}, \text{r\_stk}, \text{r\_0}, r, r_{args,1}, \ldots, r_{args,m}\}$. The variable $off_{stk} = 5$ which is the offset to the address where the old stack pointer is stored on the stack.

a fetch of the malloc capability followed by a setup of a return pointer. Finally, the macro crtcls $\overline{(x_i, r_i)}$ r allocates a closure where r points to the closure's code and a new environment is allocated (using malloc) and the contents of $\overline{r_i}$ are stored in the environment. In the code referred to by r, an implicit load from the environment happens when an instruction refers to x_i.

The Appendix contains the implementation of all the macros used in `scall`. The technical appendix [Skorstengaard et al. 2019a] contains more detailed descriptions of all the macros as well as all implementations. We stress that the macros correspond to series of instructions as seen in Figure 11; the macros are introduced for intelligibility. The examples of Section 8, the program examples are stated using the macros, but the proofs work on the expanded macros.

7 REASONING ABOUT PROGRAMS ON A CAPABILITY MACHINE

There are many details to get right when programming in assembly. These details carry over to proofs about assembly programs, so many of the proofs in Section 8 about example programs are a bit cumbersome. It is especially annoying when the same line of reasoning is applied in multiple places. To mitigate this, we have defined a number of lemmas that capture common reasoning patterns in these proofs. In this section, we present the most central lemmas used to reason about assembly programs.

Our capability machine only allows the final memory of an execution to be observed, so naturally the correctness lemmas we prove in Section 8 are statements about the memory in the halted configuration. In order to prove a property about part of the final memory, we create a world with a region the ensures the desired property and show that the initial configuration is in the $\mathcal{O}$-relation w.r.t. that world. The region that ensures the property must be permanent, so it is not revoked during execution. The $\mathcal{O}$-relation says that if the initial configuration halts successfully, then the memory in the halting state must still respect a private future world of the initial world. It is, however, bothersome and error prone to try to argue about the entire execution in one go. Instead we want to reason modularly in the sense that we only want to reason about parts of the execution at a time. To this end, we prove an anti-reduction lemma that essentially says that if we can show that an initial configuration Φ steps to a configuration Φ' and Φ' is in the $\mathcal{O}$-relation, then Φ must also be in the observation relation.

Lemma 7.1 (Anti-reduction for $\mathcal{O}$).

$$\begin{aligned}
&\forall n, n', i, reg, reg', ms, ms', ms_r, W, W'.\\
&\quad n' \geq n - i \wedge W' \sqsupseteq^{priv} W \wedge\\
&\quad (\forall ms_f.\,(reg, ms \uplus ms_r \uplus ms_f) \rightarrow_i (reg', ms' \uplus ms_r \uplus ms_f)) \wedge\\
&\quad (n', (reg', ms')) \in \mathcal{O}(W')\\
&\qquad \Rightarrow (n, (reg, ms \uplus ms_r)) \in \mathcal{O}(W)
\end{aligned}$$

We use the anti-reduction lemma when we reason about the execution of known code. When we want to reason about unknown code, the anti-reduction lemma does not apply because we do not know what instructions are being executed. This is where the FTLR (Theorem 4.4) comes into play. As a reminder, the FTLR says that if a capability only has access to safe values with respect to a world, then it is safe to use the capability for execution in the same world. The unknown code we consider will be assumed to only have access to safe values which typically means it has access to a linking table with a malloc capability and otherwise consists of instructions. These assumptions allow us to use the FTLR to reason about the unknown code as malloc is a safe value (cf. Lemma 5.2) and instructions are integers; integers are always safe values. It is important to remember that safety is relatively to the semantic model of a memory and not the actual memory itself. For this reason, the assumptions we make on unknown code must be expressed as a region in a world.

In order to argue that a specific configuration is safe, we need to argue that the configuration is safe with respect to the world. That is, we need to argue that the memory satisfies the world and the register-file is in the $\mathcal{R}$-relation. For instance in the case where the untrusted code assumes control first, we will use the FTLR to show that the capability for the unknown code can be used for execution. The unknown code will get access to some known, trusted code through a capability in the initial register-file, so we will have to argue that the known code is safe. We argue about the known code with Lemma 7.1.

Another common pattern in proofs about programs on our capability machine concerns the use of `scall`. The following lemma captures the commonalities of reasoning about programs using `scall`. For instance, setting up the local stack frame and constructing stack pointers and protected return pointers for the callee always amount to the same line of reasoning which is captured by the lemma. From another point of view, it can be seen as a specification for `scall`.

Lemma 7.2 (`scall` works). *For all* $n \in \mathbb{N}$, $ms, ms_{stk}, ms_{unused}, ms_f \in \mathrm{MemSeg}$, $W \in \mathrm{World}$, $reg \in \mathrm{Reg}$, $r, \overline{r_{arg}}, \overline{r_{priv}} \in \mathrm{RegName}$, *and* $c_{next} \in \mathrm{Cap}$, *if*

(1) $ms :_n revokeTemp(W)$

(2) $\mathrm{dom}(ms_f) \cap (\mathrm{dom}(ms_{stk} \uplus ms_{unused} \uplus ms)) = \emptyset$

(3) (reg, ms) *is looking at* `scall` $r(\overline{r_{arg}}, \overline{r_{priv}})$ *followed by* c_{next}

(4) reg *points to stack with* ms_{stk} *used and* ms_{unused} *unused*

(5) *Hyp-Callee For all* $ms_{rec}, ms'_{unused}, ms'' \in \mathrm{MemSeg}$, $W' \in \mathrm{World}$, $c_{ret}, c_{stk} \in \mathrm{Cap}$, *and* $reg' \in \mathrm{Reg}$, *if*
- $\mathrm{dom}(ms_{unused}) = \mathrm{dom}(ms_{rec} \uplus ms'_{unused})$,
- $W' = revokeTemp(W)[\iota^{sta}(\mathrm{temp}, ms_{stk} \uplus ms_{rec} \uplus ms_f), \iota^{pwl}(\mathrm{dom}(ms'_{unused}))]$[10],
- $ms'' :_{n-1} W'$
- reg' *points to stack with* $\emptyset$ *used and* ms'_{unused} *unused*
- $reg' = reg_0[\mathrm{pc} \mapsto updPcPerm(reg(r)), \overline{r_{arg}} \mapsto reg(\overline{r_{arg}}), r_0 \mapsto c_{ret}, r_{stk} \mapsto c_{stk}, r \mapsto reg(r)]$
- $(n-1, c_{ret}) \in \mathcal{V}(W')$
- $(n-1, c_{stk}) \in \mathcal{V}(W')$

then we have $(n-1, (reg', ms'')) \in \mathcal{O}(W')$

(6) *Hyp-Cont For all* $n' \in \mathbb{N}$, $W'' \in \mathrm{World}$, ms'', ms''_{unused}, *and* $reg' \in \mathrm{Reg}$, *if*
- $n' \leq n-2$
- $W'' \sqsupseteq^{pub} revokeTemp(W)$
- $ms'' :_{n'} revokeTemp(W'')$
- *for all* r, *we have:*

$$reg'(r) \begin{cases} = c_{next} & \text{if } r = \mathrm{pc} \\ = reg(r) & \text{if } r \in \overline{r_{priv}} \\ \in \mathcal{V}(revokeTemp(W'')) & \text{if } reg'(r) \text{ is a global capability and } r \notin \{\mathrm{pc}, \overline{r_{priv}}, r_{stk}\} \end{cases}$$

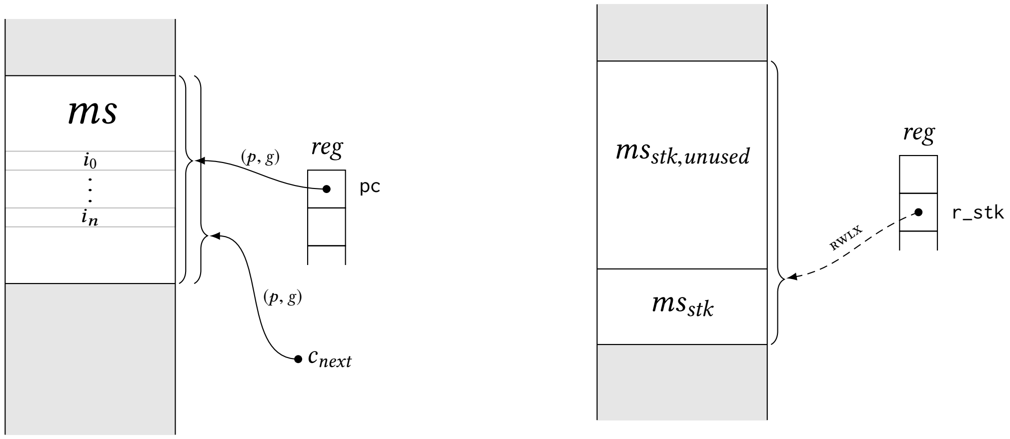
- reg' *points to stack with* ms_{stk} *used and* ms''_{unused} *unused*

then we have $\left(n', (reg', ms'' \uplus ms_f \uplus ms_{stk} \uplus ms''_{unused})\right) \in \mathcal{O}(W'')$

Then
- $\left(n, (reg, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{unused})\right) \in \mathcal{O}(W)$

The `scall` lemma is stated in terms of a number of auxiliary definitions found in the appendix. The region $\iota^{sta}(g, ms)$ is a static region with locality g. It is static in the sense that it only accepts the memory segment ms (Appendix A.7). The function $revokeTemp : \mathrm{World} \to \mathrm{World}$ yields the input world but with all the temporary regions replaced with revoked regions (Appendix A.1). The definition of (reg, ms) *is looking at* $[i_0 \ldots i_n]$ *followed by* c_{next} is visualized in Figure 12a; it requires the capability in pc of reg to point to the first address of the instructions $[i_0 \ldots i_n]$ in ms and c_{next}

[10] We use the update-notation without a region name to indicate that it should be a fresh region name.



