JFP **31**, e9, 75 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S095679682100006X

StkTokens: *Enforcing well-bracketed control flow and stack encapsulation using linear capabilities*

LAU SKORSTENGAARD[*]
Toitware, Aarhus, Denmark
(*e-mail:* lau.skorstengaard@gmail.com)

DOMINIQUE DEVRIESE
Department of Computer Science, Vrije Universiteit Brussel, Brussels, Belgium
(*e-mail:* dominique.devriese@vub.be)

LARS BIRKEDAL
Department of Computer Science, Aarhus University, Aarhus, Denmark
(*e-mail:* birkedal@cs.au.dk)

Abstract

We propose and study StkTokens: a new calling convention that provably enforces well-bracketed control flow and local state encapsulation on a capability machine. The calling convention is based on linear capabilities: a type of capabilities that are prevented from being duplicated by the hardware. In addition to designing and formalizing this new calling convention, we also contribute a new way to formalize and prove that it effectively enforces well-bracketed control flow and local state encapsulation using what we call a fully abstract overlay semantics.

1 Introduction

Secure compilation is an active topic of research (see, e.g., New *et al.*, 2016; Juglaret *et al.*, 2016; Devriese *et al.*, 2017; Patrignani & Garg, 2017; Abate *et al.*, 2018; Patrignani *et al.*, 2019; Barthe *et al.*, 2019), but real secure compilers are still a rare sight. Secure compilers preserve source-language (security-relevant) properties even when the compiled code interacts with arbitrary target-language components. Generally, properties that hold in the source language but not in the target language need to be somehow enforced by the compiler. Two properties that hold in many high-level source languages are well-bracketed control flow and encapsulation of local state, but they are usually not enforced after compilation to assembly.

Well-bracketed control flow (WBCF) expresses that invoked functions must either return to their callers, invoke other functions themselves, or diverge and generally holds in programming languages that do not offer a primitive form of continuations (or related features). At the assembly level, this property does not hold. Invoked functions get direct

[*]Research performed while the author was affiliated with Aarhus University.

access to return pointers that they are supposed to jump to a single time at the end of their execution. There is, however, no guarantee that untrusted assembly code respects this intended usage. In particular, adversarial code may invoke return pointers that were intended to be called from other stack frames than theirs: either from frames higher in the call stack or from ones that no longer exist as they have already returned.

Local state encapsulation (LSE) is the guarantee that when a function invokes another function, its local variables (saved on its stack frame) will not be read or modified until the invoked function returns. At the assembly level, this property also does not hold. The calling function's local variables are stored on the stack during the invocation, and functions are not supposed to touch stack frames other than their own. However, untrusted assembly code is free to ignore this requirement and read or overwrite the local state of other stack frames.

To enforce these properties, target language security primitives are needed to prevent untrusted code from misbehaving without imposing too much overhead on well-behaved code. The security primitives based on virtual memory on commodity processors do not seem sufficiently fine grained to efficiently support this. More suitable security primitives are offered by a type of CPUs known as capability machines (Levy, 1984; Watson *et al.*, 2015a). These processors use tagged memory to enforce a strict distinction between integers and *capabilities*: pointers that carry authority. Capabilities come in different flavors. Memory capabilities allow reading from and writing to a block of memory. Additionally, capability machines offer some form of *object capabilities* that represent low-level encapsulated closures, i.e. a piece of code coupled with private state that it gains access to upon invocation. The concrete mechanics of object capabilities vary between different capability machines. For example, on a recent capability machine called CHERI they take the form of pairs of capabilities that represent the code and data parts of the closure. Both capabilities are sealed with a common seal. This makes them opaque and unusable until they are invoked. When they are invoked, the hardware, or a special OS-provided exception handler, transparently unseals the pair (Watson *et al.*, 2015b, 2016).

To enforce WBCF and LSE on a capability machine, there are essentially two approaches. The first is to use separate stacks for mutually distrusting components, and a central, trusted stack manager that mediates cross-component invocations. This idea has been applied in CheriBSD (an operating system built on CHERI) (Watson *et al.*, 2015b), but it is not without downsides. First, it requires reserving separate stack space for all components, which scales poorly to large amounts of components. Also, in the presence of higher-order values (e.g., function pointers, objects), the stack manager needs to be able to decide which component a higher order value belongs to in order to provide it the right stack pointer upon invocation. It is not clear how it can do this efficiently in the presence of large amounts of components. Finally, this approach does not allow passing stack references between components.

A more scalable approach retains a single stack shared between components. Enforcing WBCF and LSE in this approach requires a way to temporarily provide stack and return capabilities to an untrusted component and to revoke them after it returns. While capability revocation is expensive in general, some capability machines offer restricted forms of revocation that can be implemented efficiently. For example, CHERI offers a form of *local* capabilities that can only be stored in registers or on the stack but not in other parts of memory. Skorstengaard *et al.* (2018a) have demonstrated that by making the stack and return pointer local, and by introducing a number of security checks and measures, the two properties can be guaranteed. In fact, a similar system was envisioned in earlier work on CHERI (Watson *et al.*, 2012). However, a problem with this approach is that revoking the local stack and return capabilities on every security boundary crossing requires clearing the entire unused part of the stack, an operation that may be prohibitively expensive (although the hardware could optimize the clearing to a significant extent Joannou *et al.*, 2017).

In this work, we propose and study StkTokens: an alternative calling convention that enforces WBCF and LSE with a single shared stack. Instead of CHERI's local capabilities, it builds on *linear* capabilities; a new form of capabilities that has not previously been described in the published literature, although related ideas have been described by Szabo (1997, 2004, "scarce objects") and in technical documents. Concurrently with our work, Watson *et al.* have developed a (more realistic) design for linear capabilities in CHERI that is detailed in the latest CHERI ISA reference (Watson *et al.*, 2020). The hardware prevents these capabilities from being duplicated. We propose to make stack and return pointers linear and make components hand them out in cross-component invocations and require them back on return. The nonduplicability of linear capabilities together with some security checks allows us to guarantee WBCF and LSE without large overhead on boundary crossings and in particular without the need for clearing large blocks of memory. To avoid confusion, it is worth pointing out that our linear capabilities are really affine rather than linear: erasing them is allowed. However, we have chosen to use the term "linear" to align with common usage in the world of linear type systems.

A second contribution of this work is the way in which we formulate these two properties. Although the terms "well-bracketed control flow" and "local state encapsulation" sound precise, it is actually far from clear what they mean and how best to formalize them. Existing formulations are either partial and not suitable for reasoning (Abadi *et al.*, 2005a) or lack evidence of generality (Skorstengaard *et al.*, 2018a). We propose a new formulation using a technique we call *fully abstract overlay semantics*. It starts from the premise that security results for a calling convention should be reusable as part of a larger proof of a secure compiler. To this end, we define two operational semantics for our target language: the first one is unsurprising with just a register bank and a linear memory, but the second features a native well-bracketed call stack and primitive ways to do calls and returns. This second, well-behaved semantics guarantees WBCF and LSE natively for components using our calling convention. As such, these components can be sure that they will only ever interact with other well-behaved components that respect our desired properties. To express security of our calling convention, we then show that considering the same components in the original semantics does not give adversaries additional ways to interact with them. More formally, we show that mapping a component in the well-behaved semantics to the same component in the original semantics is fully abstract (Abadi, 1999), i.e. components are indistinguishable to arbitrary adversaries in the well-behaved language if they are indistinguishable to arbitrary adversaries in the original language.

Compared to Skorstengaard *et al.* (2018a) that prove LSE and WBCF for a handful of examples, this approach expresses what it means to enforce the desirable properties in a general way and makes it clear that we can support a very general class of programs. Additionally, formulating security of a calling convention in this way makes it potentially reusable in a larger security proof of a full compiler. The idea is that such a compiler could be proven fully abstract with respect to the well-behaved semantics of the target language, so that the proof could rely on native well-bracketedness and local stack frame encapsulation. Such an independent result could then be composed with ours to obtain security of the compiler targeting the real target language, by transitivity of full abstraction.

In this paper, we make the following contributions:

- We present LCM: A formalization of a simple CHERI-like capability machine with linear capabilities (Section 2).
- We present a new calling convention StkTokens that provably guarantees LSE and WBCF on LCM (Section 3).
- We present a new way to formalize these guarantees based on a novel technique called *fully-abstract overlay semantics* and we prove LSE and WBCF claims. This includes
  - oLCM: an overlay semantics for LCM with built-in LSE and WBCF (Section 4).
  - proving full-abstraction for the embedding of oLCM into LCM (Section 5) by
  - using and defining a cross-language, step-indexed, Kripke logical relation with recursive worlds (Section 5).

This text is an extended version of a paper that was presented at POPL 2019 (Skorstengaard *et al.*, 2019b). Compared to the earlier text, this version adds and explains aspects of our work that were left out in the conference version due to space restrictions. The added details include a better motivation of sealing (Section 2.1), the requirements of well formedness (Section 4.2) and reasonability of components (Section 4.3). The section on proving full abstraction (Section 5) has been rewritten to better explain the method used for the full-abstraction proof. This includes a description and motivation of the Kripke worlds (Section 5.1) and logical relation (Section 5.3) that we use to do this. This paper is accompanied by a technical report (Skorstengaard *et al.*, 2018b) with technical details and proofs.

Generally, we have only added material that we believe is valuable to some readers, and we have worked hard to explain the more technical material and make it digestible. Additionally, while Sections 2, 3, 4, 6, and 7 are intended for all readers, we have kept the more technical sections about the proof of full abstraction separate in Sections 5 and particularly Section 5.2, so that it can be easily skipped by readers who prefer to do so.

2 A capability machine with sealing and linear capabilities

In this section, we introduce a simple but representative capability machine with linear capabilities, which we call Linear Capability Machine (LCM). LCM is mainly inspired by CHERI (Watson *et al.*, 2015a) with linear capabilities as the main addition. For simplicity, LCM assumes an infinite address space and unbounded integers.[1]

[1] Although we have not thoroughly investigated, we do not believe our results depend heavily on these assumptions and we expect they could be lifted without a great impact on the proofs.

$$
\begin{array}{rrcl}
\mathrm{a}, \mathrm{base} \in & \mathrm{Addr} & \overset{def}{=} & \mathbb{N} \\
\mathrm{end} & & \in & \mathrm{Addr} \cup \{\infty\} \\
\mathrm{perm} \in & \mathrm{Perm} & ::= & \mathrm{RWX} \mid \mathrm{RX} \mid \mathrm{RW} \mid \mathrm{R} \mid 0 \\
\mathrm{l} & & ::= & \mathrm{linear} \mid \mathrm{normal} \\
\sigma_{\mathrm{base}}, \sigma \in & \mathrm{Seal} & \overset{def}{=} & \mathbb{N} \\
\sigma_{\mathrm{end}} & & \in & \mathrm{Seal} \cup \{\infty\} \\
sc \in & \mathrm{Sealables} & ::= & ((\mathrm{perm}, \mathrm{l}), \mathrm{base}, \mathrm{end}, \mathrm{a}) \mid \mathrm{seal}(\sigma_{\mathrm{base}}, \sigma_{\mathrm{end}}, \sigma) \\
c \in & \mathrm{Cap} & ::= & \mathrm{Sealables} \mid \mathrm{sealed}(\sigma, \mathrm{sc}) \\
w \in & \mathrm{Word} & \overset{def}{=} & \mathbb{Z} \uplus \mathrm{Cap} \\
r \in & \mathrm{RegName} & ::= & \mathrm{pc} \mid \mathrm{r}_{\mathrm{rdata}} \mid \mathrm{r}_{\mathrm{rcode}} \mid \mathrm{r}_{\mathrm{stk}} \mid \mathrm{r}_{\mathrm{data}} \mid \mathrm{r}_{\mathrm{t1}} \mid \mathrm{r}_{\mathrm{t2}} \mid \ldots \\
reg \in & \mathrm{RegFile} & \overset{def}{=} & \mathrm{RegName} \rightarrow \mathrm{Word} \\
mem \in & \mathrm{Memory} & \overset{def}{=} & \mathrm{Addr} \rightarrow \mathrm{Word} \\
ms \in & \mathrm{MemFrag} & \overset{def}{=} & \mathrm{Addr} \rightharpoonup \mathrm{Word} \\
\Phi \in & \mathrm{ExecConf} & \overset{def}{=} & \mathrm{Memory} \times \mathrm{RegFile} \\
& \mathrm{Conf} & \overset{def}{=} & \mathrm{ExecConf} \cup \{\mathrm{failed}\} \cup \{\mathrm{halted}\}
\end{array}
$$

$$
\begin{array}{ll}
r \in \mathrm{RegisterName} & \qquad\qquad rn ::= r \mid \mathbb{N} \\
\mathrm{Instr} ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \texttt{minus}\ r\ rn\ rn \mid \\
& \texttt{lt}\ r\ rn\ rn \mid \texttt{gettype}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \\
& \texttt{cca}\ r\ nrn \mid \texttt{seta2b}\ r \mid \texttt{restrict}\ r\ rn \mid \texttt{cseal}\ r\ r \mid \texttt{xjmp}\ r\ r \mid \texttt{split}\ r\ r\ r\ rn \mid \\
& \texttt{splice}\ r\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Fig. 1. The syntax of our capability machine with seals and linear capabilities.

The concept of a capability is the cornerstone of any capability machine. In its simplest form, a (memory) capability is a permission and a range of authority. The former dictates the operations the capability can be used for, and the latter specifies the range of memory it can act upon. Memory capabilities on LCM are of the form $((perm, lin), base, end, addr)$ (defined in Figure 1 with the rest of the syntax of LCM). Here *perm* is the permission, and $[base, end]$ is the range of authority. The available permissions are read-write-execute (RWX), read-write (RW), read-execute (RX), read-only (R), and null-permission (0) ordered by $\leq$ as illustrated in Figure 2. In addition to the permission and range, capabilities also have a current address *addr* and a linearity *lin*. The current address is a CHERI design choice that makes it easier to implement pointers as capabilities in C-like languages Woodruff *et al.* (2014). The linearity is either normal for traditional capabilities or linear for linear ones. A linear capability is a capability that cannot be duplicated. This is enforced dynamically on the capability machine, so when a linear capability is moved between registers or memory, the source is cleared. The nonduplicability of linear capabilities means that a linear capability cannot become aliased if it was not to begin with.

Any reasonable capability machine needs a way to set up boundaries between security domains with different authorities. It also must have a way to cross these boundaries such that (1) the security domain we move from can encapsulate itself and later regain its authority and (2) the security domain we move to regains all of its authority. On LCM we have CHERI-like sealed capabilities to achieve this (Watson *et al.*, 2015a, 2016). A sealed

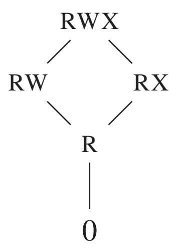

Fig. 2. Permission hierarchy.

capability sealed(σ, sc) is a pair of a seal σ (represented simply as a natural number) and a capability sc. A sealed capability makes an underlying capability opaque, which means that the underlying capability cannot be changed or used for the operations it normally gives permission to. In other words, the authority the underlying capability represents is encapsulated under the seal. In order to seal a capability with a seal σ, it is necessary to have the authority to do so. The permission to make sealed capabilities is represented by a second form of capability (in addition to the memory capabilities we saw above): a set-of-seals capability seal($\sigma_{base}, \sigma_{end}, \sigma_{current}$). Such a capability represents the authority to seal other capabilities with seals in the range $[\sigma_{base}, \sigma_{end}]$. In the spirit of memory capabilities, a set-of-seals capability has a current seal $\sigma_{current}$ that is selected for use in the next seal operation. As we will see later, sealed capabilities get unsealed when they are invoked using an `xjmp`, an operation that operates on a pair of capabilities sealed with the same seal. The instruction will be explained in more detail below, but essentially, it unseals the pair of capabilities, transfers control to one of them (the code part of the pair), and makes the other one (the data part) available to the invoked code. The combination of sealed capabilities and `xjmp` gives us the properties (1) and (2) mentioned above: encapsulation of components' authority such that authority is restored upon invocation.

Words on LCM are either capabilities or data (represented by integers $\mathbb{Z}$). We assume a finite set of register names RegName containing at least the registers pc, $\mathrm{r}_{\mathrm{rdata}}$, $\mathrm{r}_{\mathrm{rcode}}$, $\mathrm{r}_{\mathrm{stk}}$, $\mathrm{r}_{\mathrm{data}}$, r_{t1}, and r_{t2}. We define register files as functions from register names to words. Complete memories map all addresses to words and memory fragments map some addresses to words (i.e. they are partial functions). LCM has two terminated configurations halted and failed that, respectively, signify a successful execution and an execution where something went wrong, e.g., an out-of-bounds memory access. An executable configuration is a register file and memory pair.

LCM's instruction set is somewhat basic with the instructions one expects on most low-level machines as well as capability-related instructions. The standard instructions are unconditional and conditional jump (`jmp` and `jnz`), copy between registers (`move`), instructions that load from memory and store to memory (`load` and `store`), and arithmetic operations (`plus`, `minus`, and `lt`). The simplest of the capability instructions inspect the properties of capabilities: type (`gettype`), linearity (`getl`), range (`getb` and `gete`), current address or seal (`geta`), or permission (`getp`). The current address (or seal) of a capability (or set-of-seals) can be shifted by an offset (`cca`) or set to the base address (`seta2b`). The `restrict` instruction reduces the permission of a capability according to the permission order $\leq$. Generally speaking, a capability machine needs an instruction for

$$\frac{\Phi(\text{pc}) = ((p, \_), b, e, a) \qquad b \le a \le e \qquad p \in \{\text{RWX}, \text{RX}\}}{\Phi \to [\![decode(\Phi.mem(a))]\!] \, (\Phi)} \qquad\qquad \frac{\forall \Phi' \neq \text{failed}.\ \Phi \not\to \Phi'}{\Phi \to \text{failed}}$$

$$updPc(\Phi) = \begin{cases} \Phi[reg.\text{pc} \mapsto w] & \Phi(\text{pc}) = ((p,l),b,e,a) \land w = ((p,l),b,e,a+1) \\ \Phi & \text{otherwise} \end{cases}$$

$$linClear(w) = \begin{cases} 0 & isLinear(w) \\ w & \text{otherwise} \end{cases}$$

$$xjmpRes(c_1, c_2, \Phi) = \begin{cases} \Phi[reg.\text{pc} \mapsto c_1][reg.\text{r}_{\text{data}} \mapsto c_2] & nonExec(c_2) \\ \text{failed} & \text{otherwise} \end{cases}$$

Fig. 3. An excerpt of the operational semantics of LCM (part 1/2).

reducing the range of authority of a capability. Because LCM has linear capabilities, the instruction for this splits the capability in two (`split`) rather than reducing the range of authority. The reverse is possible using `splice`. Sealables can be sealed using `cseal`, and pairs of sealed capabilities can be unsealed by crossing security boundaries (`xjmp`, see below). Finally, LCM has instructions to signal whether an execution was successful or not (`halt` and `fail`).

