Reasoning About a Machine with Local Capabilities

Provably Safe Stack and Return Pointer Management

Lau Skorstengaard[1], Dominique Devriese[2], and Lars Birkedal[1]

[1] Aarhus University
{lau,birkedal}@cs.au.dk
[2] imec-DistriNet, KU Leuven
dominique.devriese@cs.kuleuven.be

Abstract. 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.

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 only certain common forms of attacks (leading to an attack-defense arms race) and/or rely on techniques like machine code rewriting [**?**, **?**], machine code verification [**?**], virtual machines with a native stack [**?**] or randomization [**?**]. The latter techniques essentially emulate protection techniques on existing hardware, at the cost of performance, system complexity and/or security.

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 [**?**, **?**, **?**], but 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 [**?**, **?**]. 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 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 [**?**]: the standard contiguous C stack is split into a central, trusted stack, managed by trusted call and return instructions, and disjoint, private, per-compartment stacks. 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 **??**).
- We define a novel calling convention enforcing control-flow correctness and encapsulation of stack frames (Section **??**). 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: 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 [**?**]. 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 [**?**, **?**]), 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. [**?**]'s public and private future worlds [**?**] to express the special nature of local capabilities. The logical relation and the fundamental theorem expressing capability safety are presented in Section **??**.
- 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 **??**). The examples demonstrate both the power of our formulation of capability safety and our calling convention.

For reasons of space, some details and all proofs have been omitted; please refer to the technical appendix [**?**] for those.

2 A Capability Machine with Local Capabilities

In this paper, we work with a formal 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 [**?**] and CHERI [**?**]. To avoid uninteresting details, we assume an infinite address space and unbounded integers.

We define the syntax of our capability machine in Figure **??**. 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. There is no notion of pointers other than capabilities, so we will use the terms interchangeably. The available permissions and their ordering are depicted in Figure **??**: the permissions include 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.

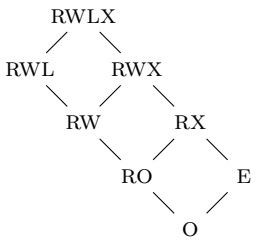

Figure 3: Permission hierarchy

$$
\begin{array}{rrcl}
a \in & \text{Addr} & \stackrel{\text{def}}{=} & \mathbb{N} \\
w \in & \text{Word} & \stackrel{\text{def}}{=} & \mathbb{Z} + \text{Cap} \\
perm \in & \text{Perm} & ::= & \text{O} \mid \text{RO} \mid \text{RW} \mid \text{RWL} \mid \\
 & & & \text{RX} \mid \text{E} \mid \text{RWX} \mid \text{RWLX} \\
g \in & \text{Global} & ::= & \text{global} \mid \text{local} \\
 & \text{Conf} & ::= & \text{ExecConf} + \{\mathit{failed}\} + \{\mathit{halted}\} \times \text{Mem} \\
 & \text{Cap} & ::= & \{((\mathit{perm}, g), b, e, a) \mid b, a \in \text{Addr}, e \in \text{Addr} \cup \{\infty\}\}
\end{array}
$$

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

Figure 1: The syntax of our capability machine assembly language.

$$
\Phi \to \begin{cases} [\![\mathit{decode}(n)]\!]\,(\Phi) & \text{if } \Phi.\text{reg}(\text{pc}) = ((\mathit{perm}, g), b, e, a) \text{ and } b \le a \le e \\ & \quad \text{and } \mathit{perm} \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \text{ and } \Phi.\text{mem}(a) = n \\ \mathit{failed} & \text{otherwise} \end{cases}
$$