(a) Visualization of "(reg, ms) is looking at $[i_0 \dots i_n]$ followed by c_{next}". $reg(\mathrm{pc})$ points to i_0 and c_{next} points to the address after i_n

(b) Visualization of "*reg* points to stack with ms_{stk} used and $ms_{stk,unused}$ unused". $reg(r_{stk})$ is a local RWLX-capability that points to the top most address of ms_{stk}.

Fig. 12

to point to the address immediately after the instructions. The definition of *reg points to stack with* ms_{stk} *used and* $ms_{stk,unused}$ *unused* is visualized in Figure 12b; it requires ms_{stk} and $ms_{stk,unused}$ to be adjacent and continuous memory segments with the local RWLX-capability in `r_stk` of *reg* governing the two memory segments and pointing to the top most address of ms_{stk} (Appendix A.5).

Roughly, the `scall` lemma (Lemma 7.2) states that an invocation of scall is safe if `scall` is executed in a reasonable state (1-3), the callee is safe to execute (5), and returning to the code after the scall is safe (6).

In the common case, `scall` is used to reason about untrusted code that we only have general assumptions about. As explained previously, we use the FTLR (Theorem 4.4) to argue about unknown code; this is no exception. That is, when we argue that assumption 5 in the `scall` lemma is satisfied, we use the FTLR along with the assumptions we have (e.g. linking table, malloc capability etc.). In Hyp-Callee, we assume the memory is safe, so it suffices to show that the register-file contains safe values, which amounts to showing that the arguments in the call are safe.

By using Lemma 7.2 to reason about `scall`, the proofs become agnostic to the actual implementation of `scall`. In other words, should we change the implementation of `scall`, then we just need to prove that Lemma 7.2 holds for the new implementation in order to get that all our results still hold true.

The `malloc` macro is also common in our examples, so we prove Lemma 7.3 to help reason about it. The structure of the `malloc` lemma is close to that of the `scall` lemma. It also has requirements on the configuration just before `malloc` is executed (1-8) as well as requirements on the execution afterwards (9). It does not have any requirements on the callee as the `malloc` specification defines its behavior.

Lemma 7.3 (`malloc` works). *If*

(1) (reg, ms) *is looking at* `malloc` r k *followed by* c_{next}
(2) $k \geq 0$
(3) (reg, ms) *links* $malloc$ *as* k *to* c_{malloc}
(4) c_{malloc} *satisfies the malloc specification with* $\iota_{malloc,0}$
(5) $W \sqsupseteq^{priv} [i \mapsto \iota_{malloc,0}]$
(6) $ms :_n W$
(7) $ms = ms' \uplus ms_{footprint}$
(8) $ms_{footprint} :_n [i \mapsto W(i)]$
(9) *Hyp-Cont If*

- $n' \leq n - 1$
- $\iota_{malloc} \sqsupseteq^{pub} W(i)$
- $ms'_{footprint} \uplus ms' :_{n'} W[i \mapsto \iota_{malloc}]$
- $ms'_{footprint} :_{n'} [i \mapsto \iota_{malloc}]$

$$reg'(r') = \begin{cases} c_{next} & r' = \text{pc} \\ ((\textsc{rwx}, \text{global}), b, e, a) & r' = r \\ reg(r) & r' \notin \text{RegName}_t \cup \{\text{pc}, r, r_1\} \end{cases}$$

- $e - b = k - 1$
- $\text{dom}(ms_{alloc}) = [b, e]$
- $\forall a \in [b, e].\ ms_{alloc}(a) = 0$

Then we have $\left(n', (reg', ms' \uplus ms'_{footprint} \uplus ms_{alloc})\right) \in \mathcal{O}(W[\iota_{malloc}])$

Then

$$(n, (reg, ms)) \in \mathcal{O}(W)$$

In the technical appendix [Skorstengaard et al. 2019a], we also define a lemma for the macro used to create closures, `crtcls`.

8 EXAMPLES

In this section, we demonstrate how our formalization of capability safety allows us to prove local-state encapsulation and control-flow correctness properties for challenging program examples. The security measures of Section 3 are deployed to ensure these properties. Since we are dealing with assembly language, there are many details to the formal treatment. Therefore, we omit some details in the lemma statements. The examples may look deceivingly short, but it is because we use the macro instructions described in Section 6. The examples would be unintelligible without the macros as each macro expands to multiple basic instructions.

8.1 Encapsulation of Local State

The programs `f1` and `f2` in Figure 13 demonstrate the capability machine's encapsulation of local state. They are very similar: both store some local state, call an untrusted piece of code (*adv*), and test whether the local state is unchanged after the call. They differ in the way they do this. Program `f1` uses our stack-based calling convention (captured by `scall`) to call the adversary, so it can use the available stack to store its local state. On the other hand, `f2` uses malloc to allocate memory for its local state and uses `call` a calling convention based on heap allocated activation records (described in Appendix A.4) to invoke the adversarial code.

For both programs, we prove that if they are linked with an adversary, *adv*, allowed to allocate memory but has no other capabilities, then the assertion will never fail during execution (see

```
f1: push 1
    fetch r1 adv
    scall r1([],[])
    pop r1
    assert r1 1
    halt
```

```
f2: malloc rl 1
    store rl 1
    fetch r1 adv
    call r1([],[rl])
    assert rl 1
    halt
```

Fig. 13. Two example programs that rely on local-state encapsulation. f1 uses our stack-based calling convention. f2 does not rely on a stack.

Lemmas 8.1 and 8.2 below). The two examples also illustrate the versatility of the logical relation. The logical relation is not specific to any calling convention, so we can use it to reason about both programs even though they use different calling conventions.

In order to formulate results about f1 and f2, we need a way to observe whether the assertion fails. To this end, we assume they have access to a flag (an address in memory). If the assertion fails, then the flag is set to 1 and execution halts.

Lemma 8.1 (f1 is correct). *Let*

$$\begin{aligned} c_{adv} &\overset{def}{=} ((\textsc{e}, \text{global}), \dots) & c_{stk} &\overset{def}{=} ((\textsc{rwlx}, \text{local}), \dots) \\ c_{f1} &\overset{def}{=} ((\textsc{rwx}, \text{global}), \dots) & c_{link} &\overset{def}{=} ((\textsc{ro}, \text{global}), \dots) \\ c_{malloc} &\overset{def}{=} ((\textsc{e}, \text{global}), \dots) & reg &\in \text{Reg} \end{aligned}$$

$$m \overset{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}$$

where each of the capabilities have an appropriate range of authority and pointer[11]*. Furthermore*

- c_{malloc} *satisfies the specification for malloc with* $\iota_{malloc,0}$
- $ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]$
- ms_{f1} *contains* c_{link}, c_{flag} *and the code of* f1
- $ms_{flag}(flag) = 0$
- ms_{link} *contains* c_{adv} *and* c_{malloc}
- ms_{adv} *contains* c_{link} *and otherwise only instructions.*

If $(reg[\text{pc} \mapsto c_{f1}][r_{stk} \mapsto c_{stk}], m) \rightarrow^* (halted, m')$*, then* $m'(flag) = 0$

To prove Lemma 8.1, it suffices to show that the start configuration is safe (in the $\mathcal{O}$ relation) for a world with a permanent region that requires the assertion flag to be 0. By Lemma 7.1, it suffices to show that the configuration is safe after some reduction steps. We then use the scall lemma (Lemma 7.2) by which it suffices to show that (1) the configuration that scall will jump to is safe and (2) the configuration just after scall is done cleaning up is safe. We use the Fundamental Theorem to reason about the unknown adversarial code as described in Section 7, but notice that the adversary capability is an enter capability, which the Fundamental Theorem says nothing about. Luckily, the enter capability has rx-permission after the jump at which point the Fundamental Theorem applies.

We have a similar lemma for f2:

Lemma 8.2 (f2 is correct). *Making similar assumptions about capabilities and linking as in Lemma 8.1 but assuming no stack pointer and assuming* c_{f2} *points to* f2*, if* $(reg[\text{pc} \mapsto c_{f2}], m) \rightarrow^* (halted, m')$*, then* $m'(flag) = 0$.

[11] These assumptions are kept intentionally vague for brevity. Full statements are in the Appendix.

```
1  f3: push 1
2       fetch r1 adv
3       scall r1([],[r1])
4       pop r2
5       assert r2 1
6       push 2
7       scall r1([],[])
8       halt
```

```
1  g1:  malloc r2 1
2       store r2 0
3       move pc r3
4       lea r3 off
5       crtcls [(x, r2)] r3
6       rclear RegName \ {pc, r0, r1}
7       jmp r0
8  f4:  reqglob r1
9       prepstk rstk
10      (continues in next column)
11      (continued from previous column)
12      store x 0
13      scall r1([],[r0, r1, renv])
14      store x 1
15      scall r1([],[r0, renv])
16      load r1 x
17      assert r1 1
18      mclear rstk
19      rclear RegName \ {r0, pc}
20      jmp r0
```

Fig. 14. Two programs that rely on well-bracketedness of scalls to function correctly. *off* is the offset to f4.

8.2 Well-Bracketed Control-Flow

The stack-based calling convention scall ensures well-bracketed control-flow. This is illustrated by program examples f3 and g1 in Figure 14.

The program f3 has two calls to an adversary. In order for the assertion on line 5 to succeed, the calls must be well-bracketed. If the adversary were able to store the return pointer from the first call and invoke it in the second call, then f3 would have 2 on top of its stack and the assertion would fail. However, the security measures in Section 3 prevent this attack. Specifically, the return pointer is local, so it can only be stored on the stack. However, the part of the stack that is accessible to the adversary is cleared before the second invocation preventing attempts to store the return pointer. In fact, the following lemma shows that there are also no other attacks that can break well-bracketedness of this example, i.e. the assertion never fails. It is similar to the two previous lemmas:

Lemma 8.3. *Making similar assumptions about capabilities and linking as in Lemma 8.1 and assuming* c_{f3} *points to* f3 *if* $(reg[\mathrm{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \longrightarrow^* (halted, m')$*, then* $m'(flag) = 0$.