The operational semantics of LCM is displayed in Figures 3 and 4. The operational semantics is defined in terms of a step relation that executes the next instruction in an executable configuration Φ which results in a new executable configuration or one of the two terminated configurations. The executed instruction is determined by the capability in the pc register, i.e. $\Phi(\text{pc})$ (we write $\Phi(r)$ to mean $\Phi.reg(r)$). In order for the machine to take a step, the capability in the pc must have a permission that allows execution, and the current address of the capability must be within the capability's range of authority. If both conditions are satisfied, then the word pointed to by the capability is decoded to an instruction which is interpreted relative to Φ. The interpretations of some of the instructions are displayed in Figures 3 and 4. In order to step through a program in memory, most of the interpretations use the function *updPc* which simply updates the capability in the pc to point to the next memory address. The instructions that stop execution or change the flow of execution do not use *updPc*. For instance, the `halt` and `fail` instructions are simply interpreted as the halted and failed configurations, respectively, and they do not use *updPc*.

The `move` instruction simply moves a word from one register to another. It is, however, complicated slightly by the presence of the nonduplicable linear capabilities. When a linear capability is moved, the source register must be cleared to prevent duplication of the capability. This is done uniformly in the semantics using the function *linClear* that returns 0 for linear capabilities and is the identity for all other words. When a word w is transferred on the machine, then the source of w is overwritten with $linClear(w)$ which clears the source if w was linear and leaves it unchanged otherwise. In the case of `move`, the source register rn is overwritten with $linClear(\Phi(rn))$.

The `store` and `load` instructions are fairly standard. They require a capability with permission to write or read (respectively), they check that the capability points within the range of authority. Linear capabilities introduce one extra complication for `load` as it needs to clear the loaded memory address when it contains a linear capability in order to

$i \in$ Instr	$[\![i]\!]$ (Φ)	Conditions
`halt`	halted	
`fail`	failed	
`move` r rn	$updPc($ $\Phi[reg.rn \mapsto w_2]$ $[reg.r \mapsto w_1])$	$rn \in$ RegName and $w_1 = \Phi(rn)$ and $w_2 = linClear(\Phi(rn))$
`load` r_1 r_2	$updPc($ $\Phi[reg.r_1 \mapsto w_1]$ $[mem.a \mapsto w_a])$	$\Phi(r_2) = ((p, \_), b, e, a)$ and $b \leq a \leq e$ and $p \in \{\text{RWX}, \text{RW}, \text{RX}, \text{R}\}$ and $w_1 = \Phi.mem(a)$ and $isLinear(w_1) \Rightarrow p \in \{\text{RWX}, \text{RW}\}$ and $w_a = linClear(w_1)$
`store` r_1 r_2	$updPc($ $\Phi[reg.r_2 \mapsto w_2]$ $[mem.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = ((p, \_), b, e, a)$ and $p \in \{\text{RWX}, \text{RW}\}$ and $b \leq a \leq e$ and $w_2 = linClear(\Phi(r_2))$
`geta` r_1 r_2	$updPc($ $\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, \_, a)$ or $\Phi(r_2) = \text{seal}(\_, \_, a)$, then $w = a$ and otherwise $w = -1$
`cca` r rn	$updPc($ $\Phi[reg.r \mapsto w])$	$\Phi(rn) = n \in \mathbb{Z}$ and either $\Phi(r) = ((p, l), b, e, a)$ or $\Phi(r) = \text{seal}(\sigma_b, \sigma_e, \sigma)$ and $w = ((p, l), b, e, a + n)$ or $w = \text{seal}(\sigma_b, \sigma_e, \sigma + n)$, respectively
`jmp` r	$\Phi[reg.r \mapsto w]$ $[reg.\text{pc} \mapsto \Phi(r)]$	$w = linClear(\Phi(r))$
`xjmp` r_1 r_2	Φ'	$\Phi(r_1) = \text{sealed}(\sigma, c_1)$ and $\Phi(r_2) = \text{sealed}(\sigma, c_2)$ and $w_1 = linClear(c_1)$ and $w_2 = linClear(c_2)$ and $\Phi' = xjmpRes(c_1, c_2, \Phi[reg.r_1, r_2 \mapsto w_1, w_2])$
`split` r_1 r_2 r_3 rn	$updPc(\Phi[reg.r_3 \mapsto w]$ $[reg.r_1 \mapsto c_1]$ $[reg.r_2 \mapsto c_2])$	$\Phi(r_3) = ((p, l), b, e, a)$ and $\Phi(rn) = n \in \mathbb{N}$ and $b \leq n < e$ and $c_1 = ((p, l), b, n, a)$ and $c_2 = ((p, l), n + 1, e, a)$ and $w = linClear(\Phi(r_3))$
`splice` r_1 r_2 r_3	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[reg.r_3 \mapsto w_3]$ $[reg.r_1 \mapsto c])$	$\Phi(r_2) = ((p, l), b, n, \_)$ and $\Phi(r_3) = ((p, l), n + 1, e, a)$ and $b \leq n < e$ and $c = ((p, l), b, e, a)$ and $w_2, w_3 = linClear(\Phi(r_2), \Phi(r_3))$
`cseal` r_1 r_2	$updPc($ $\Phi[reg.r_1 \mapsto sc])$	$\Phi(r_1) \in$ Sealables and $\Phi(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma)$ and $\sigma_b \leq \sigma \leq \sigma_e$ and $sc = \text{sealed}(\sigma, \Phi(r_1))$
	…	
_	failed	otherwise

Fig. 4. An excerpt of the operational semantics of LCM (part 2 of 2).

not duplicate the capability. In this case, we require that the memory capability used for loading also has write-permission.

The instruction `geta` projects the current address (or seal) from a capability (or set-of-seals), and returns -1 for data and sealed capabilities. `cca` (change current address) changes the current address or seal of a capability or set-of-seals, respectively, by a given offset. Note that this instruction does not need to use *linClear* like the previous ones, because it modifies the capability in-place, i.e. the source register is also the target register. The `jmp` instruction is a simple jump that just sets register pc.

Two instructions manipulate seals in LCM: `cseal` for sealing a capability and `xjmp` for unsealing a pair of capabilities. Given a sealable *sc* and a set-of-seals where the current seal σ is within the range of available seals, the `cseal` instruction seals *sc* with σ. Apart from dealing with linearity, `xjmp` takes a pair of sealed capabilities, unseals them, and puts one in the pc register and the other in the $\mathrm{r}_{\mathrm{data}}$ register, but only if they are sealed with the same seal and the data capability (the one placed in $\mathrm{r}_{\mathrm{data}}$) is nonexecutable. A pair of sealed capabilities can be seen as a closure where the code capability (the capability placed in pc) is the program and the data capability is the local environment. Because of the opacity of sealed capabilities, the creator of the closure can be sure that execution will start where the code capability points and only in an environment with the related data, i.e. with access to a data capability sealed with the same seal. This makes `xjmp` the mechanism on LCM that transfers control between security domains. Opaque sealed capabilities encapsulate a security domain's local state and authority, and they only become accessible again when control is transferred to the security domain. Some care should be taken for sealing because reusing the same seal for multiple closures makes it possible to jump to the code of one closure with the environment of another. LCM does not have an instruction for unsealing capabilities directly, but we will explain below how it can be (partially) simulated using `xjmp`.

Instructions for reducing the authority of capabilities are commonplace on capability machines as they allow us to limit a capability's authority before passing it to other code. For normal capabilities, reduction of authority can be done without actually giving up any authority by duplicating the capability first. With linear capabilities, authority cannot be preserved in this fashion as they are nonduplicable. In order to make a lossless reduction of the range of authority, LCM provides special hardware support in the form of `split` and `splice`. The `split` instruction takes a capability with range of authority $[base, end]$ and an address n and creates two new capabilities, with $[base, n]$ and $[n+1, end]$ as ranges of authority. Everything else, i.e. permission, linearity, and current address, is copied without change to the new capabilities. With `split`, we can reduce the range of authority of a linear capability without losing any authority as we retain it in the second capability. The `splice` instruction essentially does the inverse of `split`. Given two capabilities with adjacent ranges of authority and the same permissions and linearity, `splice` splices them together into one capability. The two instructions work in the same way for set-of-seals capabilities. We do not provide special support for lossless reduction of capability permissions, but this could probably be achieved with more fine-grained permissions. This would also allow linear capabilities to have aliases, but only by linear capabilities with disjoint permissions.

The interpretation of the remainder of the instructions is displayed in Appendix A. The instructions `getb`, `gete`, `getl`, and `getp` all query information about capabilities. The getb and gete instructions, respectively, project the base and end address of the range of authority. The linearity of a permission is projected with getl, and finally the permission is projected with getp. The instructions getb and gete also work on set-of-seal capabilities. The instruction gettype returns an integer representation of the type of a word (capability or number). LCM also has arithmetic instructions plus, minus, and lt. The latter instruction compares two numbers and writes 1 or 0 to a target register depending on whether one number is less than the other. The instruction seta2b sets the current address (or seal) of a memory (or set-of-seals) capability to the base address of the range of authority (or range of seals). This instruction makes it easy to work relatively to the base address of a capability. This instruction is not strictly necessary as it can be emulated with other instructions. Finally, we have the restrict instruction which restricts the permission of a capability according to the $\leq$ relation.

2.1 The purpose of sealing

To motivate the necessity of an encapsulation mechanism like sealing, consider the scenario where we are executing some trusted code and want to transfer control to untrusted code. We want to give the untrusted code the means to return to us. That is, we need to give them a return capability. For now, let us pretend that there does not exist a stack and only a single invocation of untrusted code happens in the entire execution.

If we did not have an encapsulation mechanism, our only option would be to give the adversary an executable capability for the address we want them to return to. The untrusted code could use the return capability as intended, but it could also manipulate it to point elsewhere in our code. Jumping to such a capability could cause the trusted program to execute in an unintended and potentially insecure way.

Additionally, when the untrusted code returns, we need to regain access to the capabilities that represent our authority. To achieve this, we have to store them in a location that we can access after the return. However, without an encapsulation mechanism, the untrusted code would also have the same authority as us, including access to the authority we stored away. This is why any reasonable capability machine must have an encapsulation mechanism to allow programs to set up boundaries between security domains.

To establish such a boundary in LCM, we can seal the return capability. Specifically, we can seal the return capability (which points to the instruction to be executed after the return) as a pair, together with a pointer to the stack frame where we have stashed away our capabilities (see further). By doing this, the untrusted code is prevented from changing the target of the return capability forcing them to return to the point we specified. Additionally, they do not get access to the capabilities we stashed away. Nevertheless, the automatic unsealing of sealed capability pairs upon invocation will ensure that the return code can still proceed as before. In other words, the sealing mechanism in LCM allows us to transfer control to untrusted code without giving up our capabilities or handing them over and also control the locations in our code they can jump back to.

It is worth pointing out that LCM does not have a direct unsealing instruction to extract a capability from its sealed version when the seal is available, but one can be (partially) emulated. Say you have a sealed nonexecutable word sealed(σ, w) as well as a set-of-seals capability seal$(\sigma_b, \sigma_e, \sigma)$ that contains σ, i.e. $\sigma \in [\sigma_b, \sigma_e]$. It is possible to extract w in $\mathrm{r}_{\mathrm{data}}$

using `xjmp` in the following way. The idea is to use the seal to construct a sealed version of the pc capability incremented by one, and then perform an `xjmp` to it in combination with the sealed capability. This does not work for sealed executable capabilities because `xjmp` fails if the data capability is executable. However, even though we cannot extract the executable capability from sealed(σ, c), we can still invoke it with arbitrary arguments, since we can use the seal to construct an arbitrary data part and `xjmp` to the combination.[2]

Sealing is meant for encapsulation, but it relies on seals being kept private as should be clear from the explanations so far. For this reason, it is important that trusted code does not leak its seals to adversaries and that the system is initialised so that each component has access to unique seals. We return to this in Section 4.2.

2.2 Decoding and encoding functions

The operational semantics of the capability machine uses the function *decode* to decode instructions. We also assume a function *encode* to make it easy to specify programs in terms of instructions. Rather than defining such a decode function and an encode function, we assume that they are given with certain properties. Most importantly, *encode* : Instr $\rightarrow$ $\mathbb{Z}$ should be the right inverse of *decode*, i.e.

$$\forall i \in \text{Instr}.\ \mathit{decode}(\mathit{encode}(i)) = i$$

The *decode* : Word $\rightarrow$ Instr should only map numbers to non`fail` instructions, i.e. capabilities are mapped to `fail`:

$$\forall c \in \text{Cap}.\ \mathit{decode}(c) = \texttt{fail}$$

These assumptions are sufficient to construct program examples in terms of instructions rather than machine words (using *encode*) and run them on the machine (using the fact that *decode* is the left inverse of *encode*).

The full machine semantics in Appendix A also assumes decode and encode functions for permissions *encodePerm* : Perm $\rightarrow \mathbb{Z}$ and *decodePerm* : $\mathbb{Z} \rightarrow$ Perm. We assume the *decodePerm* function to be the left inverse of *encodePerm*. For *encodePerm*, we assume that it does not encode anything to the `getp` error value -1, i.e.

$$\forall p \in \text{Perm}.\ \mathit{encodePerm}(p) \neq -1$$

For linearity we assume similar functions. Finally in the interpretation of `gettype`, the machine uses an encode function for word types *encodeType*. This function encodes each kind of word as an integer. This is very much like the previous functions. It encodes each kind of word differently and all words of the same kind to the same integer.

2.3 Components, linking, programs, and contexts

The executable configuration describes the machine state, but it does not make it clear what components run on the machine and how they interact with each other. To clarify this, we

[2] Alternatively, we could remove the current restriction in `xjmp` that requires the data capability to be non-executable but then the invoked code would often have to manually perform this check instead.

$$\begin{array}{rlcrl}
s & \in \mathrm{Symbol} & \quad \mathrm{import} ::= a \mapsfrom s & \quad \mathrm{export} & ::= s \mapsto w \\
comp_0 & ::= (ms_{\mathrm{code}}, ms_{\mathrm{data}}, \overline{\mathrm{import}}, \overline{\mathrm{export}}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, A_{\mathrm{linear}}) & & & \\
comp & ::= comp_0 \mid (comp_0, c_{\mathrm{main},c}, c_{\mathrm{main},d}) & & &
\end{array}$$

$$\frac{\begin{array}{c}
comp_0 = (ms_{\mathrm{code},1}, ms_{\mathrm{data},1}, \overline{import_1}, \overline{export_1}, \overline{\sigma_{\mathrm{ret},1}}, \overline{\sigma_{\mathrm{clos},1}}, A_{\mathrm{linear},1}) \\
comp'_0 = (ms_{\mathrm{code},2}, ms_{\mathrm{data},2}, \overline{import_2}, \overline{export_2}, \overline{\sigma_{\mathrm{ret},2}}, \overline{\sigma_{\mathrm{clos},2}}, A_{\mathrm{linear},2}) \\
comp''_0 = (ms_{\mathrm{code},3}, ms_{\mathrm{data},3}, \overline{import_3}, \overline{export_3}, \overline{\sigma_{\mathrm{ret},3}}, \overline{\sigma_{\mathrm{clos},3}}, A_{\mathrm{linear},3}) \\
ms_{\mathrm{code},3} = ms_{\mathrm{code},1} \uplus ms_{\mathrm{code},2} \\
ms_{\mathrm{data},3} = (ms_{\mathrm{data},1} \uplus ms_{\mathrm{data},2})[a \mapsto w \mid (a \mapsfrom s) \in (\overline{import_1} \cup \overline{import_2}), (s \mapsto w) \in \overline{export_3}] \\
\overline{export_3} = \overline{export_1} \cup \overline{export_2} \qquad \overline{import_3} = \{a \mapsfrom s \in (\overline{import_1} \cup \overline{import_2}) \mid s \mapsto \_ \notin \overline{export_3}\} \\
\overline{\sigma_{\mathrm{ret},3}} = \overline{\sigma_{\mathrm{ret},1}} \uplus \overline{\sigma_{\mathrm{ret},2}} \qquad \overline{\sigma_{\mathrm{clos},3}} = \overline{\sigma_{\mathrm{clos},1}} \uplus \overline{\sigma_{\mathrm{clos},2}} \qquad A_{\mathrm{linear},3} = A_{\mathrm{linear},1} \uplus A_{\mathrm{linear},2} \\
\mathrm{dom}(ms_{\mathrm{code},3}) \# \mathrm{dom}(ms_{\mathrm{data},3}) \qquad \overline{\sigma_{\mathrm{ret},3}} \# \overline{\sigma_{\mathrm{clos},3}}
\end{array}}{comp''_0 = comp_0 \bowtie comp'_0}$$

$$\frac{comp''_0 = comp_0 \bowtie comp'_0}{\begin{aligned}(comp''_0, c_{\mathrm{main},c}, c_{\mathrm{main},d}) &= comp_0 \bowtie (comp'_0, c_{\mathrm{main},c}, c_{\mathrm{main},d}) \\ &= (comp_0, c_{\mathrm{main},c}, c_{\mathrm{main},d}) \bowtie comp'_0\end{aligned}}$$

Fig. 5. Components and linking of components.

introduce notions of components and programs from which we construct executable configurations. A component (defined in Figure 5) is basically a program with entry points in the form of imports that need to be linked. It has exports that can satisfy the imports of other components. A base component *comp*$_0$ consists of a code memory fragment, a data memory fragment, a list of imported symbols, a list of exported symbols, two lists specifying the available seals (see Section 4), and a set of all the linear addresses (addresses governed by a linear capability). The import list specifies where in memory imports should be placed, and imports are matched to exports via their symbols. An export associates a word with a symbol. A component is either a library component (without a main entry point) or an incomplete program with a main entry point in the form of a pair of sealed capabilities. The latter can be seen as a program that still needs to be linked with libraries. Components are combined into new components by linking them together, as long as only one has a main function. Two components can be linked when their memories, seals, and linear addresses are disjoint. They are combined by taking the union of each of their constituents. For every import that is satisfied by an export of the other component, the data memory is updated to have the exported word on the imported address. The satisfied imports are removed from the import list in the resulting linked component and the exports are obtained by combining the components' exports.

We can now define the notion of a program as well as a context.