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

i	$[\![i]\!]\,(\Phi)$	Conditions
`fail`	*failed*	
`halt`	$(\mathit{halted}, \Phi.\text{mem})$	
`move` r_1 r_2	$\mathit{updPc}(\Phi[\text{reg}.r_1 \mapsto w])$	$r_2 \in \text{Reg} \Rightarrow w = \Phi.\text{reg}(r_2)$ and $r_2 \in \mathbb{Z} \Rightarrow w = r_2$
`load` r_1 r_2	$\mathit{updPc}(\Phi[\text{reg}.r_1 \mapsto w])$	$\Phi.\text{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $w = \Phi.\text{mem}(a)$ and $b \le a \le e$ and $\mathit{perm} \in \{\text{RWX}, \text{RWLX}, \text{RX}, \text{RW}, \text{RWL}, \text{RO}\}$
`restrict` r_1 r_2	$\mathit{updPc}(\Phi[\text{reg}.r_1 \mapsto w])$	$\Phi.\text{reg}(r_2) = ((\mathit{perm}, g), b, e, a)$ and $(\mathit{perm}', g') = \mathit{decodePermPair}(\Phi.\text{reg}(r_2))$ and $(\mathit{perm}', g') \sqsubseteq (\mathit{perm}, g)$ and $w = ((\mathit{perm}', g'), b, e, a)$
`geta` r_1 r_2	$\mathit{updPc}(\Phi[\text{reg}.r_1 \mapsto a])$	$\Phi.\text{reg}(r_2) = ((\_, \_), \_, \_, a)$
`jmp` r	$\Phi[\text{reg.pc} \mapsto \mathit{newPc}]$	if $\Phi.\text{reg}(r) = ((\text{E}, g), b, e, a)$, then $\mathit{newPc} = ((\text{RX}, g), b, e, a)$ otherwise $\mathit{newPc} = \Phi.\text{reg}(r)$
`store` r_1 r_2	$\mathit{updPc}(\Phi[\text{mem}.a \mapsto w])$	$\Phi.\text{reg}(r_1) = ((\mathit{perm}, g), b, e, a)$ and $\mathit{perm} \in \{\text{RWX}, \text{RWLX}, \text{RW}, \text{RWL}\}$ and $b \le a \le e$ and $w = \Phi.\text{reg}(r_2)$ and if $w = ((\_, \text{local}), \_, \_, \_)$, then $\mathit{perm} \in \{\text{RWLX}, \text{RWL}\}$
	...	
_	*failed*	otherwise

Figure 2: An excerpt from the operational semantics.

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 keeps hold of 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 addition 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). 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 **??** shows an excerpt of the operational semantics for a few representative instructions. Essentially, a configuration Φ either decodes and executes the instruction at Φ.reg(pc) if it is executable and its address is in the valid range or otherwise fails. The table in the figure shows for instructions i the result of executing them in configuration Φ. `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). `restrict` updates a capability's permissions and global/local tag in the register file, but only if the new permissions are weaker than the original. It also never turns local capabilities into global ones. `geta` queries the current address of a capability and stores it in a register.

The `jmp` instruction updates the program counter to a requested location, but it is complicated by the presence of *enter capabilities*, modeled after the M-Machine's [**?**]. 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, but 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 $c = ((\text{E}, g), b, e, a)$ 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 but when c is jumped to, the instructions can start executing and have access to the local data through the updated version of c that is then in pc.

Finally, the `store` instruction updates the memory to the requested value if the capability has write authority for the requested location. However, the

instruction is complicated by the presence of *local capabilities*, modeled after the ones in the CHERI processor [**?**]. Basically, 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 take care to clear the capability from the register file after control is passed back to us, they will not have been able 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.

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, but we refer to the technical appendix [**?**] for full details.

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 this convention's high-level approach, the security measures to be taken in a number of situations (motivating each 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 next 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 a special 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.

Returning to the adversary Perhaps the simplest situation is returning to the adversary after they invoked our code. 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.

However, if we are returning from a function that has itself invoked adversary code, 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 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. 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. Since we may be talking about a large part of memory, this requirement is the most problematic aspect of our calling convention for performance, but see Section **??** 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 ca-

pabilities 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 through write-local capabilities, which means (because of our assumptions above): on the stack. Since the stack pointer itself is also local, it can also only be stored on the stack. Because we clear the part of the stack that the adversary has had access to before we pass control back, there is no way for them to recover either of these local capabilities.

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 the adversary stores their stack pointer, invokes trusted code that uses part of the stack to store private data and then invokes the adversary again with a stack pointer restricted to exclude the part containing the private data. 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 So 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 of these capabilities 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 **??**. 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

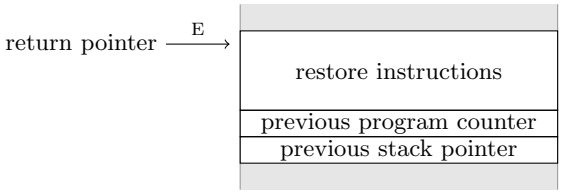

Figure 4: Structure of an activation record

of restore instructions in the record, so that upon invocation, these instructions are executed and can load the old stack pointer and program counter back into the register file. As the return pointer is an enter pointer, the adversary cannot get hold of the activation record's contents, but 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 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.[3]

[3] 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.