The final example, g1 with f4, is a faithful translation of a tricky example known from the literature (known as the awkward example) [Dreyer et al. 2012; Pitts and Stark 1998]. For comparison, we show a version of the original example in Figure 15, highlighting the code locations corresponding to g1 and f4.

```
1       fun _ =>
2  g1:   let x = ref 0 in
3        fun adv =>
4  f4:     x := 0;
5          adv();
6          x := 1;
7          adv();
8          assert(x == 1)
```

Fig. 15. The original awkward example from Dreyer et al. [2012]; Pitts and Stark [1998], in ML notation.

Let us first look at this ML program. At the top-level, it is a lambda function that can be invoked by the context. When it is invoked, it allocates a fresh mutable variable x of integer type that initially contains the value 0. Next, the function returns a second closure (let's call this cl_x) that can be invoked by the context whenever it chooses. cl_x itself takes a callback function adv that it will invoke twice after setting x to 0 and 1 respectively. After the second invocation of adv, cl_x will verify that x is still set to 1.

The assertion on line 8 of Figure 15 should never fail. From the code, this seems natural since adv does not have access to x. Therefore after the second invocation on line 7, x should still be in the state it was before that invocation. However, adv could have access to cl_x and could invoke cl_x again within the second invocation of adv. Then cl_x would set x to 0 again (on line 5) and reinvoke another callback. If that second callback were somehow able to skip its own caller and return directly to the caller of adv, it would end up on line 8 of Figure 15 with x set to 0, causing the assertion to fail. Without going into details, such an attack is perfectly possible if adv has access to a call/cc primitive (or equivalent). In other words, if we are able to prove that the assertion will never fail, this attack, and other similar attacks, on well-bracketed control flow are adequately prevented.

Our low-level version of the awkward example in Figure 14, consists of two parts, g1 and f4, corresponding to the code locations marked in Figure 15. The program g1 corresponds to the top-level closure in the ML code. It is a closure generator that generates closures with a mutable variable x set to 0 in its environment and f4 as the program (note that we omit some calling convention security measures because the stack is not used in the closure generator).

The program f4 expects one argument, the callback adv, in r1. f4 sets x to 0 and invokes adv. When it returns, f4 sets x to 1 and calls adv again. When it returns the second time, f4 asserts x is still 1 and returns.

This example is more complicated than the previous ones because it involves a closure invoked by the adversary and an adversary callback invoked by us. As explained in Section 3, we need to check that (1) the stack pointer that the closure receives from the adversary has write-local permission and (2) the adversary callback is global.

The attack that we explained above is actually more natural at this low-level. The callback adv now gets a return capability. As explained, it could invoke cl_x again, during its second invocation, with a second callback function adv′ for cl_x to invoke. If adv had some way of passing its return capability to adv′ (by storing it on the stack or in the heap, or hide it in an unused register), then the assertion could be made to fail. However, our calling convention prevents any of this from happening as we prove in the following lemma.

Lemma 8.4. *Let*

$$c_{adv} \overset{def}{=} ((\textsc{rwx}, \text{global}), \dots) \quad c_{g1} \overset{def}{=} ((\textsc{e}, \text{global}), \dots)$$

and otherwise make assumptions about capabilities and linking similar to Lemma 8.1. Then if $(reg_0[\mathrm{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \to^* (halted, m')$, *then* $m'(flag) = 0$.

Proof sketch. Define a world W_1 with the following regions

- A malloc region, $\iota_{malloc,0}$.
- A permanent region for the linking table that only accepts ms_{link} and requires everything to be in $\mathcal{V}$.
- A stack region, $\iota^{pwl}_{b_{stk},e_{stk}}$.
- An adversary region, $\iota^{nwl,p}_{b_{adv},e_{adv}}$
- A permanent static region for the flag and g1, i.e. a region that only accepts ms_{flag} and ms_{g1}.

If we can show

$$(reg_0[\mathrm{pc} \mapsto c_{adv}, r_{stk} \mapsto c_{stk}, r_1 \mapsto c_{g1}], ms_{malloc} \uplus ms_{link} \uplus ms_{stk} \uplus ms_{adv} \uplus ms_{flag} \uplus ms_{g1}) \in \mathcal{O}(W_1), \tag{1}$$

[12]then the $\mathcal{O}$-relation ensures that a successfully halting configuration terminates in a memory that respects a private future world of W which in particular means that it respects the permanent static region that governs the assertion flag.

As described in Section 7, we use the FTLR (Theorem 4.4) to reason about unknown code, so we use it to reason about c_{adv}. With the standard region $\iota^{nwl,p}$ chosen for the adversary, it is easy to show that the *readCond* and *writeCond* holds for c_{adv} which gives us $c_{adv} \in \mathcal{E}(W)$ by the FTLR. In order to get (1), we need to show that (a) the memory satisfies the world and (b) the register file is in the $\mathcal{R}(W)$. We have defined the world, so there (almost) is a one-to-one correspondence between

[12] We ignore step indexes in this proof sketch.

regions in the world and memory segments in the initial configuration, so (a) easily follows. In order to show (b), we need to show $c_{stk} \in \mathcal{V}(W)$ and $c_{g1} \in \mathcal{V}(W)$. The former follows easily from the choice of a ι^{pwl}-region for the stack in W. In order to argue the latter, we basically have to argue that c_{g1}, the closure generator, respects the world W. This amounts to showing that the closures generated by `g1` also respect the invariants of W. We reason about the local variable x in the closure in the same way Dreyer et al. [2012] does. We ignore x in the remainder of this proof sketch to focus on the parts of the proof related to the setting of the capability machine.

The capability for the generated closure is a global enter capability that we call c_{f4}. The remainder of the proof amounts to showing that it is safe to return c_{f4} to the adversary, i.e. $c_{f4} \in \mathcal{V}(W')$ where W' is W with relevant regions added after executing `g1`. The adversary can use c_{f4} whenever, so all we may assume about the configuration that c_{f4} is invoked in is that the register-file *reg* and memory *ms* satisfies a world W_1 that is a private future world of W', i.e. $reg \in \mathcal{R}(W_1)$ and $ms : W_1$. We have to show that the invocation respects W_1 which corresponds to showing $(reg[\mathrm{pc} \mapsto updPcPerm(c_{f4})], ms) \in \mathcal{O}(W_1)$. When c_{f4} is invoked, we know exactly what instructions are executed up until the `scall`, namely it is ensured that the callback is global and the stack pointer has read/write-local/execute permission, and x is set to 0. Because we know part of the execution, we can apply the anti-reduction lemma (Lemma 7.1).

The next part in the execution is an `scall`, so, according to Section 7, we should apply the `scall` lemma (Lemma 7.2). For the sake of presentation, we here sketch some of the things the `scall` lemma actually takes care of. Based on the stack pointer's permission, we know by Lemma 4.3 that the capability is local (because it is a system wide assumption that there are no global read/write-local/execute capabilities *cf.* Section 3) which means that the region that governs it must be temporary. This allows us to construct a world $W_2 \sqsupseteq^{priv} W_1$ where the stack region of W_1 has been revoked. In W_2, two new regions govern the stack. One of the two new regions governs the caller stack frame, i.e. the part of the stack that contains the contents of private registers and stack recovery code. The region that takes care of this is a temporary static region which ensures that our local stack frame remains unchanged during the execution of the callback. The other new region takes care of the unused part of the stack which we are going to let the callback use for its execution. This is taken care of by a standard region ι^{pwl}-region which allows the callback to store any safe value on the stack. The callback is global, so it is safe to invoke it in a private future world of W_1 (even though the code is unknown, we do not need to use the FTLR because we assume that the arguments for the invocation of c_{f4} are safe). Notice that had the callback been local, this would not be the case, but it would also not be safe to invoke as it might be a stack pointer as discussed in Section 3. Note also that arguing that the memory satisfies W_2 when we invoke the callback only works because we cleared the stack entirely (including the unused part) before the invocation. Otherwise, it might contain local values that are only known to be safe in W_1, but for which we do not know that they will remain safe in the private future world W_2. This part corresponds to arguing about Hyp-Callee in the `scall` lemma.

We need to argue that it is safe to give the return pointer that `scall` constructs to the adversary which corresponds to reasoning about the remainder of the execution of `f4` (corresponding to the Hyp-Cont case of the `scall` lemma). The return pointer is a local capability, so we may assume that it is invoked in some configuration that satisfies $W_3 \sqsupseteq^{pub} W_2$. This means that none of the temporary regions have been revoked, so the regions we replaced the old stack region with are still present in W_3. We know exactly how the execution proceeds upon return (the recovery code is executed, and x is set to 1). We use the anti-reduction lemma to reason about the execution until the second `scall`.

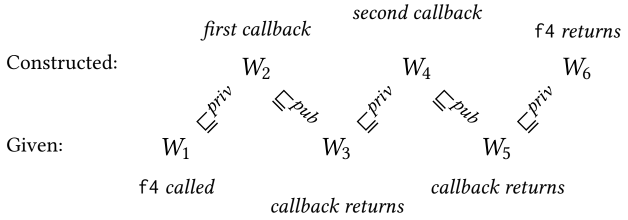


Fig. 16. Illustration of the worlds in the proof of Lemma 8.4. In the proof, the top row of worlds are constructed by us, while the bottom row of worlds are given. W_6 is constructed such that $W_6 \sqsupseteq^{priv} W_5$ and $W_6 \sqsupseteq^{pub} W_1$.

For the `scall` itself we apply the `scall` lemma. As the private stack has changed, we need to replace the two regions that govern the stack. This means that the second invocation of the callback takes place in a world $W_4 \sqsupseteq^{priv} W_3$. The reasoning about `scall` is similar to the first callback invocation. The callback capability is still safe because it is global which basically covers the Hyp-Callee case. For the Hyp-Cont case, we get a world $W_5 \sqsupseteq^{pub} W_4$ in which we need to argue that the remainder of the execution is safe. At this point, we can use the anti-reduction lemma one final time to get to the point where we jump to the return pointer.