Definition 1 (Programs and Contexts)**.** *A program is a component* $(comp_0, c_{\mathrm{main},c}, c_{\mathrm{main},d})$ *with an empty import list. A context for a component comp is a component comp′ such that comp* $\bowtie$ *comp′ is a program.*

How a program is initialized to create an executable configuration is discussed in Section 4. Some simplifications have been made in this presentation of LCM. These

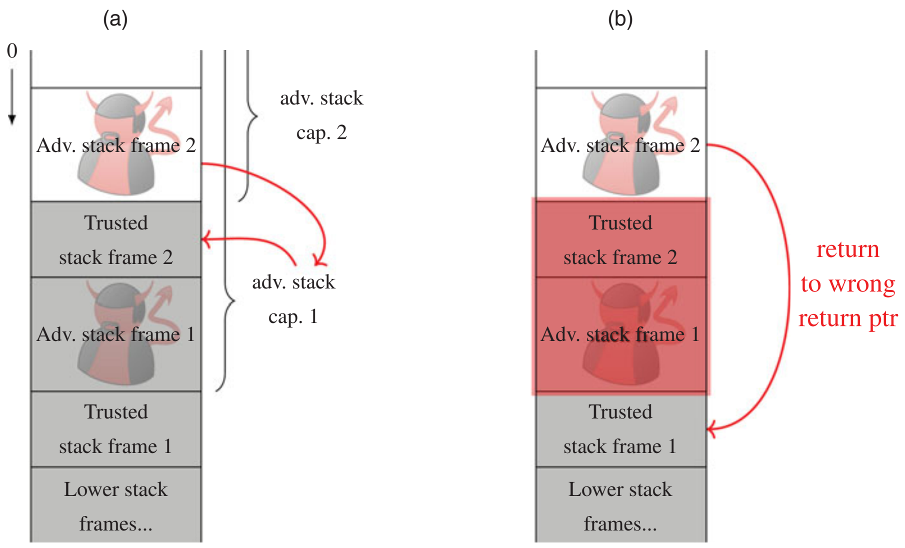

Fig. 6. Possible ways to abuse stack and return capabilities. Grey and white frames indicate inactive and active stack frames respectively. The red frame emphasizes stack frames being skipped by an adversarial return. (a) An adversary uses a previous stack frame's stack pointer. (b) An adversary jumps to a previous stack frame's stack pointer.

simplifications concern minor details, for example, we have omitted the fact that `move`s to the `pc` register are disallowed. Full details can be checked in Skorstengaard *et al.* (2018b).

3 Linear stack and return capabilities

In this section, we introduce our calling convention STKTOKENS that ensures LSE and WBCF. We will gradually explain each of the security measures STKTOKENS takes and motivate them with the attacks they prevent.

STKTOKENS is based on a traditional single stack, shared between all components. To explain the technique, let us first consider how we might already add extra protection to stack and return pointers on a capability machine. First, we replace stack pointers with stack capabilities. When a new stack frame is created, the caller provisions it with a stack capability, restricted to the appropriate range, i.e. not covering the caller's stack frame. Return pointers, on the other hand, are replaced by a pair of sealed return capabilities, as explained in Section 2.1. They form an opaque closure that the callee can only jump to, and when that happens, the caller's data become available to the caller's return code.

While the above adds extra protection, it is not sufficient to enforce WBCF and LSE. Untrusted callees receive a stack capability and a return pair that they are supposed to use for the call. However, a malicious callee (which we will refer to as an adversary[3]) can store the provided capabilities away on the heap in order to use them later. Figure 6 illustrates two examples of this. In both examples, our component and an adversarial component have been taking turns calling each other, so the stack now contains four stack frames

[3] See Section 4.2 for more details on our attacker model.

alternating between ours and theirs. The figure on the left (Figure 6(a)) illustrates how we try to ensure LSE by restricting the stack capability to the unused part before every call to the adversary. However, restricting the stack capability does not help when we, in the first call, give access to the part of the stack where our second stack frame will reside as nothing prevents the adversary from duplicating and storing the stack pointer. Generally speaking, we have no reason to ever trust a stack capability received from an untrusted component as that stack capability may have been duplicated and stored for later use. In the figure on the right (Figure 6(b)), we have given the adversary two pairs of sealed return capabilities, one in each of the two calls to the adversarial component. The adversary stores the pair of sealed return capabilities from the first call in order to use it in the second call where they are not allowed. The figure illustrates how the adversarial code uses the return pair from the first call to return from the second call and thus break WBCF.

As the examples illustrate, this naive use of memory and object capabilities does not suffice to enforce LSE and WBCF. The problem is essentially that the stack and return pointers that a callee receives from a caller remain in effect outside their intended lifetime: either when the callee has already returned or when they have themselves invoked other code. Linear capabilities offer a form of revocation[4] that can be used to prevent this from happening.

The linear capabilities are put to use by requiring the stack capability to be linear. On call, the caller splits the stack capability in two: one capability for their local stack frame and another one for the unused part of the stack. The local stack frame capability is sealed and used as the data part of the sealed return pair. The capability for the remainder of the stack is given to the callee. Because the stack capability is linear, the caller knows that the capability for their local stack frame cannot have an alias. This means that an adversary cannot access the caller's local data because the caller has a linear capability for it and there cannot exist an alias. The caller gives this capability to the adversary only in a sealed form, rendering it opaque and unusable. This is illustrated in Figure 7(a) and prevents the issue illustrated in Figure 6(a).

In a traditional calling convention with a single stack, the stack serves as a call stack keeping track of the order calls were made in and thus in which order they should be returned to. A caller pushes a stack frame to the stack on call and a callee pops a stack frame from the stack upon return. However without any enforcement, there is nothing to prevent a callee from popping more from the stack than they should and returning on an arbitrary call on the call stack. This is exactly what the adversary does in Figure 6(b) when they skip two stack frames. In the presence of adversarial code, we need some mechanism to enforce that the order of the call stack is kept. One way to enforce this would be to hand out a token on call that can only be used when the caller's stack frame is on top of the call stack. The callee would have to present this token on return to prove that it is allowed to return to the caller, and on return the token would be taken back by the caller to prevent it from being spent multiple times. As it turns out, the stack capability for the unused part of the stack can be used as such a token in the following way: On return the callee has to give back the stack capability they were given on invocation. When the caller receives a stack

[4] Revocation in the sense that if we hand out a linear capability and later get it back, then the adversary cannot have kept a copy of it as it is nonduplicable. In other words, the adversary's access has effectively been revoked in this situation.

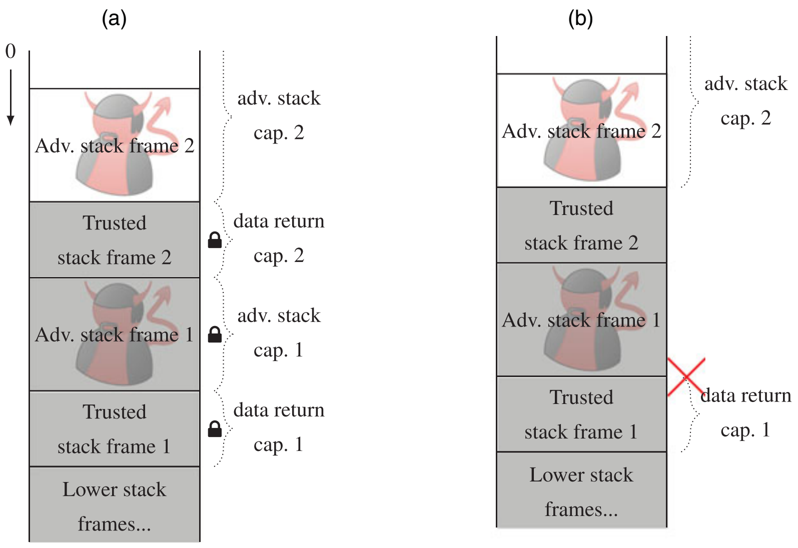

Fig. 7. Abuse of stack and return capabilities prevention. Grey and white frames indicate inactive and active stack frames respectively. Linear braces indicate memory owned by linear capabilities. (a) The non-duplicable linear stack capability for the trusted code's stack frame and the opacity of sealed capabilities ensures LSE. (b) The trusted caller fails to splice the stack capability returned by the adversary with the capability for the trusted caller's local stack frame.

capability back on return, it checks that this token is actually spendable, i.e. whether its stack frame is on top of the logical call stack or not. This check can be done by attempting to recombine (splice) the return token with the stack capability for the local stack frame (which at this point has been unsealed again) in order to reconstruct the original stack capability. If the splice is successful, then the caller knows that the two capabilities are adjacent. On the other hand, if the splice fails, then they are alerted to the fact that their stack frame may not be the topmost. STKTOKENS uses this approach; and as illustrated in Figure 7(b), it prevents the issue in Figure 6(b) as the adversary does not return a spendable token when they return.

In order for a call to have a presence on the call stack, its stack frame must be nonempty. We cannot allow empty stack frames on the call stack, because then it would be impossible to tell whether the topmost nonempty stack frame has an empty stack frame on top of it. Nonempty stack frames come naturally in traditional C-like calling convention as they keep track of old stack pointers and old program counters on the stack, but in STKTOKENS these things are part of the return pair which means that a caller with no local data may only need an empty stack frame. In other words, a caller using STKTOKENS needs to take care that their stack frame is nonempty in order to reserve their spot in the return order. There is also a more practical reason for a STKTOKENS caller to make sure their stack frame is nonempty: They need a fragment of the stack capability in order to perform the splice that verifies the validity of the return token.

At this point, the caller checks that the return token is adjacent to the stack capability for the caller's local stack frame and they have the means to do so. However, this still does not ensure that the caller's stack frame is on top of the call stack. The issue is that stack frames may not be tightly packed leaving space between stack frames in memory. An adversarial

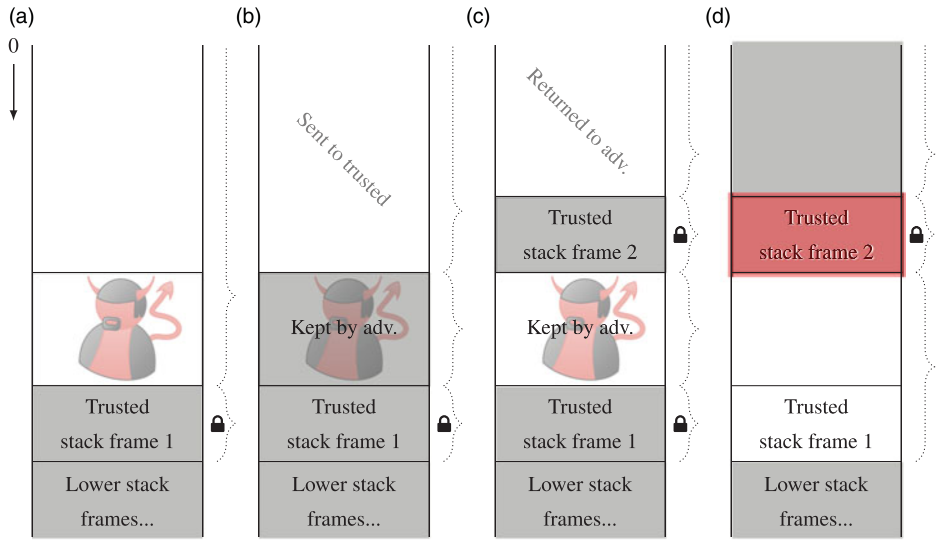

Fig. 8. Partial return token used to return out of order. Grey and white frames indicate inactive and active stack frames respectively. Linear braces indicate memory owned by linear capabilities. The red frame emphasizes a stack frame that is being skipped by an adversarial return.

callee may intentionally leave a bit of space in memory above the caller's stack frame, so that they can later return out of order by returning the bit of the return token for the bit of memory left above the caller's stack frame. This is illustrated in Figure 8: In Figure 8(a), a trusted caller has called an adversarial callee. The adversary calls the trusted code back, but first they split the return token in two and store on the heap the part for the memory adjacent to the trusted caller's call frame (Figure 8(b)). The trusted caller calls the adversary back using the precautions we have described so far (Figure 8(c)). At this point (Figure 8(d)), the adversary has access to a partial return token adjacent to the trusted caller's first stack frame which allows the adversary to return from this call breaking WBCF.

For the caller to be sure that there are no hidden stack frames above its own, they need to make sure that the return token is exactly the same as the one they passed to the callee. In StkTokens, the base address of the stack capability is fixed as a compile-time constant (Note: the stack grows downward, so the base address of the stack capability is the top-most address of the stack). The caller verifies the validity of the return token by checking whether the base address of a returned token corresponds to this fixed base address. In the scenario, we just sketched, the base address check would fail in Figure 8(d), thus alerting the caller to the attempt to break WBCF.

In StkTokens, the stack memory is only referenced by a single linear stack capability at the start of execution. Because of this, the return token can be verified simply by checking its base address and splicing it with the caller's stack frame. There is no need to check linearity because only linear capabilities to this memory exist.

The return pointer in the StkTokens scheme is a pair of sealed capabilities where the code part of the pair is the old program counter, and the data part is the stack capability for the local stack frame of the caller. Both of the capabilities in the pair are sealed with the same seal. All call points need to be associated with a unique seal (a return seal) that is only used for the return capabilities for that particular call point. The return seal is what associates the stack frame on the call stack with a specific call point in a program, so if we allowed return seals to be reused, it would be possible to return to a different call point than the one that gave rise to the stack frame, breaking WBCF. For similar reasons, we cannot allow return seals to be used to seal closures. Return seals should also never be leaked to adversarial code as this would allow them to unseal the local stack frame of a caller and thus break LSE. This goes for direct leaks (leaving a seal in a register or writing it to adversarial memory), as well as indirect leaks (leaking a different capability that can be used for reading, either directly or indirectly, a return seal from memory).

We have sometimes phrased the description of the STKTOKENS calling scheme in terms of “them versus us.” This may have created the impression of an asymmetric calling convention that places a special status on trusted components allowing them to protect themselves against adversaries. However, STKTOKENS is a modular calling scheme: no restriction is put on adversarial components that we do not expect trusted components to meet. Specifically, we will only assume that both trusted and adversarial components are initially syntactically well formed (described in more detail in Section 4.2) which basically just ensures that their initial state does not break machine guarantees (e.g. no aliases for linear capabilities or access to seals of other components). This means that mutually distrusting components can ensure WBCF and LSE for themselves by employing STKTOKENS.

To summarize, STKTOKENS consists of the following measures:

1. Check the base address of the stack capability before and after calls.
2. Make sure that local stack frames are nonempty.
3. Create token and data return capability on call: split the stack capability in two to get a stack capability for your local stack frame and a stack capability for the unused part of the stack. The former is sealed and used as the data part of the return pair. The latter is passed to the callee as the stack pointer.
4. Create code return capability on call: Seal the old program counter capability.
5. Reasonable use of seals: Return seals are only used to seal old program counter capabilities, every return seal is only used for one call site, and they are not leaked (directly or indirectly).

Items 1–4 are captured by the code in Figure 9 , except for checking stack base before calls. We do not include this check because it only needs to happen once between two calls, so that the check after a call suffices if the stack base is not changed subsequently.

4 Formulating security with a fully abstract overlay semantics

As mentioned, the STKTOKENS calling convention guarantees WBCF and LSE. However, before we can prove these properties, we need to know how to even formulate them. Although the properties are intuitively clear and sound precise, formalizing them is actually far from obvious.

```
// Ensure non-empty stack.
 1 :  move r_t1 42
 2 :  store r_stk r_t1
 3 :  cca r_stk (−1)
// Split stack in local stack frame and unused.
 4 :  geta r_t1 r_stk
 5 :  split r_stk r_rdata r_stk r_t1
// Load the call seal.
 6 :  move r_t1 pc
 7 :  cca r_t1 (off_pc − 5)
 8 :  load r_t1 r_t1
 9 :  cca r_t1 off_σ
// Seal the local stack frame.
10 :  cseal r_rdata r_t1
// Construct code return pointer.
11 :  move r_rcode pc
12 :  cca r_rcode 5
13 :  cseal r_rcode r_t1
// Clear tmp registers and jump.
14 :  move r_t1 0
15 :  xjmp r1 r2
// The following is the return code.
// Check that returned stack pointer has base stk_base.
16 :  getb r_t1 r_stk
17 :  minus r_t1 r_t1 stk_base
18 :  move r_t2 pc
19 :  cca r_t2 5
20 :  jnz r_t2 r_t1
21 :  cca r_t2 1
22 :  jmp r_t2
23 :  fail
// Splice with capability for local stack frame.
24 :  splice r_stk r_stk r_data
// Pop 42 from the stack
25 :  cca r_stk 1
// Clear tmp register
26 :  move r_t2 0
```

Fig. 9. The instructions for a $\texttt{call}^{\mathit{off}_{\mathrm{pc}},\mathit{off}_\sigma}\ r_1\ r_2$. Registers r_1 and r_2 are assumed to contain the callee closure as a sealed capability pair. The code uses $\mathrm{r}_{\mathrm{t}1}$ and $\mathrm{r}_{\mathrm{t}2}$ as temporary registers. The number $\mathit{off}_{\mathrm{pc}}$ is the offset between the memory location where the first instruction of the call is stored to the memory location where the set-of-seals capability is stored that should be used for sealing the return capability. This set-of-seals capability may give access to a range of seals and the offset off_σ identifies which of these should be used. stk_base is the globally agreed on stack base. There are some magic numbers in the code: line 1: 42, garbage data to ensure a nonempty stack. Line 7: -5, offset from line 6 (where pc was copied into $\mathrm{r}_{\mathrm{t}1}$) to line 1. Line 12: 5, offset to the return address. Line 19: 5, offset to fail. Line 21: offset to address after fail.

Ideally, we would like to define the properties in a way that is

1. *intuitive*
2. *useful for reasoning:* we should be able to use WBCF and LSE when reasoning about correctness and security of programs using StkTokens.
3. *reusable in secure compiler chains:* for compilers using StkTokens, one should be able to rely on WBCF and LSE when proving correctness and security of other compiler passes and then compose such results with ours to obtain results about the full compiler.
4. *arguably "complete"*: the formalization should arguably capture the entire meaning of WBCF and LSE and should arguably be applicable to any reasonable program.
5. *potentially scalable*: although dynamic code generation and multithreading are currently out of scope, the formalization should, at least potentially, extend to such settings.

Previous formalisations in the literature are formulated in terms of a static control flow graph (e.g., Abadi *et al.*, 2005b). While these are intuitively appealing (1), it is not clear how they can be used to reason about programs (2) or other compiler passes (3), they lack temporal safety guarantees (4) and do not scale (5) to settings with dynamic code generation (where a static control flow graph cannot be defined). Skorstengaard *et al.* (2018a) provide a logical relation that can be used to reason about programs using their calling convention (2,3), but it is not intuitive (1), there is no argument for completeness (4), and it is unclear whether it will scale to more complex features (5).