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, 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.

Reusable macro instructions We define a number of reusable macros capturing the calling convention and other conveniences. All macros that use the stack assume a stack pointer in register r_{stk}. The macro `fetch` r *name* fetches the capability related to *name* from the linking table and stores it in register r. The macros `push` r and `pop` r add and remove elements from the stack. The macro `prepstk` r is used when a callback is invoked by the adversary and prepares the received stack pointer by checking that it has permission RWLX. The macro `scall` r`(`$\overline{r_{args}}$`,`$\overline{r_{priv}}$`)` jumps to the capability in register r in the manner described above. That is, it pushes local state (the contents of registers $\overline{r_{priv}}$) and the activation record (return code, return pointer, stack pointer) to the stack, creates an E return pointer, restricts the stack pointer, clears the unused part of the stack, clears the necessary registers and jumps to r. Upon return, the private state is

restored. The macro `mclear` r clears all the memory the capability in register r has authority over. The macro `rclear` *regSet* clears all the registers in *regSet*. The macro `reqglob` r checks whether the word in register r is a global capability. 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) where the contents of $\overline{r_i}$ is stored. In the code referred to by r, an implicit fetch happens when an instruction refers to x_i.

The technical appendix [**?**] contains detailed descriptions of all the macros.

4 Logical Relation

In this section, we formalize the guarantees provided by the capability machine, including the specific guarantees for local capabilities, by means of a step-indexed Kripke logical relation with recursively defined worlds. We use the logical relation in the following section to show local-state encapsulation and control-flow integrity properties for challenging example programs.

4.1 Worlds

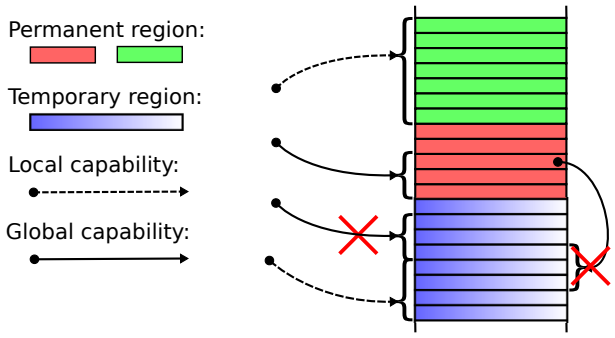

Figure 5: The relation between local/global capabilities and temporary/permanent regions. The colored fields are regions governing parts of memory. Global capabilities cannot depend on temporary regions.

A world is a finite map from region names, modeled as natural numbers, to regions that each correspond to an invariant of 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, to describe how the invariants are allowed to change over time. In other words, they are protocols for the region's memory. These are similar to what has been used in logical relations for high-level languages [**?**,**?**,**?**]. 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 **??**.

For technical reasons, 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 [**?**] and was also used by Birkedal et al. [**?**].

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 lemma uses the method of Birkedal and Bizjak [**?**]; Birkedal et al. [**?**] 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 $\blacktriangleright$ operator can be safely ignored).

Theorem 1. *There exists a c.o.f.e. (complete ordered family of equivalences)* Wor *and preorders* $\sqsupseteq^{priv}$ *and* $\sqsupseteq^{pub}$ *such that* $(\text{Wor}, \sqsupseteq^{priv})$ *and* $(\text{Wor}, \sqsupseteq^{pub})$ *are preordered c.o.f.e.'s, and there exists an isomorphism* ξ *such that*

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

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

In the above theorem, State $\times$ Rels corresponds to the aforementioned state transition system where Rels contains pairs of relations corresponding to the public and private transitions, and State is an unspecified set that we assume to contain at least 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 reflects 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: $\text{UPred}(\text{MemSeg}) = \{A \subseteq \mathbb{N} \times \text{MemSeg} \mid \forall (n, ms) \in A. \forall m \leq n. (m, ms) \in A\}$.