If we can argue that it is safe to jump to the return pointer that we got initially from the adversary, then the proof is done. We have made no checks on the return pointer, so we have no idea whether it is local or global[13]. W.l.o.g. assume the return pointer is local. This means that it is safe in public future worlds of W_1 (remember that it came from *reg* and $reg \in \mathcal{R}(W_1)$). Hence we need to construct a world W_6 with all the temporary and revoked regions of W_1 (this corresponds to restoring the invariants the adversary relies on for its safe execution). Further, for W_6 to be a public future world, for all these regions, W_6 must use the same region names as W_1. As we used the anti-reduction lemma to get to this point, W_6 must also be constructed to be a private future world of W_5. Before we construct W_6, let us consider where each of the worlds we have seen so far came from: As illustrated in Figure 16, we constructed W_2 and W_4. These are the only private future worlds we have considered so far. This means that we know exactly what changes they made. In particular, we did not mask any of the temporary or revoked regions in W_1 with a permanent region. This means that in W_6, we can reinstate every temporary or revoked region of W_1. In the private future worlds we constructed, we only took transitions that are public relative to W_1. The worlds we were given, W_3 and W_5, may have taken arbitrary public transitions, but this is no problem with respect to the public future world relation. In W_5 there were regions to handle the stack. These regions need to be revoked as the W_1 stack regions replace them. The non-stack regions in W_5 that are not present in W_1 are also added in W_6, which the public future world relation permits as it is extensional. All in all, it is possible to construct W_6, so it is both a public future world of W_1 and a private world of W_5 which means that it is indeed safe to return to the adversary.

For the sake of presentation, we have omitted many details and made several simplifications in the above proof. The complete proof can be found in the technical appendix [Skorstengaard et al. 2019a]. □

[13] We may assume that it is a capability that is executable when jumped to since otherwise the execution fails which is considered safe.

9 DISCUSSION

Calling convention

Formulating control flow correctness While we claim that our calling convention enforces control-flow correctness, we do not prove a general theorem that shows this because it is not clear what such a theorem should look like. Formulations in terms of a control-flow graph, like the one by Abadi et al. [2005], creates a view with all capabilities that may be available at some point in time. Hence control-flow-graph based formulations lead one to consider capabilities that are actually not available at a given point in time. Example g1 relies on a more fine-grained view of the control flow, in particular when returning from the higher-order callbacks. In fact, our examples show that our logical relation implies a stronger form of control-flow correctness than control-flow-graph based formulations, although this is not made very explicit. In later work, Skorstengaard et al. [2019b] provide a more explicit and useful definition of control-flow correctness and local state encapsulation. The idea of their definition is to define a variant of a simple capability machine with a built-in stack as well as call and return instructions. The call and return instructions use the built-in stack which gives well-bracketed control and local state encapsulation by construction. They prove full abstraction [Abadi 1998] for a compilation from the capability machine with the built-in stack to one without which means that the programs running on the capability machine without a built-in stack behave as though there was a built-in stack. Their full abstraction proof uses a logical relation, but it differs in a number of ways from the one we present. Their logical relation is a binary cross language logical relation tailored specifically to prove the full abstraction result. This means that it could not be used to prove program correctness results similar to ours. Their calling convention is based on something called linear capabilities rather than local capabilities. Linear capabilities offer a different kind of limited revocation.

Performance and the requirement for stack clearing The additional security measures of the calling convention described in Section 3 impose an overhead compared to a calling convention without security guarantees. However, most require only a few atomic checks or register clearings on boundary crossings between trusted code and adversary, which should produce an acceptable performance overhead. The only exception is the stack clearing requirement that we have in two situations: when returning to the adversary and when invoking an adversary callback. As we have explained, we need to clear all of the stack that we are not using ourselves not just the part that we have actually used. In other words on every boundary cross between trusted code and adversary code, a potentially large region of memory must be cleared.

First, contrary to what we explained before, we actually suspect that this overhead can be avoided when returning to the adversary. In that case, we now think it would suffice to clear only the (much smaller) part of the stack used by the trusted code itself. To understand this, it is useful to take another look at the illustrations in Figure 4 related to this case. If we do not clear the stack upon return (as illustrated in Figure 4f), then the adversary might have used that stack to store local capabilities they received in a previous invocation (see Figure 4e). In other words, the stack clearing is necessary for revoking those local capabilities: the stack pointer and return pointer for the invocation illustrated in Figure 4e. While this approach is safe, we now suspect that we could do without the revocation in this case. The stack pointer which the adversary was given access to in the stack frame depicted in Figure 4d only carries authority that the adversary has access to in the higher stack frame anyway. Similarly, the return pointer is merely an entry pointer pointing into the trusted code's stack frame, and this is also a capability that the adversary we return to could have constructed themselves from their stack pointer.

This improvement of our calling convention is a recent insight and not yet reflected in our examples and our proofs. We do believe the proofs could be updated to accomodate for this change, but it would require some non-trivial changes. Consider the awkward example proof, for example, we would have to ensure that the world W_6, depicted in Figure 16 would not just be a private future world of W_5, but a public one (in addition to being a public future world of W_1). This would allow us to do without the stack clearing, but it would entail some changes to the regions we use, and an extra argument that the old adversary's stack and return pointer remain valid after clearing the trusted code's stack frame and relaxing the invariant that used to ensure it could not be modified. While this change is a clear improvement, we do not actually believe that it fundamentally changes the efficiency characteristics of the approach: the cost is halved, but remains asymptotically the same.

A second important remark we want to make is that the need for stack clearing in our calling convention is an instance of a general caveat when using CHERI's local capabilities as a restricted form of capability revocation. Consider how our use of local capabilities can be interpreted as temporarily delegating the stack and return capability to callees and then revoking the granted authority after the callee returns. From this perspective, local capabilities are a general feature enabling this temporary delegation of authority for the duration of an invocation and this is also how they are described by the authors [N. M. Watson et al. 2015]. However, our requirement for stack clearing on boundary crossings is also general. Revoking authority that was granted temporarily using local capabilities requires clearing all memory for which the invokee had write-local authority (or at least erasing all local capabilities from that memory). Without micro-architectural support for efficiently clearing large ranges of memory, local capabilities can only be used for revocation in scenarios where the duration of a revocation is unimportant or the adversary only has write-local access to small amounts of memory.

CheriBSD's use of local capabilities in CCall does not actually involve a form of revocation. CheriBSD's model involves a trusted stack manager that gives every compartment access to its own private stack using a local, write-local capability [N. M. Watson et al. 2015]. The locality of the stack capability allows the trusted stack manager to prevent compartments from leaking their stack pointer in a boundary crossing, but those capabilities are never actually revoked. In fact, a compartment can easily store away such local capabilities in its own private stack and recover them there during future invocations.

Since local capabilities seem intended to provide a restricted form of revocation, perhaps capability machines like CHERI should consider to provide special support for this requirement. Ideally, such support would take the form of a highly-optimized instruction for erasing a large block of memory. Recent work suggests that such a feature could perhaps be added to processors like CHERI, using a special hardware cache that tracks whether or not a memory location contains zero [Joannou et al. 2017].

Modularity It is important that our calling convention is modular, i.e. we do not assume that our code is specially privileged w.r.t. the adversary, and they can apply the same measures to protect themselves from us as we do to protect ourselves from them. More concretely, the requirements we have on callbacks and return pointers received from the adversary are also satisfied by callbacks and return pointers that we pass to them. For example, our return pointers are local capabilities because they must point to memory where we can store the old stack pointer, but the adversary's return pointers are also allowed to be local. Adversary callbacks are required to be global, but the callbacks we construct are allocated on the heap and also global.

Arguments and local capabilities Local capabilities are a central part of the calling convention as they are used to construct stack and return pointers. The use of local capabilities for the calling convention unfortunately limits the extent to which local capabilities can be used for other things. Say we are using the calling convention and receive a local capability other than the stack and return pointer, then we need to be careful if we want to use it because it may be an alias to the stack pointer. That is, if we first push something to the stack and then write to the local capability, then we may be (tricked into) overwriting our own local state. The logical relation helps by telling us what we need to ascertain or check in such scenarios to guarantee safety and preserve our invariants, but such checks may be costly and it is not clear to us whether there are practical scenarios where this might be realistic.

We also need to be careful when we receive a capability from an adversary that we want to pass on to a different (instance of the) adversary. It turns out that the logical relation again tells us when this is safe. Namely, the logical relation says that we can only pass on arguments that are safe in the world we invoke the adversary in. For instance, when we receive a stack pointer from an adversary, then we may at some point want to pass on part of this stack pointer to, say, a callback. In order to do so, we need to make sure the stack pointer is safe which means that, if we have revoked temporary invariants, the stack must not directly or indirectly allow access to local values that we cannot guarantee safety of. When received from an adversary, we have to consider the contents of the stack unsafe, so before we pass it on, we have to clear it, or perform a dynamic safety analysis of the stack contents and anything it points to. Clearing everything is not always desirable and a dynamic safety analysis is hard to get right and potentially expensive.

In summary, the use of local capabilities for other things than stack and return pointers is likely only possible in very specific scenarios when using our calling convention. While this is unfortunate, it is not unheard of that processors have built-in constructs that are exclusively used for handling control flow, such as, for example, the call and return instructions that exist in some instruction sets.

Single stack A single stack is a good choice for the simple capability machine presented here because it works well with higher-order functions. An alternative to a single stack would be to have a separate stack per component. The trouble with this approach is that, with multiple stacks and local stack pointers, it is not clear how components would retrieve their stack pointer upon invocation without compromising safety. A safe approach could be to have stack pointers stored by a central, trusted stack management component, but it is not clear how that could scale to large numbers of separate components. Handling large numbers of components is a requirement if we want to use capability machines to enforce encapsulation of, for example, every object in an object-oriented program or every closure in a functional program.

Capability machine formalization

Simplifications Our capability machine formalization assumes unbounded integers and an infinite address space. Further, it has a much simpler instruction set than that of a full ISA. By making these simplifications, we avoid tedious details but end up with a less realistic machine. However, the intent of this work is to gain ground in the formal work necessary to prove properties about low-level assembly programs on a capability machine not to apply it to a full fledged capability machine.