We contribute a new way to formalize the properties using a novel approach we call fully abstract overlay semantics. The idea is to define a second operational semantics for programs in our target language. This second semantics uses a different abstract machine and different runtime values, but it executes in lock step with the original semantics and there is a very close correspondence between the state of both machines.

The main difference between the two semantics is that the new one satisfies LSE and WBCF by construction: the abstract machine comes with a built-in stack, inactive stack frames are unaddressable and well-bracketed control flow is built into the abstract machine. Important run-time values like return capabilities and stack pointers are represented by special syntactic tokens that interact with the abstract machine's stack, but during execution, there remains a close, structural correspondence to the actual regular capabilities that they represent. For example, stack capabilities in the overlay semantics correspond directly to linear capabilities in the underlying semantics, and they have authority over the part of memory that the overlay views as the stack. The new run-time values in the overlay semantics are treated appropriately in functions like *encodeType*. All the new values correspond to concrete capabilities on the LCM machine which the encoding function reflects. For instance, the encoding of a stack pointer in the overlay semantics is the same as the encoding of a linear memory capability on the LCM machine.

The fact that StkTokens enforces LSE and WBCF is then formulated as a theorem about the function that maps components in the well-behaved overlay semantics to the underlying components in the regular semantics. The theorem states that this function constitutes a fully abstract compiler, a well-known property from the field of secure compilation (Abadi, 1999). Intuitively, the theorem states that if a trusted component interacts with (potentially malicious) components in the regular semantics, then these components have no more expressive power than components which the trusted component interacts with in the well-behaved overlay semantics. In other words, they cannot do anything that doesn't correspond to something that a well-behaved component, respecting LSE and WBCF, can also do. More formally, our full-abstraction result states that two trusted components are indistinguishable to arbitrary other components in the regular semantics if and only if they are indistinguishable to arbitrary other components in the overlay semantics.

Our formal results are complicated by the fact that they only hold on a sane initial configuration of the system and for components that respect the basic rules of the calling convention. For example, the system should be set up so that seals used by components for constructing return pointers are not shared with other components. We envision distributing seals as a job for the linker, so this means our results depend on the linker to do this properly. As another example, a seal used to construct a return pointer can be reused but only to construct return pointers for the same return point. Different seals must be used for different return points. Such seals should also never be passed to other components. These requirements are easy to satisfy: components should request sufficient seals from the linker, use a different one for every place in the code where they make a call to another component, and make sure to clear them from registers before every call. The general pattern is that StkTokens only protects components that do not shoot themselves in the foot by violating a few basic rules. In this section, we define a well-formedness judgment for the syntactic requirements on components as well as a reasonability condition that semantically disallows components to do certain unsafe things. Well formedness is a

$$
\begin{array}{rcl}
\text{Sealables} & ::= & \text{Sealables} \mid \text{stack-ptr(perm, base, end, a)} \mid \\
 & & \text{ret-ptr-data(base, end)} \mid \text{ret-ptr-code(base, end, a)} \\
\text{StackFrame} & \overset{def}{=} & \text{Addr} \times \text{MemFrag} \qquad\qquad \text{Stack} \overset{def}{=} \text{StackFrame}^{*} \\
\text{ExecConf} & \overset{def}{=} & \text{Memory} \times \text{RegFile} \times \text{Stack} \times \text{MemFrag}
\end{array}
$$

$$
\text{Instr} \quad ::= \quad \text{Instr} \mid \texttt{call}^{\mathit{off}_{\mathrm{pc}},\mathit{off}_{\sigma}}\, r\, r \qquad\qquad \mathit{off}_{\mathrm{pc}}, \mathit{off}_{\sigma} \quad \in \quad \mathbb{N}
$$

Fig. 10. The syntax of oLCM. oLCM extends LCM by adding stack pointers, return pointers, and a built-in stack. Everything specific to the overlay semantics is written in blue.

requirement for all components (trusted and untrusted), but the reasonability requirement only applies to trusted components, i.e. those components for which we provide LSE and WBCF guarantees.

4.1 Overlay semantics

The overlay semantics oLCM for LCM views part of the memory as a built-in stack (Figure 10). To this end, it adds a call stack and a free stack memory to the executable configurations of LCM. The call stack is a list with all the stack frames that are currently inaccessible because they belong to previous calls. Every stack frame contains encapsulated stack memory as well as the program point that execution is supposed to return to. The free stack memory is the active part of the stack that has not been encapsulated by a previous call and thus can be used by the currently executing code. In order to distinguish capabilities for the stack from the capabilities for the rest of the memory, oLCM adds stack pointers. A stack pointer has a permission, range of authority, and current address, just like capabilities on LCM, but they are always linear. The final syntactic constructs added by oLCM are code and data return pointers. The data return pointer corresponds to a stack capability, and the code return pointer corresponds to a executable capability for the corresponding return point. Syntactically, the return pointers contain just enough information to reconstruct what they correspond to on the underlying machine. During oLCM function calls, return pointers are generated from the stack and program counter capabilities, and they are turned back to those capabilities upon return.

The opaque nature of the return pointers is reflected in the interpretation of instructions as oLCM does not add special interpretation for them in non-`xjmp` instructions. Stack pointers, on the other hand, need to behave just like capabilities, so oLCM adds new cases for them in the semantics, e.g. `cca` can now also change the current address of a stack pointer as displayed in Figure 11. Similarly, `load` and `store` work on the free part of the stack when provided with a stack pointer. A store attempted with a stack capability that points to an address outside the free stack results in the failed configuration because that action is inconsistent with the view of the overlay semantics on the underlying machine. In oLCM executions, there should only be stack pointers for the stack memory.

As discussed earlier, our formal results only provide guarantees for components that respect the calling convention. Untrusted components are not assumed to do so. To formalize this distinction, oLCM has a set of trusted addresses T_A. This T_A is a constant parameter of the oLCM step relation. Only instructions at these addresses will be

$$\frac{\begin{array}{c}\Phi(\mathrm{pc}) = ((p, \_), b, e, a) \\ [a, a+\mathrm{call\_len}-1] \subseteq T_A \qquad [a, a+\mathrm{call\_len}-1] \subseteq [b, e] \qquad p \in \{\mathrm{RWX}, \mathrm{RX}\} \\ \Phi.mem(a, \ldots, a+\mathrm{call\_len}-1) = \mathtt{call}_0^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_\sigma}\; r_1\; r_2 \cdots \mathtt{call}_{\mathrm{call\_len}-1}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_\sigma}\; r_1\; r_2\end{array}}{\Phi \to^{T_A, \mathrm{stk\_base}} \left[\!\left[\mathtt{call}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_\sigma}\; r_1\; r_2\right]\!\right](\Phi)}$$

$i \in \mathrm{Instr}$	$[\![i]\!]\ (\Phi)$	Conditions
`halt`	halted	
… (the operational semantics of LCM)		
`store` $r_1\ r_2$	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[ms_{stk}.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = \text{stack-ptr}(p, b, e, a)$ and $p \in \{\mathrm{RWX}, \mathrm{RW}\}$ and $b \le a \le e$ and $w_2 = linClear(\Phi(r_2))$ and $a \in \mathrm{dom}(ms_{stk})$
`cca` $r\ rn$	$updPc(\Phi[reg.r \mapsto w])$	$\Phi(rn) = n \in \mathbb{Z}$ and $\Phi(r) = \text{stack-ptr}(p, b, e, a)$ and $w = \text{stack-ptr}(p, b, e, a+n)$
$\mathtt{call}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_\sigma}$ $r_1\ r_2$	$xjmpRes(c_1, c_2, \left(\begin{array}{l}\Phi[reg.r_1 \mapsto w_1] \\ \;[reg.r_2 \mapsto w_2] \\ \;[reg.\mathrm{r_{rcode}} \mapsto s_c] \\ \;[reg.\mathrm{r_{rdata}} \mapsto s_d] \\ \;[reg.\mathrm{r_{stk}} \mapsto c_{stk}] \\ \;[ms_{stk} \mapsto ms_{stk,rest}] \\ \;[stk \mapsto stk']\end{array}\right))$	$ms_{stk,local}, c_{local}, ms_{stk,rest}, c_{stk} =$ $splitStack(\Phi.reg(\mathrm{r_{stk}}), \Phi.ms_{stk})$ and $opc, c_{opc} = setupOpc(\Phi.reg(\mathrm{pc}))$ and $stk' = (opc, ms_{stk,local}) :: \Phi.stk$ and $\sigma = getCallSeal($ $\Phi.reg(\mathrm{pc}), \Phi.mem, \mathit{off}_{\mathrm{pc}}, \mathit{off}_\sigma)$ and $s_c, s_d = sealReturnPair(\sigma, c_{opc}, c_{local})$ and $w_1, w_2 = linClear(\Phi.reg(r_1, r_2))$ and $\Phi.reg(r_1, r_2) =$ $\mathrm{sealed}(\sigma', c_1), \mathrm{sealed}(\sigma', c_2)$
…		
_	failed	otherwise

$$xjmpRes(c_1, c_2, \Phi) = \begin{cases} \Phi[reg.\mathrm{pc} \mapsto c_1] & nonExec(c_2) \text{ and } c_1 \ne \text{ret-ptr-code}(\_) \text{ and} \\ \quad [reg.\mathrm{r_{data}} \mapsto c_2] & c_2 \ne \text{ret-ptr-data}(\_) \\ \Phi[reg.\mathrm{pc} \mapsto c_{opc}] & (opc, ms_{local}) :: stk' = \Phi.stk \wedge \\ \quad [reg.\mathrm{r_{stk}} \mapsto c_{stk}] & c_1 = \text{ret-ptr-code}(b, e, opc) \\ \quad [reg.\mathrm{r_{data}} \mapsto 0] & c_2 = \text{ret-ptr-data}(a_{stk}, e_{stk}) \wedge \mathrm{dom}(ms_{local}) = [a_{stk}, e_{stk}] \\ \quad [stk \mapsto stk'] & c_{stk} = reconstructStackPointer(\Phi.reg(\mathrm{r_{stk}}), c_2) \wedge \\ \quad [ms_{stk} \mapsto ms_{stk} \uplus ms_{local}] & c_{opc} = ((\mathrm{RX}, \text{normal}), b, e, opc) \\ \text{failed} & \text{otherwise} \end{cases}$$

Fig. 11. An excerpt of the operational semantics of oLCM (some details omitted). Auxiliary definitions are found in Figure 12.

interpreted as native oLCM calls and push frames to the call stack. As such, WBCF and LSE will be guaranteed only for calls from these addresses. In addition to T_A, StkTokens assumes a fixed base address of the stack memory, that is also passed around as such a parameter, for use in the native semantics of calls.

In addition to the (modified) reduction rules of LCM, OLCM has an overlay step that takes precedence over the others. This step is shown in Figure 11, and it is different from the others in the sense that it interprets a sequence of instructions rather than a single one. The sequence of instructions have to correspond to a call, i.e. the instructions in Figure 9 ($\texttt{call}_i^{\mathit{off}_{\mathrm{pc}},\mathit{off}_\sigma}\ r_1\ r_2$ corresponds to the ith instruction in the figure and call_len is always 26, i.e. the number of instructions). Calls are only executed when the well-behaved component executes, so the addresses where the call resides must be in T_A, and the executing capability must have the authority to execute the call.

The interpretation of $\texttt{call}^{\mathit{off}_{\mathrm{pc}},\mathit{off}_\sigma}\ r_1\ r_2$ is also shown in Figure 11 and essentially does the following: The registers r_1 and r_2 are expected to contain a code-data pair sealed with the same seal and the unsealed values are invoked by placing them in the pc and $\mathrm{r}_{\mathrm{data}}$ registers, respectively. The current active stack and the stack capability are split into the local stack frame of the caller and the rest. `call` also constructs a return capability c_{opc} and its address opc, pointing after the call instructions. The local stack frame and return address are pushed onto the stack, and the local stack capability and return capability are converted into a pair of return capabilities and sealed with the seal designated for the call.

Since the return capabilities ret-ptr-code and ret-ptr-data are sealed, they can only be used using the `xjmp` instruction, to perform a return. When this happens, the topmost call stack frame (opc, ms_{local}) is popped from the call stack. In order for the return to succeed, the return address in the code return pointer must match opc, and the range of addresses in the data return pointer must match the domain of the local stack. If the return succeeds, the stack pointer is reconstructed, and the local stack becomes part of the active stack again.

OLCM supports tail calls. A tail call is a call from a caller that is done executing, and thus does not need to be returned to or preserve local state. This means that a tail call should not reserve a slot in the return order by pushing a stack frame on the call stack, i.e. it should not use the built-in OLCM call. Instead, to perform a tail call, the caller simply transfers control to the callee using `xjmp`. The tail-callee should return to the caller's caller, so the caller leaves the return pair they received for the callee to use.

It is important to observe that the operational semantics of OLCM natively guarantee WBCF and (local stack encapsulation) for calls made by trusted components. By inspecting the operational semantics of OLCM, we can see that it never allow reads or writes to inactive stack frames on the call stack. The built-in call for trusted code pushes the local stack frame to the inactive part of the stack, together with the return address. Such frames can be reactivated by `xjmp`ing to a return capability pair, but only for the topmost stack frame and if the return address corresponds to the one stored in the call stack. In other words, WBCF and LSE are natively enforced in this semantics.

4.2 Well-formed components

The components introduced in Section 2.3 are pretty much unconstrained. For instance, a component can have multiple linear capabilities for the same piece of memory and might have arbitrary set-of-seals capabilities. In a real system, the operating system and linker would make sure everything is set up correctly. For instance, they would not allocate multiple linear capabilities for the same memory, and they would ensure sane seal allocation.

$$\mathit{splitStack}(\text{stack-ptr}(\text{RW}, b_{stk}, e_{stk}, a_{stk}), ms_{stk}) = ms_{stk,local}, c_{local\_data}, ms_{stk,unused}, c_{stk} \textit{ iff } \begin{cases} b_{stk} < a_{stk} \leq e_{stk} \\ ms_{stk,local} = ms_{stk}|_{[a_{stk}, e_{stk}]}[a_{stk} \mapsto 42] \\ ms_{stk,unused} = ms_{stk}|_{[b_{stk}, a_{stk}-1]} \\ c_{stk} = \text{stack-ptr}(\text{RW}, b_{stk}, a_{stk}-1, a_{stk}-1) \\ c_{local\_data} = \text{ret-ptr-data}(a_{stk}, e_{stk}) \end{cases}$$

$$\mathit{setupOpc}(((\_,\_), b, e, a)) = opc, c_{opc} \textit{ iff } \begin{cases} opc = a + \text{call\_len} \wedge \\ c_{opc} = \text{ret-ptr-code}(b, e, opc) \wedge \end{cases}$$

$$\mathit{getCallSeal}(c_{\text{pc}}, mem, \mathit{off}_{\text{pc}}, \mathit{off}_{\sigma}) = \sigma \textit{ iff } \begin{cases} c_{\text{pc}} = ((\_,\_), b, e, a) \wedge b \leq a + \mathit{off}_{\text{pc}} \leq e \wedge \\ mem(a + \mathit{off}_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_a) \wedge \sigma_b \leq \sigma \leq \sigma_e \wedge \\ \sigma = \sigma_a + \mathit{off}_{\sigma} \end{cases}$$

$$\mathit{sealReturnPair}(\sigma, c_{opc}, c_{local}) = \text{sealed}(\sigma, c_{opc}), \text{sealed}(\sigma, c_{local})$$

$$\mathit{reconstructStackPointer}(\text{stack-ptr}(\text{RW}, \text{stk\_base}, a_{stk}-1, \_), \text{ret-ptr-data}(a_{stk}, e_{stk})) = \\ \text{stack-ptr}(\text{RW}, \text{stk\_base}, e_{stk}, a_{stk}) \textit{ iff } \text{stk\_base} \leq \text{a}_{stk}$$

Fig. 12. Auxiliary definitions used in the operational semantics of oLCM.

In this section, we introduce a syntactic well-formedness judgment, which captures requirements that would normally be enforced by the operating system and linker. It applies to both trusted and adversarial components. Well formedness will only be required when the system starts executing, i.e. the requirement serves to ensure a sane initial state. It does not prevent code from organizing its memory differently when the system is executing.

For simplicity, the well-formedness judgment imposes a quite rigid structure on components. A component's code memory may contain only data (including encoded instructions) and set-of-seals capabilities. The data memory may contain only data, memory capabilities to the component's data memory (respecting linearity) or sealed memory capabilities to the component's data memory, sealed with closure seals. This produces a clear separation between code and data memory as neither can contain capabilities for the other. Well formedness does not allow executable capabilities to data memory or writable capabilities to code memory, effectively enforcing a kind of Write-XOR-Execute policy. This means we currently exclude dynamic code generation, but see Section 6.3 for thoughts on how this restriction might be lifted. A component's exports should be sealed code or data capabilities, sealed with a closure seal.

As explained in the previous section, we make use of a set of trusted addresses T_A to distinguish trusted components from adversarial components. To prevent ambiguity, the well-formedness judgment requires that any component's code memory must fall entirely within or outside of T_A, so every component is either trusted or adversarial. Additionally, some well-formedness requirements are specific for adversarial and trusted components respectively. Well-formed adversarial components cannot have return seals. On the other hand, well-formed trusted components can have return seals and must allocate unique return seals to every call site.

$$\dfrac{\begin{array}{c}\mathrm{dom}(ms_{\mathrm{code}}) = [b, e] \qquad [b-1, e+1] \mathrel{\#} \mathrm{dom}(ms_{\mathrm{data}}) \\ ms_{\mathrm{pad}} = [b-1 \mapsto 0] \uplus [e+1 \mapsto 0] \qquad \exists A_{\mathrm{own}} : \mathrm{dom}(ms_{\mathrm{data}}) \to \mathscr{P}(\mathrm{dom}(ms_{\mathrm{data}})) \\ \mathrm{dom}(ms_{\mathrm{data}}) = A_{\mathrm{non-linear}} \uplus A_{\mathrm{linear}} \qquad A_{\mathrm{linear}} = \biguplus_{a \in \mathrm{dom}(ms_{\mathrm{data}})} A_{\mathrm{own}}(a) \\ \overline{export} = \overline{s_{\mathrm{export}} \mapsto w_{\mathrm{export}}} \qquad \overline{import} = \overline{a_{\mathrm{import}} \hookleftarrow s_{\mathrm{import}}} \qquad \{\overline{a_{\mathrm{import}}}\} \subseteq \mathrm{dom}(ms_{\mathrm{data}}) \\ \overline{s_{\mathrm{import}}} \mathrel{\#} \overline{s_{\mathrm{export}}} \qquad (\emptyset \neq \mathrm{dom}(ms_{\mathrm{code}}) \subseteq T_A) \vee (\mathrm{dom}(ms_{\mathrm{code}}) \mathrel{\#} T_A \wedge \overline{\sigma_{\mathrm{ret}}} = \emptyset) \\ \mathrm{dom}(ms_{\mathrm{data}}) \mathrel{\#} T_A \qquad \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}} \\ \forall a \in \mathrm{dom}(ms_{\mathrm{data}}).\ \mathrm{dom}(ms_{\mathrm{code}}), A_{\mathrm{own}}(a), A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} ms_{\mathrm{data}}(a) \\ \forall w_{\mathrm{export}} \in \overline{w_{\mathrm{export}}}.\ \mathrm{dom}(ms_{\mathrm{code}}), A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-export}} w_{\mathrm{export}} \end{array}}{T_A \vdash (ms_{\mathrm{code}} \uplus ms_{\mathrm{pad}}, ms_{\mathrm{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, A_{\mathrm{linear}})} \ \textsc{Base}$$

$$\dfrac{\begin{array}{c} comp_0 = (ms_{\mathrm{code}}, ms_{\mathrm{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, A_{\mathrm{linear}}) \\ T_A \vdash comp_0 \qquad (\_ \mapsto c_{\mathrm{main},c}), (\_ \mapsto c_{\mathrm{main},d}) \in \overline{export} \end{array}}{T_A \vdash (comp_0, c_{\mathrm{main},c}, c_{\mathrm{main},d})} \ \textsc{Main}$$

Fig. 13. Well-formedness judgment.