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:

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

Future Worlds The future world relations model how memory may evolve over time. The *public future world* $W' \sqsupseteq^{pub} W$ requires that $\text{dom}(W') \supseteq \text{dom}(W)$ and $\forall r \in \text{dom}(W).\, W'(r) \sqsupseteq^{pub} W(r)$. 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:

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

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

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:

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

$$\frac{r \in \text{Region}}{r \sqsupseteq^{priv} \text{revoked}}$$

Here, revocation of temporary regions is allowed. In fact, temporary regions can be replaced by an arbitrary other 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.

Notice that the public future region relation is a subset of the private future region relation.

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.

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

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

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

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

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

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

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

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

$$\begin{array}{rl} \mathcal{V}(W) \stackrel{def}{=} & \{(n, i) \mid i \in \mathbb{Z}\} \cup \{(n, ((\text{O}, g), b, e, a))\} \cup \\ & \left\{ (n, ((\text{RW}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{nwl}, g)(W) \end{array} \right\} \cup \\ & \{(n, ((\text{E}, g), b, e, a)) \mid (n, (b, e, a)) \in enterCond(g)(W)\} \cup \\ & \left\{ (n, ((\text{RWLX}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n, (b, e)) \in readCond(g)(W) \text{ and} \\ (n, (b, e)) \in writeCond(\iota^{pwl}, g)(W) \text{ and} \\ (n, (\{\text{RWLX}, \text{RWX}, \text{RX}\}, b, e)) \in execCond(g)(W) \end{array} \right\} \\ & \cup \ldots \textit{ and so on for permissions } \text{RO}, \text{ RWL}, \text{ RX}, \textit{ and } \text{RWX}. \end{array}$$

Figure 6: The logical relation.

4.2 Logical Relation

The logical relation defines semantically when values, program counters, and configurations are capability safe. The definition is found in Figures **??** and **??** 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 technical appendix [**?**].

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. [**?**]'s logical relation for an untyped language, but unlike typical logical relations for typed languages, which require that programs do not fail.

$$readCond(g)(W) = \left\{ (n,(b,e)) \;\middle|\; \begin{array}{l} \exists r \in \mathit{localityReg}(g,W). \\ \quad \exists [b',e'] \sqsupseteq [b,e].\, W(r) \overset{n}{\lessapprox} \iota^{pwl}_{b',e'} \end{array} \right\}$$

$$writeCond(\iota,g)(W) = \left\{ (n,(b,e)) \;\middle|\; \begin{array}{l} \exists r \in \mathit{localityReg}(g,W). \\ W(r) \text{ is address-stratified and} \\ \quad \exists [b',e'] \sqsupseteq [b,e].\, W(r) \overset{n-1}{\gtrapprox} \iota_{b',e'} \end{array} \right\}$$

$$execCond(g)(W) = \left\{ (n,(P,b,e)) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq W, a \in [b,e], \mathit{perm} \in P. \\ \quad \big(n', ((\mathit{perm},g),b,e,a)\big) \in \mathcal{E}(W') \end{array} \right\}$$

$$enterCond(g)(W) = \left\{ (n,(b,e,a)) \;\middle|\; \begin{array}{l} \forall n' < n.\, \forall W' \sqsupseteq W. \\ \quad (n', ((\textsc{rx}, g), b, e, a)) \in \mathcal{E}(W') \end{array} \right\}$$

$$\text{where } g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub} \text{ and } g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$$

Figure 7: Permission-based conditions

The *register-file relation* $\mathcal{R}$ defines safe register-files as those that contain safe words (i.e. words in $\mathcal{V}$) in all registers but pc. The *expression relation* $\mathcal{E}$ defines that a word is safe to use as a program counter if it can be plugged into a safe register file (i.e. a register file in $\mathcal{R}$) and paired with a memory satisfying the world to become a safe configuration. Note that integers and non-executable capabilities (e.g. RO and E capabilities) are considered safe program counters 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 considering 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 **??**, we define such a condition for each kind of permission a capability can have.