We do not believe that any of our simplifications are beyond reason and expanding the result seems plausible. If we added bounded integers, we would have to change the operational semantics to take over and under flows into account. This could be achieved by changing the `plus` and `minus` instruction to use modulo arithmetic. If we added a bounded address space, then we would have to take memory out of bounds into account. This would require changes to the memory operations (`store` and `load`) and the step relation of the operational semantics. The malloc specification would also need to be updated as it would have to signal failure when it runs out of memory. Finally, expanding the instruction set to a realistic ISA would add a handful of instructions that we would need to reason about in the proofs. All in all, it would add a lot of tedious work to expand this result to a realistic machine. The amount of details in the proofs is already at the threshold for what should be done with pen and paper, so expanding this work to a realistic machine would require mechanized proofs that can take care of the tedious details.

Reasoning about capability machine programs

Limitations The logical relation defined in Section 4 allows us to reason about capability machine programs. A limitation w.r.t. previous work is that the logical relation is tailored towards proofs of a specific class of properties.

Imagine, for example, that we invoke a block of adversary code in such a way that it only ever receives capabilities within a specific range of memory. After the code returns, we may try to prove that any capabilities passed back to us in the registers are still confined to that range of memory. The property talks about the specific implementation of a higher-order value rather than its behavior, like the invariants that are required/preserved when we use it.

Such properties are hard to prove in our model. For the example, it would be easy to conclude that the returned values are in the value relation (see Figure 8). This gives us a lot of behavioral information, like conditions under which the capabilities are safe to use and invariants that will be preserved when we do, but it does not tell us much about the range of authority of the capability. As a very concrete example, capabilities with permission o are always in the value relation irrespective of their range of authority. Behaviorally, this makes perfect sense; o-capabilities cannot be used for anything.

For our purposes, this restriction is unproblematic since we are only interested in proving behavioral properties (e.g., an assertion will never fail). In other situations, however, we may be interested in proving properties like the ones that are often considered in object capability literature: confinement, no authority amplification etc. Although such properties are more restrictive and tough to use for reasoning, Devriese et al. [2016] have demonstrated how a logical relation like ours can be adapted to also support them by quantifying the logical relation over a custom interpretation of effectful computations and the type of references. We expect their solution can be readily adapted to our setting modulo some details (like the fact that we do not just have read-write capabilities but also others).

Logical relation

Single orthogonal closure The definitions of $\mathcal{E}$ and $\mathcal{V}$ in Figure 8 apply a single orthogonal closure, a new variant of an existing pattern called biorthogonality. Biorthogonality is a pattern for defining logical relations [Krivine 1994; Pitts and Stark 1998] in terms of an observation relation of safe configurations (like we do). The idea is to define safe evaluation contexts as the set of contexts that produce safe observations when plugged with safe values and define safe terms as the set of terms that can be plugged into safe evaluation contexts to produce safe observations. This is an alternative to more direct definitions where safe terms are defined as terms that evaluate to safe values. An advantage of biorthogonality is that it scales better to languages with control effects like call/cc. Our definitions can be seen as a variant of biorthogonality; we take only a single orthogonal closure. We do not define safe evaluation contexts but immediately define safe terms as those that produce safe observations when plugged with safe values. This is natural because we model arbitrary assembly code that does not necessarily respect a particular calling convention. Return pointers are in principle values like all others and there is no reason to treat them specially in the logical relation.

Interestingly, Hur and Dreyer [2011] also use a step-indexed, Kripke logical relation for an assembly language (for reasoning about correct compilation from ML to assembly). However, because they only model non-adversarial code that treats return pointers according to a particular calling convention, they can use standard biorthogonality rather than a single orthogonal closure like us.

Public/private future worlds A novel aspect of our logical relation is how we model the temporary, revokable nature of local capabilities using public/private future worlds. The main insight is that this special nature generalizes that of the syntactically-enforced unstorable status of evaluation contexts in lambda calculi without control effects (of which well-bracketed control flow is a consequence). To reason about code that relies on this (particularly, the original awkward example), Dreyer et al. [2012] (DNB) formally capture the special status of evaluation contexts using Kripke worlds with public and private future world relations. Essentially, they allow relatedness of evaluation contexts to be monotone with respect to a weaker future world relation (public) than relatedness of values, formalizing the idea that it is safe to make temporary internal state modifications (private world transitions, which invalidate the continuation, but not other values) while an expression is performing internal steps, as long as the code returns to a stable state (i.e. transitions to a public future world of the original) before returning. We generalize this idea to reason about local capabilities: validity of local capabilities is allowed to be monotone with respect to a weaker future-world relation than other values, which we can exploit to distinguish between state changes that are always safe (public future worlds) and changes that are only valid if we clear all local capabilities (private future worlds). Our future world relations are similar to DNB's (for example, our proof of the awkward example uses exactly the same state transition system), but they turn up in an entirely different place in the logical relation: rather than using public future worlds for the special syntactic category of evaluation contexts, they are used in the value relation depending on the locality of the capability at hand. Additionally, our worlds are a bit more complex because, to allow local memory capabilities and write-local capabilities, they can contain (revokable) temporary regions that are only monotonous w.r.t. public future worlds, while DNB's worlds are entirely permanent.

Local capabilities in high-level languages We point out that local capabilities are quite similar to a feature proposed for the high-level language Scala: Osvald et al. [2016]'s second-class or local values. They are a kind of values that can be provided to other code for immediate use without allowing them to be stored in a closure or reference for later use. We believe reasoning about such values will require techniques similar to what we provide for local capabilities.

Why use a logical relation and not a simpler proof technique? The concept of a capability exists on different levels of abstraction on computers, so capabilities have been studied and safety properties proven about them in other contexts than assembly languages. Traditionally, logical relations have not been used for safety property proofs, so why do we need one here? To answer this question let's compare this work to Sewell et al. [2011] where they prove *integrity* and *authority confinement* for the seL4 Microkernel. The security properties they show are relative to a security policy that specifies the upperbound of capabilities a subject in the system can posses. The security policy is expressed as a collection of possible capabilities in the system, e.g. *subject A can have a write capability for memory B.* However, their policy are, in a sense, binary: subject A is either allowed to write to memory B or not. If writing is allowed, then any value can be written, as long as that value is itself legal. Sewell et al. [2011]'s results are not parametrised by a (possibly more restrictive) policy that subject A should observe on memory B. Our approach is much more fine-grained and allows us to define, for example, a policy that subject A can write to memory B, but only if the value is an even number. Because our invariants are indexed by worlds themselves, we can even define a policy that capabilities written to memory B must respect certain other invariants themselves. It is exactly the expressive invariants that enable us to prove the awkward example. We place a protocol on the memory where variable x is stored that says that it sometimes can be either 0 or 1 and at other times it has to be 1.

Capability safety is not just about the permission a capability carries, it is also about how the capability is used. Say for instance, a program has a write capability that will only be used to write even numbers. If this is the only write capability for that part of memory, then we should be able to rely on that part of memory only containing even numbers. In order to rely on this in our proofs, our notion of capability safety must be able to take the meaning of the program into account and express the memory invariant.

10 RELATED WORK

In this section, we summarize how our work relates to previous work. We do not repeat the work we discussed in Section 9.

Capability machines originate with Dennis and Van Horn [1966] and we refer to Levy [1984] and N. M. Watson et al. [2015] for an overview of previous work. The capability machine formalized in Section 2 is a simple but representative model modeled mainly after the M-Machine [Carter et al. 1994] (the enter pointers resemble the M-Machine's) and CHERI [N. M. Watson et al. 2015; Woodruff et al. 2014] (the memory and local capabilities resemble CHERI's). The latter is a recent and relatively mature capability machine. CHERI combines capabilities with a virtual memory approach in the interest of backwards compatibility and gradual adoption. As discussed, our local capabilities can cross module boundaries contrary to what is enforced by CHERI's default CCall implementation.

Plenty of other papers enforce well-bracketed control flow at a low level but most are restricted to preventing particular types of attacks and enforce only partial correctness of control flow. This includes particularly the line of work on control-flow integrity [Abadi et al. 2005]. This line of work uses a quite different attacker model than us. They assume an attacker that is unable to execute code but can overwrite arbitrary data at any time during execution (to model buffer overflows). By checking the address of every indirect jump and using memory access control to prevent overwriting code, this work enforces what they call control-flow integrity formalized as the property that every jump will follow a legal path in the control-flow graph. As discussed in Section 9, such a property ignores temporal properties and seems hard to use for reasoning.

More closely related to our work are papers that use a trusted stack manager and some form of memory isolation to enforce control-flow correctness as part of a secure compilation result [Juglaret et al. 2016; Patrignani et al. 2016]. Our work differs from theirs in that we use a different form of low-level security primitive (a capability machine with local capabilities rather than a machine with a primitive notion of compartments). Further, we do not use a trusted stack manager but a decentralized calling convention based on local capabilities. Also, both prove a secure compilation result from a high-level language which clearly implies a general form of control-flow correctness while we define a logical relation that can be used to reason about specific programs that rely on well-bracketed control flow.

Our logical relation is a unary, step-indexed Kripke logical relation with recursive worlds [Ahmed 2004; Appel and McAllester 2001; Birkedal et al. 2011; Pitts and Stark 1998], closely related to the one used by Devriese et al. [2016] to formulate capability safety in a high-level JavaScript-like lambda calculus. Our Fundamental Theorem is similar to theirs and expresses capability safety of the capability machine. Because we are not interested in externally observable side-effects (like console output or memory access traces), we do not require their notion of effect parametricity. Our logical relation uses several ideas from previous work like Kripke worlds with regions containing state transition systems [Ahmed et al. 2009], public/private future worlds [Dreyer et al. 2012] (see Section 9 for a discussion), and biorthogonality [Benton and Hur 2009; Hur and Dreyer 2011; Pitts and Stark 1998].