These requirements are formally expressed by the well-formedness judgment $T_A \vdash comp$: it defines initial syntactic requirements for components, necessary to be able to rely on unique linear capabilities, component unique seals, etc. The judgment is defined in Figure 13 with auxiliary judgments in Figure 14.

The $T_A \vdash comp$ judgment has two rules: MAIN and BASE. If components contain a main entry point (in the form of capabilities $c_{\mathrm{main},c}$, $c_{\mathrm{main},d}$), then the MAIN rule requires them to be part of the exports. The BASE rule for base components has a variety of requirements:

- *Code and data memory are disjoint.*
- *Code memory is padded with zeros (ms_{pad}), and this padded memory may not be referenced.* This prevents code memories from being spliced together. If the capabilities for two code memories can be spliced together, then the execution of one code memory can continue into the other creating an unintended control flow which would cause various complications.[5]
- *All memory can either be addressed by any number of nonlinear capabilities or at most one linear capability.* The data address space is split into $A_{non-linear}$ and A_{linear} which can be addressed by nonlinear capabilities and linear capabilities, respectively. The judgment ensures uniqueness of linear capabilities by allocating ownership of addresses in A_{linear} uniquely to linear capabilities in data memory.
- *All import addresses are part of the data memory.*
- *Import and export symbols are disjoint.*
- *One of the following is true:*
  - *The code address space is disjoint from the trusted address space T_A and there are no return seals.* In this case, the component contains untrusted code. We are interested in WBCF and LSE from the perspective of the trusted code, so we do

[5] Alternatively to this syntactic condition on components, we could require trusted components to never splice executable capabilities of unknown origin.

$$\frac{ms_{\mathrm{code}}(a) = \mathrm{seal}(\sigma_b, \sigma_e, \sigma_b) \qquad [\sigma_b, \sigma_e] = (\overline{\sigma_{\mathrm{ret}}} \cup \overline{\sigma_{\mathrm{clos}}})}{\overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{ret,owned}}}, \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}}, a} \ \textsc{C-Seals}$$

$$\frac{\begin{array}{ll} ([a \cdots a + \mathrm{call\_len} - 1] \subseteq T_A \wedge & \\ \quad ms_{\mathrm{code}}([a \cdots a + \mathrm{call\_len} - 1]) = \mathtt{call}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_{\sigma}}_{0..\mathrm{call\_len}-1}\ r_1\ r_2) \Rightarrow & ms_{\mathrm{code}}(a) \in \mathbb{Z} \\ (ms_{\mathrm{code}}(a + \mathit{off}_{\mathrm{pc}}) = \mathrm{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma_b + \mathit{off}_{\sigma} \in \overline{\sigma_{\mathrm{ret,owned}}}) & \end{array}}{\overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{ret,owned}}}, \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}}, a} \ \textsc{C-Instr}$$

$$\frac{\begin{array}{c} ms_{\mathrm{code}} \text{ has no hidden calls} \\ \overline{\sigma_{\mathrm{ret}}} \ \# \ \overline{\sigma_{\mathrm{clos}}} \qquad \exists d_{\sigma} : \mathrm{dom}(ms_{\mathrm{code}}) \to \mathscr{P}(\mathrm{Seal}).\ \overline{\sigma_{\mathrm{ret}}} = \biguplus_{a \in \mathrm{dom}(ms_{\mathrm{code}})} d_{\sigma}(a) \wedge \\ \forall a \in \mathrm{dom}(ms_{\mathrm{code}}).\ \overline{\sigma_{\mathrm{ret}}}, d_{\sigma}(a), \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}}, a \end{array}}{\overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}}} \ \textsc{C-Mem}$$

$$\frac{z \in \mathbb{Z}}{A_{\mathrm{code}}, A_{\mathrm{own}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} z} \ \textsc{W-Data}$$

$$\frac{\begin{array}{c} \mathrm{perm} \sqsubseteq \mathrm{RW} \qquad l = \mathrm{linear} \Rightarrow \emptyset \subset [b, e] \subseteq A_{\mathrm{own}} \\ l = \mathrm{normal} \Rightarrow [b, e] \subseteq A_{\mathrm{non-linear}} \end{array}}{A_{\mathrm{code}}, A_{\mathrm{own}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} ((p, l), b, e, a)} \ \textsc{W-Capability}$$

$$\frac{A_{\mathrm{code}}, A_{\mathrm{own}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} sc \qquad \sigma \in \overline{\sigma_{\mathrm{clos}}}}{A_{\mathrm{code}}, A_{\mathrm{own}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} \mathrm{sealed}(\sigma, sc)} \ \textsc{W-Sealed-Capability}$$

$$\frac{[b, e] \subseteq A_{\mathrm{code}} \qquad \sigma \in \overline{\sigma_{\mathrm{clos}}}}{A_{\mathrm{code}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-export}} s \mapsto \mathrm{sealed}(\sigma, ((\mathrm{RX}, \mathrm{normal}), b, e, a))} \ \textsc{E-Sealed-Code}$$

$$\frac{A_{\mathrm{code}}, \emptyset, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} w}{A_{\mathrm{code}}, A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-export}} s \mapsto w} \ \textsc{E-Word}$$

Fig. 14. Wellformedness judgement for code, words, and export. call_len is the length of the call code. (a) Code wellformedness. (b) Word wellformedness. (c) Export wellformedness.

not let untrusted memory have return seals. Untrusted code still has access to closure seals that it can use to protect calls.

– *The code address space is part of the trusted address space T_A.* In this case, the component contains trusted code and can use return seals for calls.

- *The data address space is disjoint from the trusted address space T_A.* Data memory is not executable, so the trusted addresses never include data memory addresses.
- *The code memory, data memory, and exports satisfy, respectively, the component-code, component-word, and components-export well-formedness judgments*, which we explain next.

Figure 14(b) defines the well-formedness judgment $\vdash_{\mathrm{comp-word}}$ for words in a component's data memory. Well-formed rules are defined as integers, memory capabilities with at most read and write permission and well-formed words sealed with a closure seal. The range of authority of linear and nonlinear memory capabilities is required to be available in A_{linear} (where it is exclusively owned) and $A_{\text{non-linear}}$, respectively. Note that data memory is not allowed to initially contain sealed code capabilities. However, a component is free to place sealed data capabilities in memory during execution.

Figure 14(c) defines the well-formedness judgment $\vdash_{\text{comp-export}}$ for component exports. Essentially, well-formed exports are either well-formed words or sealed read-execute capabilities to the component's code memory, sealed with a closure seal. Both cases together allow components to export closures as sealed code-data capability pairs.

Finally, Figure 14(a) defines the well-formedness judgment $\vdash_{\text{comp-code}}$ for components' code memory. The judgment $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}$ is defined in terms of an auxiliary judgment for an individual memory address $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{ret,owned}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}, a$. The rules allow either set-of-seals capabilities for closure and return seals or integers, including encoded instructions. If the code memory contains a call instruction in a trusted component (i.e. addresses in T_A), then there must be a set-of-seals capability available at the expected address, containing the return seal for this call. Additionally, the judgment ensures that return seals are at most used by one call by partitioning available return seals in $\overline{\sigma_{\text{ret,owned}}}$ over code addresses for use in calls. Further, return seals must be disjoint from closure seals.

4.3 Reasonable components

The static guarantees given by $T_A \vdash comp$ make sure that components initially do not undermine the security measures needed for StkTokens. However, in order for StkTokens to provide guarantees for a component, we additionally expect it not to shoot itself in the foot at runtime and perform certain necessary checks not captured by the call code (Figure 9). More precisely, we expect four things of a reasonable component:

1. It checks the stack base address before performing a call. As explained in Section 3, we do not include this check in the call code as it would often be redundant.
2. It uses the return seals only for calls and the closure seals in an appropriate way. Specifically, closure seals should only be used to seal executable capabilities for code that behaves reasonably, or for certain forms of nonexecutable capabilities.
3. It does not leak return and closure seals (or means to retrieve them). This means that set-of-seals capabilities with return or closure seals cannot be left in registers when transferring control to another module. There are also indirect ways to leak seals such as leaking a capability for code memory or leaking a capability for code memory sealed with an unknown seal.
4. It never stores return and closure seals (or means to get them) to memory. By disallowing this, we make sure that data memory always can be safely shared as it does not contain seals or means to get them.

We capture these properties in 4 definitions. Definition 2 defines the reasonable words, which means that they cannot be used to leak seals directly or indirectly. Definition 3 defines the reasonable PCs, which means that if it is plugged into a configuration with a register file filled with reasonable words, then the configuration behaves reasonably. Definition 4 defines the reasonable configurations which captures the four informal behavioral properties. Finally, Definition 5 lifts the notion of reasonability to components.

Most of the definitions in this chapter will be parameterized by a tuple $(T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}})$ that we will sometimes denote with the meta-variable gc. This tuple collects four *global constants*. In addition to T_A, the set of trusted addresses, and stk_base, the fixed base address of the stack memory, which we have already encountered in Section 4.1, the tuple contains two additional constants: $\overline{\sigma_{\text{glob\_ret}}}$ and $\overline{\sigma_{\text{glob\_clos}}}$:, respectively, the set of all return and closure seals in the system.

4.3.1 Reasonable words

To provide any guarantees, STKTOKENS relies on the program not to leak return seals in any way. But what does it mean to "leak" a return seal? It means that set-of-seals capabilities that contain return seals as well as any means to obtain such sets cannot be leaked. The following definition makes this more precise.

Definition 2 (Reasonable word)**.** *Take a set of trusted addresses T_A and sets of return and closure seals $\overline{\sigma_{\text{glob\_ret}}}$ and $\overline{\sigma_{\text{glob\_clos}}}$. We define that a word w is reasonable up to n steps in memory ms and free stack ms_{stk} if $n = 0$ or the following implications hold.*

- *If $w = \text{seal}(\sigma_b, \sigma_e, \_)$, then $[\sigma_b, \sigma_e] \# (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}})$*
- *If $w = ((p, \_), b, e, \_)$, then $[b, e] \# T_A$*
- *If $w = \text{sealed}(\sigma, sc)$ and $\sigma \notin (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}})$ then sc is reasonable up to $n - 1$ steps.*
- *If $w = ((p, \_), b, e, \_)$ and $p \in readAllowed$ and $n > 0$, then $ms(a)$ is reasonable up to $n - 1$ steps for all $a \in ([b, e] \setminus T_A)$*
- *If $w = \text{stack-ptr}(p, b, e, \_)$ and $p \in readAllowed$ and $n > 0$, then $ms_{stk}(a)$ is reasonable up to $n - 1$ steps for all $a \in [b, e]$*

The definition rules out set-of-seals capabilities that contain return or closure seals. STKTOKENS rely on return seals to be unique in order to work, but it does not rely on closure seals. However, leaking closure seals to malicious code still defeats their purpose as the malicious code can then fabricate data capabilities for the code capabilities or code capabilities for the data capabilities which effectively allows malicious code to unseal the data capability. Further, it makes reasoning about leakage of return seals difficult because the closures have to be well behaved no matter what data they are executed with. Capabilities to trusted code memory in T_A are also ruled out, because the code memory may contain set-of-seals capabilities. Capabilities that are sealed with capabilities not owned by the component must still be reasonable themselves (since such an untrusted seal provides no protection at all).

Finally, we define that stack or memory capabilities with read permission are reasonable if the memory they give access to contains only reasonable words. To accommodate this, reasonability of words is defined relative to the memory and stack. The definition is cyclic, but the step index ensures that it is well founded.

4.3.2 Reasonable pc and configuration

In order to define desired behavior, we need to specify what may happen on a machine during execution. An execution steps between executable configurations, so we need to define when a configuration is reasonable. While the step relation is defined over configurations, it is the pc that decides what instruction is executed. Therefore, we define when an executable capability is reasonable as a pc. The following definition roughly says that a pc is reasonable when you can plug it into a configuration where all the words in the register file are reasonable and the result is a reasonable configuration.

Definition 3 (Reasonable pc)**.** *We say that an executable capability $c = ((p, \mathrm{normal}), b, e, a)$ behaves reasonably up to n steps if for any Φ such that*

- $\Phi.reg(\mathrm{pc}) = c$
- *$\Phi.reg(r)$ is reasonable up to n steps in memory $\Phi.mem$ and free stack $\Phi.ms_{stk}$ for all $r \neq \mathrm{pc}$*
- *$\Phi.mem$, $\Phi.ms_{stk}$ and $\Phi.stk$ are all disjoint*

we have that Φ is reasonable up to n steps.

With Definitions 2 and 3 in place, we are all set to define when a configuration is reasonable.

Definition 4 (Reasonable configuration)**.** *We say that an execution configuration Φ is reasonable up to n steps with $(T_A, \mathrm{stk\_base}, \overline{\sigma_{\mathrm{glob\_ret}}}, \overline{\sigma_{\mathrm{glob\_clos}}})$ iff for $n' \leq n$:*

1. *Guarantee stack base address before call...*
   If Φ points to $\mathtt{call}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_{\sigma}}\ r_1\ r_2$ in T_A for any r_1 and r_2, then all of the following hold:
   - $\Phi(r_{stk}) = \text{stack-ptr}(\_, \mathrm{stk\_base}, \_, \_)$
   - $r_1 \neq \mathrm{r_{t1}}$
   - *$n' = 0$ or $\Phi(\mathrm{pc}) + \mathrm{call\_len}$ behaves reasonably up to $n' - 1$ steps (Definition 3)*
2. *Use return seals only for calls, use closure seals appropriately...*
   If Φ points to $\mathtt{cseal}\, r_1\, r_2$ in T_A and $\Phi(r_2) = \mathrm{seal}(\sigma_b, \sigma_e, \sigma)$, then one of the following holds:
   - *Φ is inside $\mathtt{call}^{\mathit{off}_{\mathrm{pc}}, \mathit{off}_{\sigma}}\ r'_1\ r'_2$ and $\sigma \in \overline{\sigma_{\mathrm{glob\_ret}}}$*
   - *$\sigma \in \overline{\sigma_{\mathrm{glob\_clos}}}$ and one of the following holds:*
     - *executable($\Phi(r_1)$) and $n' = 0$ or $\Phi(r_1)$ behaves reasonably up to $n' - 1$ steps (Definition 3).*
     - *nonExec($\Phi(r_1)$) and $n' = 0$ or $\Phi(r_1)$ is reasonable up to $n' - 1$ steps in memory $\Phi.ms$ and free stack $\Phi.ms_{stk}$ (Definition 2).*
3. *Don't store private stuff...*
   If Φ points to $\mathtt{store}\, r_1\, r_2$ in T_A, then $n' = 0$ or $\Phi.reg(r_2)$ is reasonable in memory $\Phi.mem$ up to $n' - 1$ steps.

4. *Don't leak private stuff...*
   If $\Phi \rightarrow^{T_A,\text{stk\_base}} \Phi'$*, then one of the following holds:*
   - *There exist p, l, b, e, a, a′ such that all of the following hold:*
     - $\Phi'.reg(\text{pc}) = ((p,l),b,e,a')$ *and* $\Phi.reg(\text{pc}) = ((p,l),b,e,a)$
     - Φ *does not point to* $\texttt{xjmp}\ r_1\ r_2$ *for any* r_1 *and* r_2
     - Φ *does not point to* $\texttt{call}^{\mathit{off}_{\text{pc}},\mathit{off}_{\sigma}}\ r_1\ r_2$ *for any* r_1 *and* r_2, off_{pc}, off_{σ}
     - $n' = 0$ *or* Φ' *is reasonable up to* $n' - 1$ *steps*
   - *All of the following hold:*
     - Φ *points to* $\texttt{call}^{\mathit{off}_{\text{pc}},\mathit{off}_{\sigma}}\ r_1\ r_2$ *for some* r_1 *and* r_2
     - $n' = 0$ *or* $\Phi.reg(r)$ *is reasonable in memory* $\Phi.mem$ *and free stack* $\Phi.ms_{stk}$ *up to* $n' - 1$ *steps for all* $r \neq \text{pc}$
   - *All of the following hold:*
     - Φ *points to* $\texttt{xjmp}\ r_1\ r_2$ *for some* r_1 *and* r_2
     - $n' = 0$ *or* $\Phi.reg(r)$ *is reasonable in memory* $\Phi.mem$ *and free stack* $\Phi.ms_{stk}$ *up to* $n' - 1$ *steps for all* $r \neq \text{pc}$

The four items in Definition 4 correspond to the four informal items from the introduction of this section.

Item 3 and Item 4 make sure that return seals are not leaked. Item 3 says that only reasonable words can be stored in memory. This means that sets of return seals or execute capabilities cannot be stored in memory by a reasonable component. Intuitively, a well formed and reasonable component has its seals available in the code memory, so it can always retrieve them from the code memory. In other words, it is not necessary to store them in the data memory. Further, by making sure that seals are not stored in memory, we can allow capabilities for data memory to be handed out if there is a need for that (for instance to have a shared buffer). Item 4 makes sure that seals are not leaked when transferring control to another component (i.e. on security boundary crossings). With the component setup, there are two ways to transfer control: xjmp and call. In both cases, we require that all of the argument registers contain reasonable words. An execution configuration may need to do other operations than calling other code, and seals should not be leaked at any point during execution. For this reason, Item 4 also says that if the next step is not a call or a jump, then the next execution configuration should also be reasonable.