For capabilities with read permission, the *readCond* ensures that it can only be used to read safe words, i.e. words in the value relation. To guarantee this, we require that the addressed memory is governed by a region $W(r)$ that imposes safety as a requirement on the values contained. This safety requirement is formulated in terms of a standard region $\iota^{pwl}_{b,e}$. The definition of that standard region is omitted for space reasons, but it simply requires all the words in the range $[b,e]$ to be safe, i.e. in the value relation. Requiring that $W(r) \overset{n}{\lessapprox} \iota^{pwl}_{b,e}$ means that $W(r)$ must accept only safe values like $\iota^{pwl}_{b,e}$, but can be even more restrictive if desired. The read condition also takes into account the locality of the capability because, generally speaking, global capabilities should only depend on permanent regions. Concretely, we use the function $\mathit{localityReg}(g,W)$, which projects out all active (non-revoked) regions when the locality g is local, but only the permanent regions when g is global. The the definition of the

standard region $\iota^{pwl}_{b,e}$ can be found in [**?**]; it makes use of the isomorphism from Theorem **??**.

For a capability with write permission, *writeCond* must be satisfied for the capability's range of authority. An adversary can use such a capability to write any word they can get a hold of, and we can safely assume that they can only get a hold of safe words, so the region governing the relevant memory must allow any safe word to be written there. In order to make the logical relation as liberal as possible, we make this a lower bound of what the region may allow. For write capabilities, we also have to take into account the two flavours of write permissions: write and write-local. In the case of write-local capabilities, the region needs to allow (at least) any safe word to be written, but in the case of write capabilities, the capability cannot be used to write local capabilities, so the region only needs to allow safe non-local values. In the write condition, this is handled by parameterizing it with a region. For the write-local capabilities the write condition is applied with the standard region $\iota^{pwl}_{b,e}$ that we described previously. For the write capabilities we use a different standard region $\iota^{nwl}_{b,e}$ which requires that the words in $[b, e]$ are non-local and safe. As before, we use *localityReg* to pick an appropriate region based on the capability's locality. 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", that is every memory segment where each address contains a value from one of the two accepted memory segments. An interesting property of the write condition is that they prohibit global write-local capabilities which, as discussed in Section **??**, is necessary for any safe use of local capabilities.

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. In contrast, enter capabilities are opaque and can only be used to jump to the address they point to. 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 1 (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* $(n, w) \in \mathcal{V}(W')$*.*

4.3 Safety of the Capability Machine

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

- $perm = \text{RX}$ *and* $(n, (b, e)) \in readCond(g)(W)$
- $perm = \text{RWX}$ *and* $(n, (b, e)) \in readCond(g)(W)$ *and* $(n, (b, e)) \in writeCond(\iota^{nwl}, g)(W)$
- $perm = \text{RWLX}$ *and* $(n, (b, e)) \in readCond(g)(W)$ *and* $(n, (b, e)) \in writeCond(\iota^{pwl}, g)(W)$,

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

The permission based conditions of Theorem **??** 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, an instance of a general property called capability safety [**?**,**?**]. To understand this, consider that the theorem says the capability $((perm, g), b, e, a)$ is safe as a program counter, without any assumption about what instructions it actually 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 fine and will not be able to go beyond the authority of the values it has access to. We demonstrate this in Section **??** where Theorem **??** is used to reason about capabilities that point to arbitrary instructions. The relation between Theorem **??** and local-state encapsulation and control-flow correctness, will also be shown by example in Section **??** as the examples depend on these properties for correctness. See the technical appendix [**?**] for a detailed proof (by induction over the step-index n) of the theorem.

5 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 **??** are deployed to ensure these properties. Since we are dealing with assembly language, there are many details to the formal treatment, and therefore we necessarily omit some details in the lemma statements. The examples may look deceivingly short, but it is because they use the macro instructions described in Section **??**. The examples would be unintelligible without the macros, as each macro expands to multiple

```
f1: push 1                    f2: malloc r_l 1
    fetch r_1 adv                 store r_l 1
    scall r_1([],[])              fetch r_1 adv
    pop r_1                       call r_1([],[r_l])
    assert r_1 1                  assert r_l 1
    halt                          halt
```

Figure 8: Two example programs that rely on local-state encapsulation. `f1` uses our stack-based calling convention. `f2` does not rely on a stack.

basic instructions. The interested reader can find all the technical details in the technical appendix [**?**].

5.1 Encapsulation of Local State

`f1` and `f2` in Figure **??** 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 then test whether the local state is unchanged. 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 an activation-record based calling convention (described in the technical appendix) to run the adversarial code.