Swasey et al. [2017] have recently developed a logic, OCPL, for verification of object capability patterns. The logic is based on Iris [Jung et al. 2016, 2015; Krebbers et al. 2017a], a state of the art higher-order concurrent separation logic, and is formalized in Coq building on the Iris Proof Mode for Coq [Krebbers et al. 2017b]. OCPL gives a more abstract and modular way of proving capability safety for a lambda-calculus (with concurrency) compared to the earlier work by Devriese et al. [2016]. In the future, we would like to develop a new program logic for reasoning about capability safety for our capability machine model. In fact, we think the lemmas in Section 7 are suggestive of the style of results that could be captured in such a logic. We think Iris would also be a natural starting point for such an endeavour since Iris is a framework that can be instantiated with different programming languages. OCPL was able to leverage existing Iris specifications for a high-level programming language; for our capability machine model, however, it would be necessary to devise new kinds of specifications for our low-level programs with unstructured control-flow. It is likely that we could get inspiration from earlier work on logics for assembly programming languages, such as XCAP [Ni and Shao 2006]. Building a logic around the semantic model presented here would remove some of the tedious repetitive proof details making it more realistic to prove properties about larger more realistic programs.

If we want to scale this approach even further, we would like to reason at a higher level of abstraction namely at the level of a high-level language. That is, we would rather construct a program logic for a high-level programming language, so we can reason about the programs in the language we actually write them in and at the same time get guarantees about the compiled program. To achieve this, we would have to construct a secure compilation [Patrignani et al. 2019] that preserves the security abstraction of the high-level language.

El-Korashy also defined a formal model of a capability machine, namely CHERI, and uses it to prove a compartmentalization result [El-Korashy 2016] (not implying control-flow correctness). He also adapts control-flow integrity (see above) to the machine and shows soundness, seemingly without relying on capabilities.

ACKNOWLEDGMENTS

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU). Support for an STSM was received from COST Action EUTypes (CA15123). Dominique Devriese held a Postdoctoral fellowship from the Research Foundation Flanders (FWO) during most of this research. This research was supported in part by the Research Foundation Flanders (FWO).

REFERENCES

Martín Abadi. 1998. Protection in Programming-Language Translations: Mobile Object Systems. In European Conference on Object-Oriented Programming (Lecture Notes in Computer Science). Springer Berlin Heidelberg, 291–291. https://doi.org/10.1007/3-540-49255-0_70

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In Conference on Computer and Communications Security. ACM, 340–353. https://doi.org/10.1145/1102120.1102165

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In Principles of Programming Languages. ACM, 340–353.

Amal Jamil Ahmed. 2004. Semantics of types for mutable state. Ph.D. Dissertation. Princeton University.

Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. J. Comput. Syst. Sci. 39, 3 (1989), 343–375.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. ACM Trans. Program. Lang. Syst. 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In International Conference on Functional Programming. ACM, 97–108. https://doi.org/10.1145/1596550.1596567

Lars Birkedal and Aleš Bizjak. 2014. A Taste of Categorical Logic - Tutorial Notes. http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke Models over Recursive Worlds. In Principles of Programming Languages. ACM, 119–132. https://doi.org/10.1145/1926385.1926401

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. Theor. Comput. Sci. 411, 47 (2010), 4102–4122. https://doi.org/10.1016/j.tcs.2010.07.010

Aleš Bizjak. 2017. Some theorems about mutually recursive domain equations in the category of preordered COFEs. (Feb. 2017). Manuscript. Available at http://alesb.com/documents/notes/mutually-recursive-domain-eq.pdf.

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. In 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. https://doi.org/10.1145/3037697.3037725

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

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities using Logical Relations and Effect Parametricity. In European Symposium on Security and Privacy. IEEE.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. J. Funct. Program. 22, 4–5 (2012), 477–528.

Akram El-Korashy. 2016. A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI. Master's thesis. Saarland University. https://people.mpi-sws.org/~elkorashy/files/Thesis.pdf

Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In Hot Topics in Operating Systems. IEEE, 67–72. https://doi.org/10.1109/HOTOS.1997.595185

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In Principles of Programming Languages. ACM, 133–146. https://doi.org/10.1145/1926385.1926402

Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In International Conference on Computer Design. IEEE, 641–648. https://doi.org/10.1109/ICCD.2017.112

Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In Computer Security Foundations. IEEE, 45–60. https://doi.org/10.1109/CSF.2016.11

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In International Conference on Functional Programming. ACM, 256–269.

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 Principles of Programming Languages. ACM, 637–650.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In European Symposium on Programming. Springer, Berlin, Heidelberg.

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

Jean-Louis Krivine. 1994. Classical Logic, Storage Operators and Second-Order Lambda-Calculus. Annals of Pure and Applied Logic 68, 1 (June 1994), 53–78. https://doi.org/10.1016/0168-0072(94)90047-7

Henry M. Levy. 1984. Capability-based computer systems. Vol. 12. Digital Press Bedford.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. The Java Virtual Machine Specification. Pearson Education.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In S&P. IEEE, 125–140. https://doi.org/10.1109/SP.2010.16

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. ACM Trans. Program. Lang. Syst. 21, 3 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In

Security and Privacy. IEEE, 20–37. https://doi.org/10.1109/SP.2015.9

Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In Principles of Programming Languages. ACM.

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification Gone Too Far? Affordable 2Nd-Class Values for Fun and (Co-)Effect. In Object-Oriented Programming, Systems, Languages, and Applications. ACM, 234–251. https://doi.org/10.1145/2983990.2984009

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. ACM Comput. Surv. 51, 6, Article 125 (Feb. 2019), 36 pages. https://doi.org/10.1145/3280984

Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully-Abstract Compilation. In Computer Security Foundations. IEEE, 17–30. https://doi.org/10.1109/CSF.2016.9

Andrew M. Pitts and Ian D. B. Stark. 1998. Operational Reasoning for Functions with Local State. In Higher Order Operational Techniques in Semantics, Andrew D. Gordon and Andrew M. Pitts (Eds.). Cambridge University Press, 227–274.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 Enforces Integrity. In Interactive Theorem Proving, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, 325–340.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In Symposium on Operating Systems Principles. ACM, 170–185. https://doi.org/10.1145/319151.319163

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities. In European Symposium on Programming. Springer, 475–501.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning About a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management - Technical Appendix Including Proofs and Details. Technical Report. Dept. of Computer Science, Aarhus University. https://arxiv.org/abs/1902.05283

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), 19:1–19:28 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://doi.org/10.1145/3133913

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In Security and Privacy. IEEE Computer Society, 48–62. https://doi.org/10.1109/SP.2013.13

Jacob Thamsborg and Lars Birkedal. 2011. A Kripke Logical Relation for Effect-based Program Transformations. In International Conference on Functional Programming. ACM, 445–456. https://doi.org/10.1145/2034773.2034831

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In Symposium on Operating Systems Principles. ACM, 203–216. https://doi.org/10.1145/168619.168635

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In International Symposium on Computer Architecture. IEEE, 457–468.

A APPENDIX

In this appendix, we give more precise formulations of lemmas that were mentioned in the paper, and the most important supporting definitions and lemmas. The goal is to provide details that can help to understand in more detail what we discuss in the paper. Full details and proofs are not given here, but for those we refer to the technical appendix [Skorstengaard et al. 2019a].

A.1 Logical relation

n-subset simulation

$$\frac{(s, \phi_{pub}, \phi) = (s', \phi'_{pub}, \phi') \qquad \forall \hat{W}.\, H\ (s)\ (\hat{W}) \overset{n}{\subseteq} H'\ (s')\ (\hat{W})}{(v, s, \phi_{pub}, \phi, H) \overset{n}{\underset{\sim}{\subset}} (v', s', \phi'_{pub}, \phi', H')}$$

where $\overset{n}{\subseteq}$ is defined as follows: define erasure for step-indexed sets as

$$\lfloor A \rfloor_n = \{(m, a) \in A \mid m < n\}$$

and define $\overset{n}{\subseteq}$ as

$$A \nsubseteq B \texttt{ iff } \lfloor A \rfloor_n \subseteq \lfloor B \rfloor_n$$

Transition system relations

$$\text{Rels} = \{(\phi_{pub}, \phi) \in \mathcal{P}(\text{RState}^2) \times \mathcal{P}(\text{RState}^2) \mid \phi_{pub}, \phi \text{ is reflexive and transitive and } \phi_{pub} \subseteq \phi\}$$

Erasure

$$\lfloor W \rfloor_S \overset{def}{=} \lambda r. \begin{cases} W(r) & W(r).v \in S \\ \bot & \text{otherwise} \end{cases}$$

Active region projection

$$\mathit{active} : \text{World} \longrightarrow 2^{\text{RegionName}}$$

$$\mathit{active}(W) \overset{def}{=} \text{dom}(\lfloor W \rfloor_{\{\text{perm},\text{temp}\}})$$

Revoke temporary regions in a world

$$\mathit{revokeTemp} : \text{World} \longrightarrow \text{World}$$

$$\mathit{revokeTemp}(W) \overset{def}{=} \lambda r. \begin{cases} \text{revoked} & \text{if } W(r) = (\text{temp}, s, \phi_{pub}, \phi, H) \\ W(r) & \text{otherwise} \end{cases}$$

Projection of regions based on locality

$$\mathit{localityReg}(g, W) \overset{def}{=} \begin{cases} \text{dom}(\lfloor W \rfloor_{\{\text{perm},\text{temp}\}}) & \text{if } g = \text{local} \\ \text{dom}(\lfloor W \rfloor_{\{\text{perm}\}}) & \text{if } g = \text{global} \end{cases}$$

Address stratification

$$\begin{aligned} &\iota = (v, s, \phi_{pub}, \phi, H) \text{ is address-stratified iff} \\ &\quad \forall s', \hat{W}, n, ms, ms'. \\ &\qquad (n, ms), (n, ms') \in H\ s'\ \hat{W} \Rightarrow \\ &\qquad \text{dom}(ms) = \text{dom}(ms') \wedge \\ &\qquad \forall a \in \text{dom}(ms). (n, ms[a \mapsto ms'(a)]) \in H\ s'\ \hat{W} \end{aligned}$$

A.2 Complete ordered family of equivalences (c.o.f.e)

This is an excerpt from Birkedal and Bizjak [2014] about c.o.f.e.'s.