Item 1 makes sure that the stack has the correct base before a call. In order to not have to reason about unreasonably generated code, we also add the requirement $r_1 \neq \text{r}_{\text{t1}}$ before calls. If we allowed $r_1 = \text{r}_{\text{t1}}$, then the call would be sure to fail as the first instruction of a call moves 42 to r_{t1}. Finally, this promises that the code after the call will behave reasonably.

A well-formed component makes sure that a return seal is uniquely available to every call. This is, however, not sufficient as it does not ensure that other parts of a program do not use the return seals. We do not want to specify what non-call code should look like, so we just require it not to use the call seals. This is what Item 2 ensures. It says that if the configuration points to a seal instruction, then either the instruction is part of a call and uses a return seal or the instruction seals part of a closure and uses a closure seal. In the latter case, the sealed capability must be reasonable as a pc if it is executable or just a reasonable word if it is not. Definition 4 is cyclic through Definition 3, so both definitions are step-indexed to break the cycle.

4.3.3 Reasonable component

A reasonable component has the informal behavioral properties from the introduction. Reasonability is captured by the previous definitions. These definitions are lifted to components by the following definition.

Definition 5 (Reasonable component)**.** *We say that a component*

$$(ms_{\text{code}}, ms_{\text{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}})$$

is reasonable if the following hold: For all $(s \mapsto \text{sealed}(\sigma, sc)) \in \overline{c_{\text{export}}}$*, with executable*$(sc)$*, we have that sc behaves reasonably up to any number of steps n.*

We say that a component $(comp_0, c_{\text{main},c}, c_{\text{main},d})$ *is reasonable if* $comp_0$ *is reasonable.*

In our result, we assume that adversarial components are well formed. This assumption ensures that the trusted component can rely on basic security guarantees provided by the capability machine. For instance, if we did not require linearity to be respected initially, then adversarial code could start with an alias for the stack capability. However, the adversary is not assumed to be reasonable as we do not expect them to obey the calling convention in any way. Can adversarial code call into trusted components? The answer to that question is yes but not with LSE and WBCF guarantees. Formally, adversarial code can contain the instructions that constitute a call. However, for untrusted code, OLCM will not execute those instructions as a "native call" but execute the individual instructions separately. The callee then executes in the same stack frame as the caller, so WBCF and LSE do not follow (for that call).

We will assume trusted components, for which WBCF and LSE are guaranteed, to be both well-formed and reasonable.

4.4 Full abstraction

All that is left before we state the full-abstraction theorem is to define how components are combined with contexts and executed, so that we can define contextual equivalence.

Given a program *comp*, the judgment $comp \rightsquigarrow \Phi$ in Figure 15 defines an initial execution configuration that can be executed. It works almost the same on LCM (conditions in red) and OLCM (conditions in blue). On both machines a stack containing all zeroes is added, as part of the regular memory on LCM and as the free stack on OLCM. On OLCM, the initial stack is empty as no calls have been made. The component needs access to the stack, so a stack pointer is added to the register file in r_{stk}. On LCM this is just a linear read-write capability, but on OLCM it is the representation of a stack pointer. The entry point of the program is specified by main, so the two capabilities are unsealed (they must have the same seal) and placed in the pc and r_{data} registers. Other registers are set to zero.

Contextual equivalence roughly says that two components behave the same no matter what context we plug them into.

$$\frac{\begin{array}{ccc} c_{\text{main},c} = \text{sealed}(\sigma, c'_{\text{main},c}) & c_{\text{main},d} = \text{sealed}(\sigma, c'_{\text{main},d}) & \mathit{nonExec}(c'_{\text{main},d}) \\ \mathit{reg}(\text{pc}, \text{r}_{\text{data}}) = c'_{\text{main},c}, c'_{\text{main},d} & & \mathit{reg}(\text{r}_{\text{stk}}) = \text{stack-ptr}(\text{RW}, b_{stk}, e_{stk}, e_{stk}) \\ \mathit{reg}(\text{r}_{\text{stk}}) = ((\text{RW}, \text{linear}), b_{stk}, e_{stk}, e_{stk}) & & \mathit{reg}(\text{RegName} \setminus \{\text{pc}, \text{r}_{\text{data}}, \text{r}_{\text{stk}}\}) = 0 \\ \mathit{range}(\mathit{ms}_{stk}) = \{0\} & & \mathit{mem} = \mathit{ms}_{\text{code}} \uplus \mathit{ms}_{\text{data}} \uplus \mathit{ms}_{stk} \\ [b_{stk}, e_{stk}] = \text{dom}(\mathit{ms}_{stk}) \# (\text{dom}(\mathit{ms}_{\text{code}}) \cup \text{dom}(\mathit{ms}_{\text{data}})) & & \overline{\mathit{import}} = \emptyset \end{array}}{((\mathit{ms}_{\text{code}}, \mathit{ms}_{\text{data}}, \overline{\mathit{import}}, \overline{\mathit{export}}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}}), c_{\text{main},c}, c_{\text{main},d}) \rightsquigarrow (\mathit{mem}, \mathit{reg}, \emptyset, \mathit{ms}_{stk})}$$

Fig. 15. The judgment $\mathit{prog} \rightsquigarrow \Phi$, which defines the initial execution configuration Φ for executing a program prog.

Definition 6 (Plugging a component into a context)**.** *When comp' is a context for component comp and $\mathit{comp}' \bowtie \mathit{comp} \rightsquigarrow \Phi$, then we write $\mathit{comp}'[\mathit{comp}]$ for the execution configuration Φ.*

Definition 7 (LCM and OLCM contextual equivalence)**.**

On OLCM , *we define that $\mathit{comp}_1 \approx_{\text{ctx}} \mathit{comp}_2$ iff*

$$\forall \mathscr{C}.\, \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[\mathit{comp}_1] \Downarrow_{-}^{T_{A,1},\text{stk\_base}_1} \Leftrightarrow \mathscr{C}[\mathit{comp}_2] \Downarrow_{-}^{T_{A,2},\text{stk\_base}_2}$$

with $T_{A,i} = \text{dom}(\mathit{comp}_i.\mathit{ms}_{\text{code}})$.

On LCM , *we define that $\mathit{comp}_1 \approx_{\text{ctx}} \mathit{comp}_2$ iff*

$$\forall \mathscr{C}.\, \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[\mathit{comp}_1] \Downarrow_{-} \Leftrightarrow \mathscr{C}[\mathit{comp}_2] \Downarrow_{-}$$

where $\Phi \Downarrow_i^{T_A,\text{stk\_base}}$ *iff* $\Phi \rightarrow_i^{T_A,\text{stk\_base}} \text{halted}$ *and* $\Phi \Downarrow_{-}^{T_A,\text{stk\_base}} \overset{\mathit{def}}{=} \exists i.\, \Downarrow_i^{T_A,\text{stk\_base}}$

With the above defined, we are almost ready to state our full-abstraction, and all that remains is the compiler we claim to be fully-abstract. We only care about the well-formed components, and they sport none of the new syntactic constructs OLCM adds to LCM. This means that the compilation from OLCM components to LCM components is simply the identity function.

Theorem 1. *For reasonable, well-formed components comp_1 and comp_2, we have*

$$\mathit{comp}_1 \approx_{\text{ctx}} \mathit{comp}_2 \quad \Leftrightarrow \quad \mathit{comp}_1 \approx_{\text{ctx}} \mathit{comp}_2$$

Readers unfamiliar with fully-abstract compilation may wonder why Theorem 1 proves that StkTokens guarantees LSE and WBCF. Generally speaking, behavioral equivalences are preserved and reflected by fully-abstract compilers. This means that source language abstractions (or at least the equivalences they imply[6]) must be preserved in the target language, either by translating them to similar abstractions in the target language or by using the available target language features to enforce the source language abstraction. In our case, OLCM semantics offers a native stack with LSE and WBCF, but this abstraction does not natively exist in LCM. In order to enforce the abstraction on LCM, StkTokens

[6] See Section 7 for a discussion of secure compilation properties that require preservation of more general language properties than equivalences.

can be used. Theorem 1 proves that StkTokens enforces these abstraction properties in a way that behaviorally matches oLCM which means that it enforces LSE and WBCF.

5 Proving full abstraction

To prove Theorem 1, we essentially show that trusted components in oLCM are related in a certain way to their embeddings in LCM and that untrusted LCM components are similarly related to their embeddings in oLCM. We will then prove that these relations imply that the combined programs have the same observable behavior, i.e. one terminates if and only if the other does. The difficult part is to define when components are related. In the next section, we give an overview of the relation we define, and then we sketch the full-abstraction proof in Section 5.7.

The goal of this section is not to provide full technical detail or list tedious proofs. However, we believe there are many interesting aspects about our proof that we were forced to omit from the conference version, which would be worthwhile explaining to other researchers. This includes, for example, our use of cross-language logical relations, the techniques we use for reasoning about seals and linear capabilities, or for the linking model. In the current version, we provide a more detailed explanation of the most important of these aspects, taking care to gradually introduce the different techniques we use and to not bother the reader with tedious details. Of course, this material is targeted at readers with an interest in these proof techniques and may be safely skipped by others.

5.1 Kripke worlds

The relation between oLCM and LCM components is nontrivial: essentially, we will say that components are related if invoking them with related values produces related observable behavior. However, values are often only related under certain assumptions about the rest of the system. For example, the linear data part of a return capability should only be related to the corresponding oLCM capability if no other value in the system references the same inactive stack frame and it is sealed with a seal only used for return pointers to the same code location. To accommodate such conditional relatedness, we construct the relation as a step-indexed Kripke logical relation with recursive worlds.

These assumptions are explictly represented using (Kripke) worlds. To a first approximation, a world is a semantic model of the memory. In its simplest form, it is a collection of invariants that the memory must satisfy. The invariants of a world can vary in complexity and expressiveness depending on the application. In order to relate LCM and oLCM, we model all the features of oLCM in the world which means we have to model:

- Three kinds of memory: heap, stack of local frames, and free stack
- Linearity
- Call stack
- Seals

The three kinds of memory are modeled by having three subworlds where each subworld is its own little world in a traditional sense. Linearity is modeled by adding ownership to certain parts of the world which can be claimed exclusively by capabilities. Memory satisfaction (the relation that decides whether two memories are related in the world) models the call stack by ensuring that the memory is actually shaped like a stack. Finally, the seals are modeled by seal invariants that make sure that seals are only used on permitted sealables. In the following, we present the world and go into details about how each of the four features is modeled.

5.1.1 Triple world and regions

oLCMs memory is split into three: heap, free stack, and encapsulated local stack frames. In order to model the three kinds of memory, we simply have three subworlds. That is, our world is defined as a product:

$$\mathrm{World} = \mathrm{World}_{\mathrm{heap}} \times \mathrm{World}_{\mathrm{call\_stack}} \times \mathrm{World}_{\mathrm{free\_stack}}$$

For ease of explanation, we will gradually introduce the definition of these three components, starting with simplified versions and gradually introducing different complications before obtaining the intended definition of worlds by the end of Section 5.1. That version will only be modified a bit further in Section 5.2, purely for technically justifying the recursive definition of the worlds. The final definitions can be found in Theorem 2.

The subworlds are partial maps from names RegionName (not to be confused with register names), modeled as natural numbers, to regions, which model a form of invariant. Intuitvely, a region is simply a relation over memory segments (i.e. a relation in Rel(MemorySegment × MemorySegment)). However, this is not sufficient, because the validity of memory contents must often depend on the world itself. In other words, our regions must be world indexed, i.e.

$$\mathrm{World}_{\mathrm{heap}} = \mathrm{RegionName} \rightharpoonup (\mathrm{World} \to \mathrm{Rel}(\mathrm{MemorySegment} \times \mathrm{MemorySegment}))$$

At this point, we can see that we have constructed a recursive domain equation. If we inline $\mathrm{World}_{\mathrm{heap}}$ in World, then we have a circular equation with no solution because the self-reference happens in a negative position. However, in Section 5.2, we will explain how we can solve the circular equation by moving to a different domain.

For the sake of readability, we introduce the following notation

$$W.\mathrm{heap} = \pi_1(W)$$
$$W.\mathrm{call\_stk} = \pi_2(W)$$
$$W.\mathrm{free\_stk} = \pi_3(W)$$

5.1.2 Linearity

The linear capabilities of oLCM and LCM guarantee sole authority over the memory they reference. To model this uniqueness, we need to keep track of which parts of memory are uniquely referenced and make sure that only one linear capability references them. We use the world to do this by having two kinds of regions: shared and spatial. Memory governed by shared and spatial regions can only be referenced by nonlinear and linear capabilities, respectively.

However, spatial regions are not enough. A world that contains a spatial region represents the assumption of exclusive ownership of the corresponding memory. In other words, a linear capability for this memory will be valid only in such a world. However, that does not mean all other worlds should be entirely unaware of the memory's existence. To represent knowledge about existence of a spatial region, without requiring its exclusive ownership, we have shadow regions. Specifically, we add tags spatial and shadow to the spatial regions:

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{spatial}\} \times (\text{World} \to \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \cup \\ \{\text{shadow}\} \times (\text{World} \to \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \end{cases}$$

For readability, we also add a tag shared to the shared regions:

$$\text{Region}_{\text{shared}} = \{\text{shared}\} \times (\text{World} \to \text{Rel}(\text{MemorySegment} \times \text{MemorySegment}))$$

A shadow region is a shadow copy of a spatial region in the sense that it specifies part of memory, but it does not give the right to reference that part of memory. We will see in Section 5.1.5 that a world with a shadow region is compatible with a world that has the same region as spatial. This setup will allow ownership of a memory fragment to reside with linear capabilities which are not themselves stored in the memory fragment but elsewhere in memory or in a register. In such a set-up, the region representing the memory fragment will exist only as shadow in the world where the memory fragment's contents are valid, but it will exist as spatial in the world where the linear capability owning the fragment is valid.

We will extend the regions further in Sections 5.1.3 and 5.1.4, but for now we continue the definitions of the three subworlds. The subworld $\text{World}_{\text{heap}}$ specifies the heap memory which can be referenced by both linear and normal capabilities, so it should contain both shared and spatial regions. For this reason, it is defined as

$$\text{World}_{\text{heap}} = \text{RegionName} \rightharpoonup (\text{Region}_{\text{shared}} \cup \text{Region}_{\text{spatial}})$$

OLCM internalizes the STKTOKENS stack, which will only be referenced by linear capabilities, so the two stack regions only need spatial and shadow regions. For instance, the $\text{World}_{\text{free\_stack}}$ is defined as

$$\text{World}_{\text{free\_stack}} = \text{RegionName} \rightharpoonup \text{Region}_{\text{spatial}}$$

$\text{World}_{\text{call\_stack}}$ not only models the memory contents of the local stack frames but it also models the call stack itself and specifically the code location for the return point of every frame. In a traditional C calling convention, this return location would simply be stored in the stack frame, but STKTOKENS does not require this. It is the code part of the return capability pair that references this return location and it is connected to the stack frame by the fact that it is sealed with the same seal as the linear capability that owns the frame: the data part of the return capability pair. To ensure and remember that all of this is setup correctly, the world will also keep track of the return location for every frame:

$$\text{World}_{\text{call\_stack}} = \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \times \text{Addr})$$

It will in fact also remember the seal used, as we will see in the next Section.

Given a world, we want to be able to express that a capability that is otherwise valid with respect to the world is not linear or indirectly depends on a linear capability. This is expressed by stripping the world of all its ownership which corresponds to replacing all spatial regions with shadow regions and then requiring that the capability is valid w.r.t. this new world. We refer to this as the shared part of the world and define the function *sharedPart* which turns all spatial regions into shadow copies.

Definition 8 (The shared part of a world). *For any world W, we define*

$$\begin{aligned} sharedPart(W) &\overset{def}{=} (sharedPart(W.\text{heap}), sharedPart(W.\text{call\_stk}), \\ &\qquad sharedPart(W.\text{free\_stk})) \end{aligned}$$

$$sharedPart(W_{heap}) \overset{def}{=} \lambda r. \begin{cases} (\text{shadow}, H) & \textit{if } W_{heap}(r) = (\text{spatial}, H) \\ W_{heap}(r) & \textit{otherwise} \end{cases}$$

$$sharedPart(W_{call\_stk}) \overset{def}{=} \lambda r. \begin{cases} ((\text{shadow}, H), opc) & \textit{if } W_{call\_stk}(r) = ((\text{spatial}, H), opc) \\ W_{call\_stk}(r) & \textit{otherwise} \end{cases}$$

$$sharedPart(W_{free}) \overset{def}{=} \lambda r. \begin{cases} (\text{shadow}, Hs) & \textit{if } W_{free}(r) = (\text{spatial}, Hs) \\ W_{free}(r) & \textit{otherwise} \end{cases}$$

5.1.3 Seals

So far, the world represents a collection of assumptions on the memory contents that value relatedness may depend on. However, value correctness may also depend on other assumptions. Specifically, StkTokens has certain assumption on the seals used for return capabilities and closures. For instance, a return seal must only be used to seal the return pointer of one specific return point. Therefore, in addition to a relation on memory segments, some regions also carry a *seal interpretation function* that relates the sealables that may be sealed with a given seal.

$$\text{Seal} \rightharpoonup \text{World} \to \text{Rel}(\text{Sealables} \times \text{Sealables})$$

In StkTokens, once a seal has been used for a specific purpose (e.g. for sealing return capability pairs for a specific call site), it can never be reused for a different purpose. This is because there may still be copies of return capabilities out there, signed with the seal. This situation is similar to the situation for nonlinear memory capabilities, so we only allow shared regions to carry seal interpretation functions, as we will see that those regions can never be revoked in future worlds.

$$\begin{aligned} \text{Region}_{\text{shared}} = \{\text{shared}\} \times (\text{World} \to \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \times \\ (\text{Seal} \rightharpoonup \text{World} \to \text{Rel}(\text{Sealables} \times \text{Sealables})) \end{aligned}$$

We also refer to the seal interpretation function as the seal invariant, and we will refer to the memory relation as the memory invariant or just invariant when it is unambiguous.

5.1.4 Future worlds and revocation

Very often, relatedness of two capabilities does not change if extra assumptions in the system are added. For example, two related capabilities remain related when an extra invariant is added on unrelated memory, or when a stack frame that it does not reference is dropped. In Kripke logical relations, such changes that do not invalidate the relatedness of values are modeled by the future world relation $\sqsupseteq$. The future world relation can also be thought of as the model of allowed changes in memory over time. We will say that W' is a future world of W if $W' \sqsupseteq W$.