For both programs, we can prove that if they are linked with an adversary, *adv*, that is allowed to allocate memory but has no other capabilities, then the assertion will never fail during executing (see Lemmas **??** and **??** 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. The correctness lemma for `f1` then states:

Lemma 2. *Let*

$$\begin{aligned} c_{adv} &\stackrel{def}{=} ((\text{E}, \text{global}), \dots) & c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{local}), \dots) \\ c_{f1} &\stackrel{def}{=} ((\text{RWX}, \text{global}), \dots) & c_{link} &\stackrel{def}{=} ((\text{RO}, \text{global}), \dots) \\ c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{global}), \dots) & reg &\in \text{Reg} \\ m &\stackrel{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame} \end{aligned}$$

where each of the capabilities have an appropriate range of authority and pointer[4]*. Furthermore*

[4] These assumptions are kept intentionally vague for brevity. Full statements are in the technical appendix [**?**].

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

To prove Lemma **??**, 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 an anti-reduction lemma, it suffices to show that the configuration is safe after some reduction steps. We then use a general lemma for reasoning about `scall`, by which it suffices to show that (1) the configuration that `scall` will jump to is safe and (2) that the configuration just after `scall` is done cleaning up is safe. We use the Fundamental Theorem to reason about the unknown adversarial code, but notice that the adversary capability is an enter capability, which the Fundamental Theorem says nothing about. Luckily the enter capability becomes RX after the jump and then the Fundamental Theorem applies.

We have a similar lemma for `f2`:

Lemma 3. *Making similar assumptions about capabilities and linking as in Lemma* **??** *but assuming no stack pointer, if* $(reg[\text{pc} \mapsto c_{f2}], m) \to^* (halted, m')$, *then* $m'(flag) = 0$.

5.2 Well-Bracketed Control-Flow

Using the stack-based calling convention of `scall`, we get well-bracketed control-flow. To illustrate this, we look at two example programs `f3` and `g1` in Figure **??**.

In `f3` there are two calls to an adversary and in order for the assertion in the middle to succeed, they need to 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 **??** prevent this attack: specifically, the return pointer is local, so it can only be stored on the stack, but the part of the stack that is accessible to the adversary is cleared before the second invocation. 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 4. *Making similar assumptions about capabilities and linking as in Lemma* **??** *if* $(reg[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \to^* (halted, m')$, *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) [**?**, **?**]. It consists of two parts, `g1` and `f4`. `g1` is a closure generator that generates closures with one variable x set to 0 in its environment and `f4` as the program (note we can omit some calling convention security measures because the stack is not used

```
g1:  malloc r2 1                          (continued from previous column)
     store r2 0                           store x 0
     move pc r3                           scall r1([],[r0, r1, renv])
     lea r3 offset                        store x 1
     crtcls [(x, r2)] r3                  scall r1([],[r0, renv])
     rclear RegName \ {pc, r0, r1}        load r1 x
     jmp r0                               assert r1 1
f4:  reqglob r1                           mclear rstk
     prepstk rstk                         rclear RegName \ {r0, pc}
     (continues in next column)           jmp r0

f3:  push 1
     fetch r1 adv
     scall r1([],[r1])
     pop r2
     assert r2 1
     push 2
     scall r1([],[])
     halt
```

Figure 9: Two programs that rely on well-bracketedness of `scall`s to function correctly. *offset* is the offset to `f4`.

in the closure generator). `f4` expects one argument, a callback. It sets x to 0 and calls the callback. When it returns, it sets x to 1 and calls the callback a second time. When it returns again, it asserts x is 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 **??**, this means that we need to check (1) that the stack pointer that the closure receives from the adversary has write-local permission and (2) that the adversary callback is global.

To illustrate how subtle this program is, consider how an adversary could try to make the assertion fail. In the second callback an adversary can get to the first callback by invoking the closure one more time. If there were any way for the adversary to transfer the return pointer from the point where it reinvokes the closure to where the closure reinvokes the callback, then the assertion could be made to fail. Similarly, if there were any way for the adversary to store a stack pointer or trick the trusted code into preserving it across an invocation, the assertion can likely be made to fail too. However, our calling convention prevents any of this from happening, as we prove in the following lemma.