Definition A.1 (o.f.e.). An ordered family of equivalence (o.f.e) is a set and a family of equivalences $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ that satisfy the following properties:

- $\overset{0}{=}$ is the total relation on X
- $\forall n. \forall x, y \in X. x \overset{n+1}{=} y \Rightarrow x \overset{n}{=} y$
- $\forall x, y \in X. (\forall n. x \overset{n}{=} y) \Rightarrow x = y$

We say that an o.f.e. $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ is inhabited if there exists an element $x \in X$.

If you are familiar with metric spaces observe that o.f.e.'s are but a different presentation of bisected 1-bounded ultrametric spaces.

Definition A.2 (Cauchy sequences and limits). Let $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ be an o.f.e. and $\{x_n\}_{n=0}^{\infty}$ be a sequence of elements of X. Then $\{x_n\}_{n=0}^{\infty}$ is a Cauchy sequence if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x_j \overset{k}{=} x_n$$

or in words, the elements of the chain get arbitrarily close.

An element $x \in X$ is the limit of the sequence $\{x_n\}_{n=0}^{\infty}$ if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x \overset{k}{=} x_n.$$

A sequence may or may not have a limit. If it has we say that the sequence converges. The limit is necessarily unique in this case and we write $\lim_{n\to\infty} x_n$ for it.

Definition A.3 (c.o.f.e.). A complete ordered family of equivalences (c.o.f.e) is an o.f.e $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where all Cauchy sequences have a limit.

Definition A.4. Let $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ be two ordered families of equivalences and f a function from the set X to the set Y. The function f is

- non-expansive if for any $x, x' \in X$, and any $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n}{=}_Y f(x')$$

- contractive if for any $x, x' \in X$, and any $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n+1}{=}_Y f(x')$$

Theorem A.5 (Banach's fixed point theorem). *Let* $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ *be a an inhabited c.o.f.e. and* $f : X \to X$ *a contractive function. Then* f *has a unique fixed point.*

Definition A.6 (The category $\mathcal{U}$). The category $\mathcal{U}$ of complete ordered families of equivalences has as objects complete ordered families of equivalences and as morphisms non-expansive functions.

Definition A.7. The functor ► is a functor on $\mathcal{U}$ defined as

$$\blacktriangleright \left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right) = \left(X, \left(\overset{n}{\equiv}\right)_{n=0}^{\infty}\right)$$
$$\blacktriangleright (f) = f$$

where $\overset{0}{\equiv}$ is the total relation and $x \overset{n+1}{\equiv} x'$ iff $x \overset{n}{=} x'$

From now on, we often use the underlying set X to denote a (complete) o.f.e. $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$, leaving the family of equivalence relations implicit.

Definition A.8. A functor $F : \mathcal{U}^{\text{op}} \times \mathcal{U} \to \mathcal{U}$ is locally non-expansive if for all objects X, X', Y, and Y' in $\mathcal{U}$ and $f, f' \in \mathcal{U}(X, X')$ and $g, g' \in \mathcal{U}(Y', Y)$ we have

$$f \overset{n}{=} f' \wedge g \overset{n}{=} g' \implies F(f,g) \overset{n}{=} F(f',g').$$

It is locally contractive if the stronger implication

$$f \overset{n}{=} f' \wedge g \overset{n}{=} g' \implies F(f,g) \overset{n+1}{=} F(f',g').$$

holds. Note that the equalities are equalities on function spaces.

Proposition A.9. *If F is a locally non-expansive functor then $\blacktriangleright \circ F$ and $F \circ (\blacktriangleright^{op} \times \blacktriangleright)$ are locally contractive. Here, the functor $F \circ (\blacktriangleright^{op} \times \blacktriangleright)$ works as*

$$(F \circ (\blacktriangleright^{op} \times \blacktriangleright))(X, Y) = F\left(\blacktriangleright^{op}(X), \blacktriangleright(Y)\right)$$

on objects and analogously on morphisms and $\blacktriangleright^{op} \colon \mathcal{U}^{op} \longrightarrow \mathcal{U}^{op}$ is just $\blacktriangleright$ working on $\mathcal{U}^{op}$ (i.e., its definition is the same).

Definition A.10. A fixed point of a locally contractive functor F is an object $X \in \mathcal{U}$, such that $F(X, X) \cong X$.

The following is America and Rutten's fixed point theorem [America and Rutten 1989].

Theorem A.11. *Every locally contractive functor F such that $F(1, 1)$ is inhabited has a unique fixed point. The fixed point is unique among inhabited c.o.f.e.'s. If in addition $F(\emptyset, \emptyset)$ is inhabited then the fixed point of F is unique.*

In Birkedal et al. [2010] one can find a category-theoretic generalization, which shows how to obtain fixed points of locally contractive funtors on categories enriched in $\mathcal{U}$, in particular on the category of preordered c.o.f.e.'s. A preodered c.o.f.e. is a c.o.f.e. equipped with a preorder that is closed under taking limits of converging sequences. The formulation in loc. cit. also applies to solve mutually recursive domain equations on preordered c.o.f.e.'s; see Bizjak [2017] for an explicit statement. That is the solution theorem we use to prove Theorem 4.1.

A.3 Load instruction sufficiency lemma

Lemma A.12 (Conditions for store instruction are sufficient). *If*

- $ms = ms' \uplus ms_f$
- $ms' :_n W$
- $((perm, g), b, e, a) = c$
- $(n, c) \in \mathcal{V}(W)$
- $writeAllowed(perm)$
- $withinBounds(c)$
- $(n, w) \in \mathcal{V}(W)$
- *if* $w = ((\_, \mathrm{local}), \_, \_, \_)$, *then* $perm \in \{\textsc{rwlx}, \textsc{rwl}\}$

then $a \in \mathrm{dom}(ms')$ *(i.e.* $ms[a \mapsto w] = ms'[a \mapsto w] \uplus ms_f$*) and* $ms'[a \mapsto w] :_n W$

A.4 Macros

Implementation of the macros used in `scall`. Implementations of the macros not presented here can be found in the technical appendix [Skorstengaard et al. 2019a].

push r

```
lea r_stk 1
store r_stk r
```

pop r

```
load r r_stk
minus r_t1 0 1
lea r_stk r_t1
```

rclear $r_1, \ldots, r_n$

```
move $r_1$ 0
move $r_2$ 0
...
```

```
  move r_n 0
```

`mclear` r

```
  move r_t r
  getb r_t1 r_t
  geta r_t2 r_t
  minus r_t2 r_t1 r_t2
  lea r_t r_t2
  gete r_t2
  minus r_t1 r_t2 r_t1
  plus r_t1 r_t1 1
  move r_t2 pc
  lea r_t2 off_end
  move r_t3 pc
  lea r_t3 off_iter
iter:
  jnz r_t2 r_t1
  store r_t 0
  lea r_t 1
  plus r_t1 r_t1 1
  jmp r_t3
end:
  move r_t 0
  move r_t1 0
  move r_t2 0
  move r_t3 0
```

Where off_{end} and off_{iter} are the offsets to the label end and `iter`, respectively.

`call` $r(\bar{r}_{args}, \bar{r}_{priv})$

The `call` macro constitutes a calling convention based on heap allocated activation records. This alternative to `scall` is included to illustrate that the logical relation can be used to reason about other calling conventions. In the following, $\bar{r}_{args}$ and $\bar{r}_{priv}$ are lists of registers. An overview of this call:

- Set up activation record
- Create local enter capability for activation (protected return pointer)
- Clear unused registers
- Jump
- Upon return: Run activation code
  - Restore private registers
  - Jump to return capability

```
  malloc r_t size
// store private state in activation record
  store r_t r_priv,1
  lea r_t 1
  store r_t r_priv,2
  lea r_t 1
  ...
  lea r_t 1
  store r_t r_priv,n
  lea r_t 1
```

```
// store old pc
  move r_t1 pc
  lea r_t1 off_end
  store r_t r_t1
  lea r_t1 1
// store activation record
  store r_t encode(i_1)
  lea r_t1 1
  ...
  lea r_t1 1
  store r_t encode(i_m)
  lea r_t1 k
  restrict r_t1 encodePermPair((local,e))
  move r_0 r_t1
// Clear unused registers
  rclear R // R = RegisterName - {r,pc,r_0,r_args}
  jmp r
end:
```

Where $\overline{r_{priv}} = r_{priv,1}, \ldots, r_{priv,n}$, *size* is the size of the activation record, off_{end} is the offset to the end label, and k is $m - 1$, i.e. the offset to the first instruction of the activation code.
The activation record. The instructions correspond to $i_1, \ldots, i_m$ in the above.

```
  move r_t pc
  getb r_t1 r_t
  geta r_t2 r_t
  minus r_t1 r_t1 r_t2
// load private state
  lea r_t r_t1
  load r_priv,1 r_t
  lea r_t 1
  load r_priv,2 r_t
  lea r_t 1
  ...
  lea r_t 1
  load r_priv,n r_t
  lea r_t 1
// load old pc
  load pc r_t
```

A.5 Reasoning about programs definitions

Definition A.13. We say that (reg, ms) is looking at $[i_0, \cdots, i_n]$ followed by c_{next} iff

- $reg(\mathrm{pc}) = ((p, g), b, e, a)$
- $p = \textsc{rwx}$, $p = \textsc{rx}$, or $p = \textsc{rwlx}$
- $a + n \leq e$, $b \leq a \leq e$
- $ms(a + 0, \cdots, a + n) = [i_0, \cdots, i_n]$
- $c_{next} = ((p, g), b, e, a + n + 1)$

Definition A.14. We say that "(reg, ms) links key as j to c" iff

- $reg(pc) = ((perm, g), b, e, a)$
- $ms(b) = ((\_, \_), b_{link}, \_, \_)$
- $ms(b_{link} + j) = c$