Relatedness of capabilities should be defined, so that it is monotone with respect to the future world relation. In other words, relatedness of capabilities should only be predicated on assumptions in the world that are guaranteed to remain valid in future worlds and the same holds for memory invariants. As such, we require the world-indexed memory invariants to be monotone in the world:

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}\} \times (\text{World} \overset{mon}{\longrightarrow} \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \cup \\ \{\text{spatial}\} \times (\text{World} \overset{mon}{\longrightarrow} \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \end{cases}$$

We make a similar change to $\text{Region}_{\text{shared}}$. The seal invariants must be monotone as well.

Kripke future world relations usually allow extending worlds with extra assumptions, or take steps in protocols that the system was designed to follow. However, in our setting, we sometimes allow dropping assumptions, namely when linearity tells us that no value in the system depends on this assumption any more. Specifically, if we have the only linear capability for a piece of memory, then we can be sure that there are no other capabilities for the same memory which makes it safe to repurpose the memory and drop or replace the previous assumption. We mark dropped regions as revoked in the world following Ahmed (2004) and Thamsborg & Birkedal (2011) which for all intents and purposes corresponds to actually dropping the region.

We add a revoked tag to the spatial regions $\text{Region}_{\text{spatial}}$:

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}\} \times (\text{World} \overset{mon}{\longrightarrow} \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \cup \\ \{\text{spatial}\} \times (\text{World} \overset{mon}{\longrightarrow} \text{Rel}(\text{MemorySegment} \times \text{MemorySegment})) \cup \\ \{\text{revoked}\} \end{cases}$$

The spatial region may be depended on by a linear capability so we cannot allow it to be revoked. On the other hand, no capability can depend on a shadow region, so it can be safely revoked.

We define the future world relation in terms of a future region relation which is displayed in Figure 16. Apart from being revoked, a region can stay the same, or a shadow region can become spatial. The latter allows us to reassign the spatial region to some other world, when the linear capability owning the region is erased (our linear capabilities are actually affine). The above repurposing is exemplified in Figure 17. The Figure displays two capabilities c_{normal} and c_{linear} that are normal and linear, respectively. The linear capability is valid with respect to W_1 which has the necessary spatial region, and the normal capability is valid with respect to W_2 which has a shared region. The two worlds are compatible as

$$\frac{r \in \text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}}}{r \sqsupseteq r} \qquad \frac{}{\text{revoked} \sqsupseteq (\text{shadow}, \_)}$$

$$\frac{}{(\text{spatial}, H) \sqsupseteq (\text{shadow}, H)}$$

Fig. 16. Future region relation.

W_2 has a shadow region that matches the spatial region of W_1. When the linear capability is repurposed, it must be reflected by the worlds W_1' and W_2'. In both new worlds, the r_2 region is replaced with a revoked region. W_1' has a new spatial region at r_3, and W_2' gets a matching shadow region. The new world for the linear capability, W_1', is not a future world of W_1 as it replaces a spatial region with a revoked region which is not allowed by the future region relation. The new world for the normal capability, W_2', is a future world of W_2 as the future region relation allows shadow regions to be revoked. The normal capability c_{normal} remains valid in W_2' as the shared region it depends on is monotone with respect to the future world relation.

With the future region relation in place, we can define the future world relation as follows: For worlds W and W',

$$W' \sqsupseteq W \text{ iff } \begin{cases} \text{for } i \in \{\text{heap, free\_stk, call\_stk}\} \\ \quad \exists m_i : \text{RegionName} \rightarrow \text{RegionName},\ \text{injective.} \\ \qquad \text{dom}(W'.i) \supseteq m_i(\text{dom}(W.i)) \wedge \forall r \in \text{dom}(W.i).\ W'.i(m_i(r)) \sqsupseteq W.i(r) \end{cases}$$

The relation says that each of the three worlds must be an extension of the past world and each of the existing regions must have a future region. Note that the future world relation has a mapping function m_i which allows us to change the naming of regions in future worlds. The definition is a generalization of the standard definition where m_i would be the identity.[7]

5.1.5 Joining worlds

The world serves multiple purposes as it is both a specification of memory contents as well as a specification of authority. This is best seen in the operators used to join worlds. First when we see the world as a memory specification, we have a pretty standard join $\uplus$ that simply requires the worlds to have different region names.

Definition 9 (World disjoint union $\uplus$)**.** *Given worlds W_1, W_2, W*

$$\begin{aligned} W_1 \uplus W_2 = W \textit{ iff } \ & \text{dom}(W.\text{heap}) = \text{dom}(W_1.\text{heap}) \uplus \text{dom}(W_2.\text{heap}) \wedge \\ & \text{dom}(W.\text{free\_stk}) = \text{dom}(W_1.\text{free\_stk}) \uplus \text{dom}(W_2.\text{free\_stk}) \wedge \\ & \text{dom}(W.\text{call\_stk}) = \text{dom}(W_1.\text{call\_stk}) \uplus \text{dom}(W_2.\text{call\_stk}) \end{aligned}$$

The $\uplus$ world join does not guarantee that the result is a sensible world with respect to authority or memory specification: it may contain regions with conflicting requirements for memory. In Section 5.1.6, we define memory satisfaction that also acts as a well-formedness judgment.

[7] In Skorstengaard *et al.* (2018a), the future region relation and the reasoning about the awkward example could have been simplified with this future world relation.

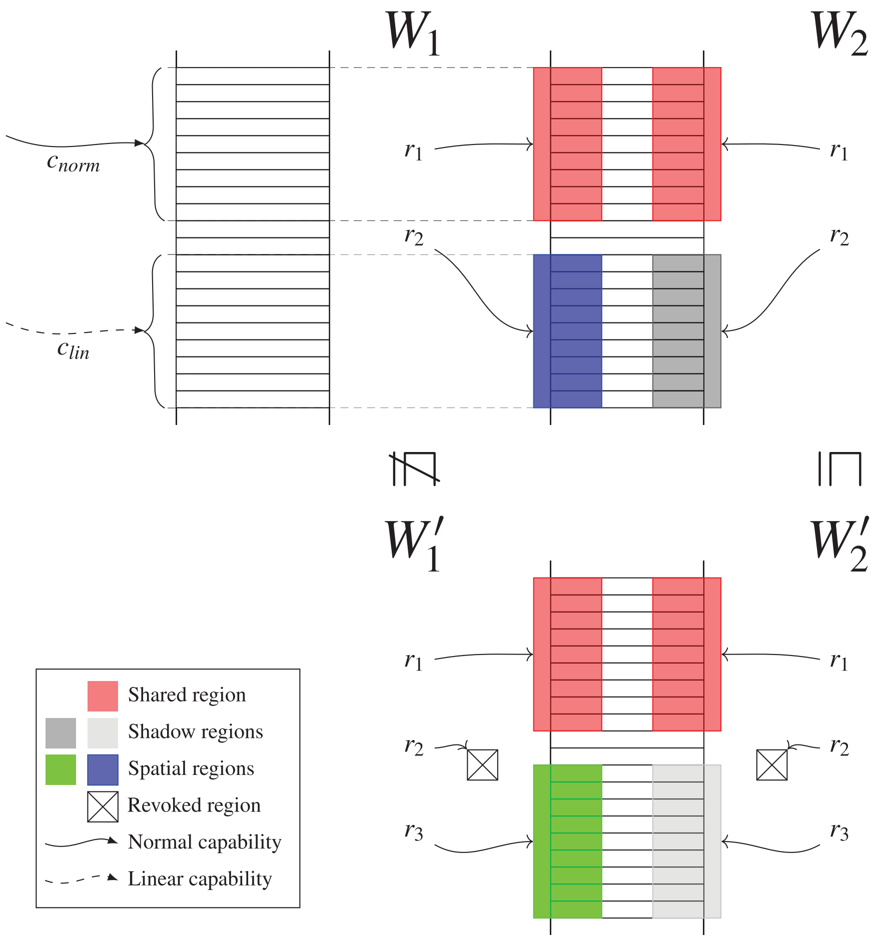

Fig. 17. An example of what happens to worlds when a linear capability c_{lin} is repurposed. The linear capability is originally valid with respect to W_1. In W_1' there is a new spatial region that the linear capability is valid with respect to (the different colored regions signify different invariants). W_1' still has r_2, but now it points to a revoked region. This means that W_1' is not a future world of W_1. The normal capability c_{norm} is originally valid with respect to W_2 and should stay valid with respect to the new world W_2'. The W_2' is a future world of W_2 as the shadow region in W_2 is replaced with a revoked region which is permitted by the future world relation.

When we view the world as a specification of authority, then the world join need to respect the region ownership. That is, when we join the authority of two worlds, then the ownership of the two worlds should not overlap. This is expressed by the $\oplus$ operator.

Definition 10 ($\oplus$, disjoint union of ownership). *$W_1 \oplus W_2 = W$ iff*

$$\begin{aligned}
&\mathrm{dom}(W.\mathrm{heap}) = \mathrm{dom}(W_1.\mathrm{heap}) = \mathrm{dom}(W_2.\mathrm{heap}) \wedge \\
&\mathrm{dom}(W.\mathrm{free\_stk}) = \mathrm{dom}(W_1.\mathrm{free\_stk}) = \mathrm{dom}(W_2.\mathrm{free\_stk}) \wedge \\
&\mathrm{dom}(W.\mathrm{call\_stk}) = \mathrm{dom}(W_1.\mathrm{call\_stk}) = \mathrm{dom}(W_2.\mathrm{call\_stk}) \wedge \\
&\forall r \in \mathrm{dom}(W.\mathrm{heap}).\ W.\mathrm{heap}(r) = W_1.\mathrm{heap}(r) \oplus W_2.\mathrm{heap}(r) \wedge \\
&\forall r \in \mathrm{dom}(W.\mathrm{free\_stk}).\ W.\mathrm{free\_stk}(r) = W_1.\mathrm{free\_stk}(r) \oplus W_2.\mathrm{free\_stk}(r) \wedge \\
&\forall r \in \mathrm{dom}(W.\mathrm{call\_stk}).\ \pi_1(W.\mathrm{call\_stk}(r)) = \pi_1(W_1.\mathrm{call\_stk}(r)) \oplus \pi_1(W_2.\mathrm{call\_stk}(r))
\end{aligned}$$

where $\oplus$ for regions is defined as

$$\begin{aligned}
(\text{shared}, H, H_{\text{seal}}) \oplus (\text{shared}, H, H_{\text{seal}}) &= (\text{shared}, H, H_{\text{seal}}) \\
(\text{shadow}, H) \oplus (\text{shadow}, H) &= (\text{shadow}, H) \\
\text{revoked} \oplus \text{revoked} &= \text{revoked} \\
(\text{spatial}, H) \oplus (\text{shadow}, H) &= (\text{shadow}, H) \oplus (\text{spatial}, H) \\
&= (\text{spatial}, H)
\end{aligned}$$

Like the $\uplus$ operator, the $\oplus$ operator does not guarantee that the resulting world is sensible.

Note that this picture is further complicated by our usage of non-authority-carrying shadow regions. They are really only in a world W as a shadow copy of a spatial region in another world W' that W will be combined with. The shadow copy is used for specifying when a memory satisfies a world: the memory should contain all memory ranges that anyone has authority over, not just the ones whose authority belongs to the memory itself. For example, if a register contains a linear pointer to a range of memory, then the register file will be valid in a world where the corresponding region is spatial, while the memory will be valid in a world with the corresponding region only shadow. However, for the memory to satisfy the world, the block of memory needs to be there, i.e. the memory should contain blocks of memory satisfying every region that is spatial, shared, but also just shadow (because it may be spatial in, for example, the register file's world).

5.1.6 Memory satisfaction

The world can be seen as a specification of the memory contents. This means that we need to define what it means for a pair of LCM and oLCM memories to satisfy the specification. The world also keeps track of the structure of the call stack, the allowed uses of designated seals, and linear capability authority, so these things also influence the definition of memory satisfaction. The world definition on its own allows the invariants imposed by regions to be overlapping. However, memory satisfaction will allow only a single region to govern every piece of memory. It is in this sense that the memory satisfaction also acts as a well-formedness judgment for worlds.

Memory satisfaction is split into four definitions. At the top level, we have $ms_S, ms_{stk}, stk, ms_T :_n^{gc} W$ which relates the source memory triple, ms_S (heap), ms_{stk} (free stack), and stk (stack frames), from oLCM to the target memory ms_T from LCM. The three parts of source memory are related to parts of ms_T by the relations $\mathscr{H}$, $\mathscr{S}$, and $\mathscr{F}$, respectively. We will consider them in this order and discuss $ms_S, ms_{stk}, stk, ms_T :_n^{gc} W$ last.

Note that the judgment $ms_S, ms_{stk}, stk, ms_T :_n^{gc} W$ (and other definitions in this section) is parameterized by $gc = (T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}})$, the global constants that we previously encountered in Section 4.3 when discussing reasonable components.

Definition 11 (Heap relation)**.** *For a set of seals $\overline{\sigma}$, memory segments ms and ms_T, and worlds W and W', we define the heap relation $\mathscr{H}$ as:*

$$(\overline{\sigma}, ms, ms_T) \in \mathcal{H}(W.\text{heap})(W') = \left\{ \begin{array}{l} \exists R_{ms} : \text{dom}(active(W.\text{heap})) \to \text{MemorySegment} \times \text{MemorySegment} \wedge \\ \quad ms_T = \biguplus_{r \in \text{dom}(active(W.\text{heap}))} \pi_2(R_{ms}(r)) \wedge \\ \quad ms = \biguplus_{r \in \text{dom}(active(W.\text{heap}))} \pi_1(R_{ms}(r)) \wedge \\ \quad \exists R_W : \text{dom}(active(W.\text{heap})) \to \text{World}. \\ \qquad W' = \oplus_{r \in \text{dom}(active(W.\text{heap}))} R_W(r) \wedge \\ \qquad \forall r \in \text{dom}(active(W.\text{heap})). \\ \qquad\quad R_{ms}(r) \in W.\text{heap}(r).H\ R_W(r) \wedge \\ \exists R_{seal} : \text{dom}(active(W.\text{heap})) \to \mathcal{P}(\text{Seal}) \wedge \\ \quad \biguplus_{r \in \text{dom}(active(W.\text{heap}))} R_{seal}(r)) \subseteq \overline{\sigma} \wedge \\ \quad \forall r \in \text{dom}(\lfloor W.\text{heap} \rfloor_{\{\text{shared}\}}).\ \text{dom}(W.\text{heap}(r).H_{\text{seal}}) = R_{seal}(r) \end{array} \right.$$

Memory satisfaction, and thus also the heap relation, only considers the nonrevoked regions. The $\mathcal{H}$-relation uses the function *active* to erase all the revoked regions from the world. To a large extent, the definition of $\mathcal{H}$ is pretty standard. It assumes the existence of a partitioning of the LCM and OLCM heap memories that can be turned into memory segment pairs each satisfying the invariant of a region. The heap satisfaction must also respect the world as an authority specification, so the heap satisfaction partitions the authority of the world using $\oplus$. Each of the memory segment pairs must be in the region invariant with respect to a specific world partition which makes sure that uniqueness of linearity of capabilities is respected. The heap subworld contains all seal invariants. Similar to memory segments, only one seal invariant should impose restrictions on a seal, which $\mathcal{H}$ makes sure is the case.

Definition 12 (Free stack relation)**.**

$$(ms_{stk}, ms_T) \in \mathcal{F}^{gc}(W) \textit{ iff} \left\{ \begin{array}{l} gc = (\_, \text{stk\_base}, \_, \_) \wedge \\ W_{stack} = W.\text{free\_stk} \wedge \\ \exists R_{ms} : \text{dom}(active(W_{stack})) \to \text{MemorySegment} \times \text{MemorySegment} \wedge \\ \quad ms_T = \biguplus_{r \in \text{dom}(active(W_{stack}))} \pi_2(R_{ms}(r)) \wedge \\ \quad ms_{stk} = \biguplus_{r \in \text{dom}(active(W_{stack}))} \pi_1(R_{ms}(r)) \wedge \\ \quad \text{stk\_base} \in \text{dom}(ms_T) \wedge \text{stk\_base} \in \text{dom}(ms_{stk}) \wedge \\ \quad \exists R_W : \text{dom}(active(W_{stack})) \to \text{World}. \\ \qquad W = \bigoplus_{r \in \text{dom}(active(W_{stack}))} R_W(r) \wedge \\ \qquad \forall r \in \text{dom}(active(W_{stack})). \\ \qquad\quad R_{ms}(r) \in W_{stack}(r).H\ R_W(r) \end{array} \right.$$

The free stack relation $\mathcal{F}$ is in most regards like the heap relation, $\mathcal{H}$. It partitions the OLCM and LCM free stack memory, it partitions the authority of the world, and it requires the memory segment pairs to be related under part of the world. For StkTokens to work, it should always work on the same stack. As discussed in Section 3, we make sure that it is always the same stack by requiring the address stk_base to be the "top" address of the free stack address space. As the free stack relation relates the stack of OLCM with the memory that represents the stack on LCM, it makes sure that stk_base is the top address of the free stack address space.

Definition 13 (Stack relation)**.**

$$(stk, ms_T) \in \mathcal{S}^{gc}(W) \textit{ iff}$$

$$\left\{\begin{array}{l}
\exists m, opc_0, \ldots, opc_m, ms_0, \ldots, ms_m, W_{stack}. \\
\quad gc = (\_, \text{stk\_base}, \_, \_) \wedge \\
\quad W_{stack} = W.\text{call\_stk} \wedge \\
\quad stk = (opc_0, ms_0), \ldots (opc_m, ms_m) \wedge \\
\quad \forall i \in \{0, \ldots, m\}.\, (\text{dom}(ms_i) \neq \emptyset \wedge \\
\qquad \forall j < i.\, \forall a \in \text{dom}(ms_i).\, \forall a' \in \text{dom}(ms_j).\, a < a' < \text{stk\_base}) \wedge \\
\quad \exists R_{ms} : \text{dom}(active(W_{stack})) \to \text{MemorySegment} \times \text{Addr} \times \text{MemorySegment}. \\
\qquad ms_T = \biguplus_{r \in \text{dom}(active(W_{stack}))} \pi_3(R_{ms}(r)) \wedge \\
\qquad ms_0 \uplus \cdots \uplus ms_m = \biguplus_{r \in \text{dom}(active(W_{stack}))} \pi_1(R_{ms}(r)) \wedge \\
\qquad \exists R_W : \text{dom}(active(W_{stack})) \to \text{World}. \\
\qquad\quad W = \bigoplus_{r \in \text{dom}(active(W_{stack}))} R_W(r) \wedge \\
\qquad\quad \forall r \in \text{dom}(active(W_{stack})). \\
\qquad\qquad (\pi_1(R_{ms}(r)), \pi_3(R_{ms}(r)) \in W_{stack}(r).H\; R_W(r) \wedge \\
\qquad\qquad \pi_2(R_{ms}(r)) = W_{stack}(r).opc \wedge \\
\qquad\qquad \exists i.\, opc_i = W_{stack}(r).opc \wedge ms_i = \pi_1(R_{ms}(r))
\end{array}\right.$$

The stack relation $\mathcal{S}$ is similar to the heap relation in some ways. The $\mathcal{S}$ relation also partitions the LCM memory but not the oLCM memory, since the oLCM stack memory is already partitioned into stack frames. The stack relation also partitions the authority of the world, so it can relate the stack frames in a way that respects linearity. The stack on oLCM represents the call stack which means that each stack frame corresponds to a call and its local data. The operational semantics of LCM does not have a built-in stack, so we emulate it by requiring that a stack like data structure resides in LCM memory. That is for a memory segment that represents a stack frame, all the addresses of memory frames lower in the stack should have strictly smaller memory addresses. Further, the stack frames should be in the part of the memory we agree to be the stack which means that the addresses should be smaller than stk_base. Informally, this just means that the stack should be laid out in memory as a downwards growing stack with no addresses above stk_base.