Lemma 5. *Let*

$$c_{adv} \stackrel{def}{=} ((\text{RWX}, \text{global}), \dots)\ c_{g1} \stackrel{def}{=} ((\text{E}, \text{global}), \dots)$$

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

As explained in Section **??**, the macro-instruction `reqglob` r_1 checks that the callback is global, essentially to make sure it is not allocated on the stack where it might contain old stack pointers or return pointers. Otherwise, the encapsulation of our local stack frame could be broken. In the proof of Lemma **??**, this requirement shows up because we invoke the callback in a world that is only a private future world of the one where we received the callback, precisely because

we have invalidated the adversary's local state (particularly their old stack and return capabilities). The callback is still valid in this private future world, but only because we know that it is global.

In Lemma **??** the order of control has been inverted compared to the previous lemmas. In this lemma, the adversary assumes control first with a capability for the closure creator `g1`. Consequently, we need to check that all arguments are safe to use and that we clean up before returning in the end. The inversion of control poses an interesting challenge when it comes to reasoning about the adversary's local state during the execution of `f4` and the callbacks where the adversary should not rely on the local state from before the call of `f4`. This is easily done by revoking all the temporary regions of the world given at the start of `f4`. However, when `f4` returns, the adversary is again allowed to rely on its old local state so we need to guarantee that the local state is unchanged. This is important because the return pointer that `f4` receives may be local, and the adversary is allowed to allocate the activation record on the stack (just like we do) so they can store and recover their old stack pointer after `f4` returns. By utilizing the reinstation mechanism of the future world relation as well as our knowledge of the future worlds used, we can construct a world in which the adversary's invariants are preserved. The details of this and the proofs of the other lemmas are found in the technical appendix [**?**].

6 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. [**?**], do not take into account temporal properties, like the well-bracketedness that Example `g1` relies on. In fact, our examples show that our logical relation imply a stronger form of control-flow correctness than such formulations, although this is not made very explicit. As future work, we consider looking at a more explicit and useful way to formalize control-flow correctness. The idea would be to define a variant of our capability machine with call and return instructions and well-bracketed control flow built-in to the operational semantics, and then prove that compiling such code to our machine using our calling convention is fully abstract [**?**].

Performance and the requirement for stack clearing The additional security measures of the calling convention described in Section **??** impose an overhead compared to a calling convention without security guarantees. However, most of our security measures 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 are the requirements for stack clearing 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. We believe this is actually a common requirement for typical usage scenarios of local capabilities and capability machines like CHERI should consider to provide special support for this requirement, in the form of a highly-optimized instruction for erasing a large block of memory. Nevertheless, from a discussion with the designers of the CHERI capability machine, we gather that it is not immediately clear whether and how such a primitive could be implemented efficiently in the CHERI context.

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 safe arguments. 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.

Logical relation

Single orthogonal closure The definitions of $\mathcal{E}$ and $\mathcal{V}$ in Figure **??** apply a single orthogonal closure, a new variant of an existing pattern called biorthogonality. Biorthogonality is a pattern for defining logical relations [**?**, **?**] 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 plugging 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, where 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 [**?**] also use a step-indexed, Kripke logical relation for an assembly language (for reasoning about correct compilation from ML to assembly), but 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. [**?**] (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. [**?**]'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.

7 Related Work

Finally, we summarize how our work relates to previous work. We do not repeat the work we discussed in Section **??**.

Capability machines originate with Dennis and Van Horn [**?**] and we refer to Levy [**?**] and Watson et al. [**?**] for an overview of previous work. The capability machine formalized in Section **??** is a simple but representative model, modeled mainly after the M-Machine [**?**] (the enter pointers resemble the M-Machine's) and CHERI [**?**, **?**] (the memory and local capabilities resemble CHERI's). The latter is a recent and relatively mature capability machine, which 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* [**?**]. Those use a quite different attacker model than us: they assume an attacker that is not able 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 **??**, 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 [**?**, **?**]. 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) and 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 [**?**, **?**, **?**, **?**], closely related to the one used by Devriese et al. [**?**] 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 [**?**], public/private future worlds [**?**] (see Section **??** for a discussion), and biorthogonality [**?**, **?**, **?**].

Swasey et al. [**?**] have recently developed a *logic*, OCPL, for verification of object capability patterns. The logic is based on Iris [**?**, **?**, **?**], a state of the art higher-order concurrent separation logic and is formalized in Coq, building on the Iris Proof Mode for Coq [**?**]. 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. [**?**].

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

Acknowledgements. 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 holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).