Definition A.15. We say that *reg* points to stack with ms_{stk} used and ms_{unused} unused iff

- $reg(r_{stk}) = ((\textsc{rwlx}, \text{local}), b_{stk}, e_{stk}, a_{stk})$
- $\text{dom}(ms_{unused}) = [a_{stk} + 1, \cdots, e_{stk}]$
- $\text{dom}(ms_{stk}) = [b_{stk}, \cdots, a_{stk}]$
- $b_{stk} - 1 \leq a_{stk}$

A.6 Example correctness lemmas

Lemma A.16 (Correctness lemma for f1, copy of Lemma 8.1).
For all $n \in \mathbb{N}$ let

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\textit{E}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f1} &\overset{def}{=} ((\textit{RWX}, \text{global}), \texttt{f1} - \mathit{offsetLinkFlag}, \texttt{1f}, \texttt{f1}) \\
c_{malloc} &\overset{def}{=} ((\textit{E}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
m &\overset{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{frame}
\end{aligned}$$

and

- c_{malloc} *satisfies the specification for malloc and* $\iota_{malloc,0}$ *is the region from the specification.*

where

$$\begin{aligned}
&\text{dom}(ms_{f1}) = [\texttt{f1} - \mathit{offsetLinkFlag}, \texttt{1f}] \\
&\text{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{link}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}$$

and

- $ms_{f1}(\texttt{f1} - \mathit{offsetLinkFlag}) = ((\textit{RO}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link})$*,* $ms_{f1}(\texttt{f1} - \mathit{offsetLinkFlag} + 1) = ((\textit{RW}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$*, the rest of* ms_{f1} *contains the code of* $f1$*.*
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- ms_{adv} *contains a global read-only capability for* ms_{link} *on its first address. The remaining cells of the memory segment only contain instructions.*

if

$$(reg[\text{pc} \mapsto c_{f1}], m) \to_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

Lemma A.17 (Correctness lemma for $\mathtt{f2}$, detailed version of Lemma 8.2). *let*

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\textsc{e}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f2} &\overset{def}{=} ((\textsc{rwx}, \text{global}), \mathtt{f2} - \mathit{offsetLinkFlag}, \mathtt{2f}, \mathtt{f2}) \\
c_{malloc} &\overset{def}{=} ((\textsc{e}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
c_{stk} &\overset{def}{=} ((\textsc{rwlx}, \text{local}), b_{stk}, e_{stk}, b_{stk} - 1) \\
c_{link} &\overset{def}{=} ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link}) \\
\mathit{reg} &\in \text{Reg} \\
m &\overset{def}{=} ms_{f2} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}$$

and

- c_{malloc} *satisfies the specification for malloc and* $\iota_{malloc,0}$ *is the region from the specification.*

where

$$\begin{aligned}
&\text{dom}(ms_{f2}) = [\mathtt{f2} - \mathit{offsetLinkFlag}, \mathtt{2f}] \\
&\text{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{link}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{stk}) = [b_{stk}, e_{stk}] \\
&\text{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}] \qquad \text{for all } n \in \mathbb{N}
\end{aligned}$$

and

- $ms_{f2}(\mathtt{f2} - \mathit{offsetLinkFlag}) = ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link})$, $ms_{f2}(\mathtt{f2} - \mathit{offsetLinkFlag} + 1) = ((\textsc{rw}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{f2} *contains the code of* $f2$.
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and* $\forall a \in [b_{adv} + 1, e]. \ ms_{adv}(a) \in \mathbb{Z}$

if

$$(\mathit{reg}[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

Lemma A.18 (Correctness lemma for $\mathtt{f3}$, detailed version of Lemma 8.3). *For all* $n \in \mathbb{N}$ *let*

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\textsc{e}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f3} &\overset{def}{=} ((\textsc{rwx}, \text{global}), \mathtt{f3} - \mathit{offsetLinkFlag}, \mathtt{3f}, \mathtt{f3}) \\
c_{stk} &\overset{def}{=} ((\textsc{rwlx}, \text{local}), b_{stk}, e_{stk}, b_{stk} - 1) \\
c_{malloc} &\overset{def}{=} ((\textsc{e}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
c_{link} &\overset{def}{=} ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link}) \\
\mathit{reg} &\in \text{Reg} \\
m &\overset{def}{=} ms_{f3} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}$$

and

- c_{malloc} *satisfies the specification for malloc.*

where

$$\begin{aligned}\mathrm{dom}(ms_{f3}) &= [\mathtt{f3} - \mathit{offsetLinkFlag}, \mathtt{3f}]\\ \mathrm{dom}(ms_{flag}) &= [\mathit{flag}, \mathit{flag}]\\ \mathrm{dom}(ms_{link}) &= [\mathit{link}, \mathit{link} + 1]\\ \mathrm{dom}(ms_{stk}) &= [b_{stk}, e_{stk}]\\ \mathrm{dom}(ms_{adv}) &= [b_{adv}, e_{adv}]\\ ms_{malloc} &:_n [0 \mapsto \iota_{malloc,0}]\end{aligned}$$

and

- $ms_{f3}(\mathtt{f3} - \mathit{offsetLinkFlag}) = ((\textsc{ro}, \mathrm{global}), \mathit{link}, \mathit{link} + 1, \mathit{link})$, $ms_{f3}(\mathtt{f3} - \mathit{offsetLinkFlag} + 1) = ((\textsc{rw}, \mathrm{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$*, the rest of* ms_{f3} *contains the code of* $f3$.
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and all other addresses of* ms_{adv} *contain instructions.*

if

$$(reg[\mathrm{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \to_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

Lemma A.19 (Correctness of $g1$, detailed version of Lemma 8.4). *For all* $n \in \mathbb{N}$ *let*

$$\begin{aligned}c_{adv} &\overset{def}{=} ((\textsc{rwx}, \mathrm{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag})\\ c_{g1} &\overset{def}{=} ((\textsc{e}, \mathrm{global}), \mathtt{g1} - \mathit{offsetLinkFlag}, \mathtt{4f}, \mathtt{g1})\\ c_{stk} &\overset{def}{=} ((\textsc{rwlx}, \mathrm{local}), b_{stk}, e_{stk}, b_{stk} - 1)\\ c_{malloc} &\overset{def}{=} ((\textsc{e}, \mathrm{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag})\\ c_{link} &\overset{def}{=} ((\textsc{ro}, \mathrm{global}), \mathit{link}, \mathit{link}, \mathit{link})\\ m &\overset{def}{=} ms_{g1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}\end{aligned}$$

where

- c_{malloc} *satisfies the specification for malloc with* $\iota_{malloc,0}$

$$\begin{aligned}\mathrm{dom}(ms_{g1}) &= [\mathtt{g1} - \mathit{offsetLinkFlag}, \mathtt{4f}]\\ \mathrm{dom}(ms_{flag}) &= [\mathit{flag}, \mathit{flag}]\\ \mathrm{dom}(ms_{link}) &= [\mathit{link}, \mathit{link}]\\ \mathrm{dom}(ms_{stk}) &= [b_{stk}, e_{stk}]\\ \mathrm{dom}(ms_{adv}) &= [b_{adv}, e_{adv}]\\ ms_{malloc} &:_n [0 \mapsto \iota_{malloc,0}]\end{aligned}$$

and

- $ms_{g1}(\mathtt{g1} - \mathit{offsetLinkFlag}) = ((\textsc{ro}, \mathrm{global}), \mathit{link}, \mathit{link}, \mathit{link})$, $ms_{g1}(\mathtt{g1} - \mathit{offsetLinkFlag} + 1) = ((\textsc{rw}, \mathrm{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$*, the rest of* ms_{g1} *contains the code of* $g1$ *immediately followed by the code of* $f4$.
- $ms_{flag} = [\mathit{flag} \mapsto 0]$

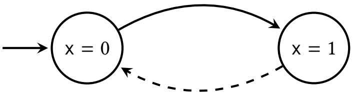


Fig. 17. Illustration of transition system in ι_x. The dashed line is the private transition.

- $ms_{link} = [link \mapsto c_{malloc}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and all other addresses of* ms_{adv} *contain instructions.*
- $\forall a \in \mathrm{dom}(ms_{stk}).\ ms_{stk}(a) = 0$

if

$$(reg_0[\mathrm{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \rightarrow_n (halted, m'),$$

then

$$m'(flag) = 0$$

A.7 Awkward example

The region for variable x The region ι_x, is the region omitted from the proof sketch for the awkward example. Figure 17 illustrates the transition system of ι_x.

Definition A.20.

$$\begin{aligned}
\iota_x &= (perm, 0, \phi_{pub}, \phi, H_x)\\
\phi_{pub} &= \{(0,1)\}^*\\
\phi &= (1,0) \cup \phi_{pub}\\
H_x\ s\ \hat{W} &= \{(n, ms) \mid ms(x) = s \wedge n > 0\} \cup \{(0, ms)\}
\end{aligned}$$

Static region This static region only requires that the memory segment is the given one. As it does not require safety, capabilities for this region cannot be gives to adversarial code.

$$\begin{aligned}
\iota^{sta}(v, ms) &= (v, 1, =, =, H^{sta}\ \ ms)\\
H^{sta}\ \ ms\ s\ \hat{W} &= \{(n, ms) \mid n > 0\} \cup \{(0, ms') \mid ms' \in \mathrm{Mem}\}
\end{aligned}$$

Static safe region Static region that also requires safety. It is safe to give adversarial code read capabilities for this region.

$$\iota^{sta,u}(v, ms) = (v, 1, =, =, H^{sta,u}\ \ ms)$$

$$H^{sta,u}\ \ ms\ s\ \hat{W} = \left\{ (n, ms') \;\middle|\; \begin{array}{l} ms' = ms \wedge \\ \forall a \in \mathrm{dom}(ms). \\ \quad ms(a) \text{ is non-local} \wedge \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms') \mid ms' \in \mathrm{Mem}\}$$