$\mathcal{S}$ requires every stack frame to be nonempty. As described in Section 3, StkTokens requires nonempty stack frames, so a missing frame can be detected. Note that each stack frame corresponds to a trusted call. Untrusted calls are not protected which means that untrusted stack frames reside in the free stack memory. This means that the protected stack frames are not necessarily packed tightly in memory, and the memory in between is part of the free stack. However, this does not prevent untrusted code from securing their own stack frames. Figure 18 sketches this.

Each stack frame in the oLCM stack contains an old program pointer which corresponds to the old program counter recorded in the region of the world associated with the stack frame. To achieve this, the partition function R_{ms} also records an *opc* for each region, and this *opc* should establish the link between the region and the stack frame.

In order to tie Definitions 11, 12, and 13 together, we define memory satisfaction. Memory satisfaction defines when an oLCM memory, consisting of a heap, a stack, and a free stack, relates to a LCM memory under a world.

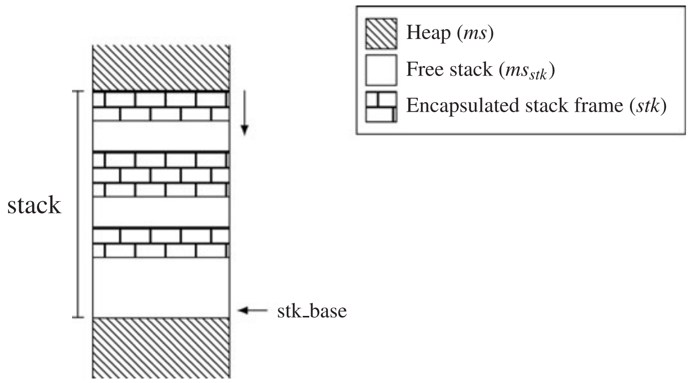

Fig. 18. A sketch of how heap, encapsulated stack and free stack are laid out in memory. The encapsulated stack frames and the free stack constitutes the stack. The encapsulated stack frames may have free stack in between them where stack frames of non-trusted code may reside. The stack grows downwards in memory with stk_base as the top address (and the base address of the free stack capability).

Definition 14 (Memory satisfaction)**.** *For memory segments ms_S, ms_{stk}, and ms_T, stack stk, and world W we define memory satisfaction as*

$$ms_S, ms_{stk}, stk, ms_T :^{gc} W \text{ iff}$$

$$\left\{\begin{array}{l}
\exists m, opc_0, \ldots opc_m, ms_0, \ldots, ms_m, W_{stack}, W_{free\_stack}, W_{heap}. \\
\quad stk = (opc_0, ms_0) :: \cdots :: (opc_m, ms_m) \wedge \\
\quad ms_S \# ms_{stk} \# ms_0 \# \ldots \# ms_m \wedge \\
\quad W = W_{stack} \oplus W_{free\_stack} \oplus W_{heap} \wedge \\
\quad \exists ms_{T,stack}, ms_{T,free\_stack}, ms_{T,heap}, ms_{T,f}, ms_{S,f}, ms'_S, \overline{\sigma}. \\
\qquad ms_S = ms_{f,S} \uplus ms'_S \wedge \\
\qquad ms_T = ms_{T,stack} \uplus ms_{T,free\_stack} \uplus ms_{T,heap} \uplus ms_{T,f} \wedge \\
\qquad \mathrm{dom}(ms_{T,stack} \uplus ms_{T,free\_stack}) = [b_{stk}, e_{stk}] \wedge \\
\qquad b_{stk} - 1, e_{stk} + 1 \in \mathrm{dom}(ms_{T,f}) \wedge \\
\qquad (stk, ms_{T,stack}) \in \mathscr{S}^{gc}(W_{stack}) \wedge \\
\qquad (ms_{stk}, ms_{T,free\_stack}) \in \mathscr{F}^{gc}(W_{free\_stack}) \wedge \\
\qquad (\overline{\sigma}, ms'_S, ms_{T,heap}) \in \mathscr{H}(W.\mathrm{heap})(W_{heap})
\end{array}\right.$$

Memory satisfaction partitions the LCM memory in a heap, stack frames, a free stack and a frame. The OLCM heap is split in two: the active heap and a frame. Our configurations describe the complete machine state, but we may only be interested in the invariants on part of it. The frame allows us to ignore the part of the memory that will not affect the computation. Just like the previous memory relations, the world is split into three to make sure that linearity is respected. Each part of the OLCM memory is related to the appropriate part of the memory from LCM by the relevant relation under a partition of the world.

STKTOKENS requires the stack to not be adjacent to heap or code memory. This is enforced in the memory satisfaction by requiring that the addresses adjacent to the memory are in the frame.

5.2 Constructing worlds: Solving the recursive domain equation

In the previous sections, we sketched what our worlds should be. However, the worlds we want constitute a self-referential domain equation for which no solution exists in set and domain theory. Therefore, we need to move to a different domain with enough structure for a solution to exist for recursive equations. Solutions to recursive domain equations can be found using standard techniques (Scott, 1976; America & Rutten, 1989; Birkedal *et al.*, 2011). Essentially, we move to a setting where instead of sets we have c.o.f.e.'s (complete ordered families of equivalences), instead of functions we have nonexpansive functions, and instead of relations we have downward-closed relations. A c.o.f.e. can be thought of as a set with added structure, specifically a step-indexed notion of equality and a limit to every Cauchy sequence (i.e. they are complete in a similar sense as to how the real numbers are complete but the rationals are not).

Explaining the construction of the world in detail would require a recap of the basic theory of c.o.f.e.'s. For conciseness, we choose to not include this here, but instead refer to the PhD thesis of Skorstengaard (Skorstengaard, 2019, Section 3.5), which includes a detailed explanation. We only include the main result, which is the following theorem, asserting the existence of a World c.o.f.e. satisfying the recursive equation we encountered before.

Theorem 2. *There exists a complete ordered family of equivalences (c.o.f.e.)* Wor *and preorder* $\sqsupseteq$ *such that* $(\text{Wor}, \sqsupseteq)$ *is a preordered c.o.f.e., and there exists an isomorphism* ξ *such that*

$$\xi : \text{Wor} \cong \blacktriangleright(\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}})$$

and for $\hat{W}, \hat{W}' \in \text{Wor}$

$$\hat{W}' \sqsupseteq \hat{W} \textit{ iff } \xi(\hat{W}') \sqsupseteq \xi(\hat{W})$$

for $\text{World}_{\text{call\_stack}}$, $\text{World}_{\text{heap}}$, *and* $\text{World}_{\text{free\_stack}}$ *defined as follows*

$$\begin{aligned}
\text{World}_{\text{heap}} &= \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}}) \\
\text{World}_{\text{call\_stack}} &= \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \times \text{Addr}) \\
\text{World}_{\text{free\_stack}} &= \text{RegionName} \rightharpoonup \text{Region}_{\text{spatial}}
\end{aligned}$$

where $\text{RegionName} = \mathbb{N}$. $\text{Region}_{\text{spatial}}$ *and* $\text{Region}_{\text{shared}}$ *defined as follows*

$$\begin{aligned}
\text{Region}_{\text{shared}} = \{\text{shared}\} \times (&\text{Wor} \xrightarrow{\textit{mon, ne}} \text{URel}(\text{MemorySegment} \times \text{MemorySegment})) \\
&\times (\text{Seal} \rightharpoonup \text{Wor} \xrightarrow{\textit{mon, ne}} \text{URel}(\text{Sealables} \times \text{Sealables}))
\end{aligned}$$

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}, \text{spatial}\} \\ \quad \times (\text{Wor} \xrightarrow{\textit{mon, ne}} \text{URel}(\text{MemorySegment} \times \text{MemorySegment})) \cup \\ \{\text{revoked}\} \end{cases}$$

Theorem 2 uses the method of Birkedal *et al.* (2011), Birkedal & Bizjak (2014) to construct the solution Wor to the recursive equation. Note that Wor is not equal to $\blacktriangleright(\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}})$; it is isomorphic. This means that whenever we encounter Wor, we have to apply ξ and go under a later before we can actually use the world. This makes it rather inconvenient to have Wor as the world, so instead we define the worlds as

$$\mathrm{World} = \mathrm{World}_{\mathrm{heap}} \times \mathrm{World}_{\mathrm{call\_stack}} \times \mathrm{World}_{\mathrm{free\_stack}}$$

For ease of presentation, we have omitted step indices and the isomorphisms ξ from the definitions in Section 5.1.6.

5.3 The logical relation

Using these Kripke worlds as assumptions, we can then define when different OLCM and LCM entities are related: values, jump targets, memories, execution configurations, components, etc. The most important relations are summarized in the following table, where we mention the general form of the relations, what type of things they relate and extra conditions that some of them imply: We use the square symbol as a meta-variable that represents either left-to-right (if $\square = \preceq$) or right-to-left (if $\square = \succeq$) approximation (see Section 5.3.1).

General form	Relates ...	and ...
$(n, (w_S, w_T)) \in \mathcal{V}^{\square, gc}_{\mathrm{untrusted}}(W)$	values (machine words)	safe to pass to adversarial code
$(n, (w_S, w_T)) \in \mathcal{V}^{\square, gc}_{\mathrm{trusted}}(W)$	values (machine words)	
$(n, (reg_S, reg_T)) \in \mathcal{R}^{\square, gc}(W)$	register files	safe to pass to adversarial code
$(n, \Phi_S, \Phi_T) \in \mathcal{O}^{\square, gc}$	execution configurations	
$(n, (w_S, w_T)) \in \mathcal{E}^{\square, gc}(W)$	`jmp` targets	
$\left(\begin{array}{l}(w_{S,1}, w_{S,2}),\\ (w_{T,1}, w_{T,2})\end{array}\right) \in \mathcal{E}^{\square, gc}_{\mathrm{xjmp}}(W)$	`xjmp` targets	
$ms_S, stk, ms_{stk}, ms_T :^{gc}_{n} W$	memory	satisfy the assumptions in W

In Section 5.1.6, we already defined memory satisfaction, the relation for memories. In the following, we define each of the remaining relations and give some intuition about the definitions. The logical relation we define ends up as a cyclic definition. The circularity is resolved by another use of step indexing in the definitions, but the circularity also poses a chicken and egg problem with respect to the order in which the definitions of the relations should be presented. There is no canonical way of presenting the logical relation as we are bound to make forward references. For this reason, we suggest making a cursory first read through to get an overview followed by a more thorough read.

5.3.1 Observation relation

The observation relation defines what machine configurations have related and permissible observable effects. Generally speaking, an observation relation captures the property we want to prove. Ultimately, we want to prove a full-abstraction theorem which is defined in terms of contextual equivalence for components that in turn is defined as cotermination in any context. This means that the observation relation should capture cotermination.

So far, we have talked about the logical relation as though we define only a single one. However, we actually define two logical relations that only differ in the observation relation. The two relations $\mathcal{O}^{\preceq}$ and $\mathcal{O}^{\succeq}$ represent approximation between oLCM and LCM configurations in both directions. The former defines that a oLCM configuration logically approximates a LCM configuration when the halting termination of the oLCM configuration implies the halting termination of the LCM configuration. This also means that oLCM configurations that terminate by failing are related to any LCM configuration. Intuitively, this is because the failed configuration signals that there was an attempt to break the guarantees of the capability machine. For instance, a piece of code could have attempted to read from a part of memory it does not have access to, or a callee could have attempted to return out of order. In both cases, we have not defined a way to recover from such attempts to break the guarantees, so we are content with failure.

$$\mathcal{O}^{\preceq,(T_A,\text{stk\_base},\_,\_)} \stackrel{def}{=}$$

$$\left\{ \left(n, \begin{pmatrix} (ms_S, reg_S, stk_S, ms_{stk,S}), \\ (ms_T, reg_T) \end{pmatrix} \right) \middle| \begin{array}{l} \forall i \leq n. \\ \quad (ms_S, reg_S, stk_S, ms_{stk,S}) \Downarrow_i^{T_A,\text{stk\_base}} \\ \qquad \Rightarrow (ms_T, reg_T) \Downarrow_{-} \end{array} \right\}$$

The step-indexing plays a role here because we are only interested in oLCM configurations that terminate in n or fewer steps. However, if the oLCM configuration terminates successfully in n steps, then the LCM configuration should just terminate in any number of step (possibly more than n steps). For the most part, it would make sense to require the LCM configuration to terminate in the same amount of steps as the oLCM configuration as they run in lockstep for most of the computation. However, when it comes to calls and returns, the two configurations stop running in lockstep. The oLCM configuration handles calls and returns in one step, whereas LCM configurations need to execute each instruction of the call preparation as well as the return code.

The second observation relation $\mathcal{O}^{\succeq}$ defines that a LCM configuration approximates a oLCM configuration in a dual way to the above.

$$\mathcal{O}^{\succeq,(T_A,\text{stk\_base},\_,\_)} \stackrel{def}{=}$$

$$\left\{ \left(n, \begin{pmatrix} (ms_S, reg_S, stk_S, ms_{stk,S}), \\ (ms_T, reg_T) \end{pmatrix} \right) \middle| \begin{array}{l} \forall i \leq n. \\ \quad (ms_T, reg_T) \Downarrow_i \\ \qquad \Rightarrow (ms_S, reg_S, stk_S, ms_{stk,S}) \Downarrow_{-}^{T_A,\text{stk\_base}} \end{array} \right\}$$

The remainder of our logical relation will be the same for both $\preceq$ and $\succeq$, so we will write $\square$ instead of the approximation.

5.3.2 Value relations

The value relation relates LCM words to oLCM words. The oLCM machine has special tokens that represent the stack capabilities and the return pointer components. These tokens do not exist on LCM, but all of the tokens correspond to capabilities on LCM, and the value relation establishes the link between then. Skorstengaard *et al.* (2018a) defines a logical relation that can be seen as a notion of capability safety. When they define their value

$$\begin{aligned}
\mathcal{V}^{\square,gc}_{\text{untrusted}}(W) = {} & \{(n, (i, i)) \mid i \in \mathbb{Z}\} \cup \\
& \{(n, (\text{stack-ptr}(p, b, e, a), ((p, \text{linear}), b, e, a))) \mid \dots\} \cup \\
& \{(n, (\text{seal}(\sigma_b, \sigma_e, \sigma), \text{seal}(\sigma_b, \sigma_e, \sigma))) \mid \dots\} \cup \\
& \{(n, (\text{sealed}(\sigma, sc_S), \text{sealed}(\sigma, sc_T))) \mid \dots\} \cup \\
& \{(n, (((p, l), b, e, a), ((p, l), b, e, a))) \mid \dots\} \\
\mathcal{V}^{\square,gc}_{\text{trusted}}(W) = {} & \mathcal{V}^{\square,gc}_{\text{untrusted}}(W) \cup \\
& \{(n, (\text{seal}(\sigma_b, \sigma_e, \sigma), \text{seal}(\sigma_b, \sigma_e, \sigma))) \mid \dots\} \cup \\
& \{(n, (((p, \text{normal}), b, e, a), ((p, \text{normal}), b, e, a))) \mid p \le \text{RX} \wedge \dots\}
\end{aligned}$$

Fig. 19. Sketches of the trusted and untrusted value relation. The untrusted and trusted value relation both relates OLCM and LCM words. The untrusted value relation $\mathcal{V}_{\text{untrusted}}$ relates words that are safe to give to untrusted programs and $\mathcal{V}_{\text{trusted}}$ relates words that are safe to give to trusted programs.

relation, they define based on the question "What is the most an adversary can be allowed to do with this word without breaking memory invariants?" This allows them to use the logical relation to reason about arbitrary (untrusted) programs. We also want to be able to say something about arbitrary (untrusted) programs, but we also want to be able to say something about somewhat arbitrary trusted programs. In our setting, a trusted program is a well-formed, reasonable program that follows the STKTOKENS calling convention, and an untrusted program is an arbitrary well-formed program. In order for a trusted program to use STKTOKENS, it needs access to return seals, but we cannot allow untrusted programs access to the return seals. A value relation based on what it is safe for an adversary to have should prohibit return seals, so such a relation cannot be used to reason about trusted programs. For this reason, we define two value relations a trusted $\mathcal{V}_{\text{trusted}}$ and an untrusted $\mathcal{V}_{\text{untrusted}}$. Anything safe for unstrusted programs is also safe to give to a trusted program, so the trusted value relation is defined as a super set of the untrusted value relation.

From time to time in this section, we will refer to safety of a capability or a word. In some sense, our logical relation actually ends up as the definition of safety, so when we refer to a capability as *safe* it is in an informal sense where it means that the capability cannot be used break memory invariants.

In Figure 19, we have sketched the two value relations. This shows that for the most part, words on OLCM are related to words on LCM that are syntactically identical. The only exception is stack pointers on OLCM that are related to linear capabilities on LCM. Note that the return pointers of OLCM are not related to anything as it is never safe for any program, trusted or not, to have them. The OLCM return pointers should only occur under a return seal, and they should only be used in a jump in which case the OLCM semantics transforms them to the capabilities they correspond to.

The value relation is defined in terms of a number of auxiliary definitions. In the following, we introduce a number of *standard regions* that express common requirements on memory. Based on the standard regions, we define what we call *permission based conditions*, conditions that a capability with a specific permission must satisfy to be safe.

Standard regions. The notion of regions we defined in Section 5.1 is general enough to allow a wide variety of regions. There are, however, some regions that may seem more natural or standard than others. In particular, when it comes to capability safety, it seems natural to have a region that requires everything in memory to be safe. This is exactly what we refer to as a *standard region* because we usually define a region like that along with a logical relation.

We define a shared, shadow, and spatial standard region. They all have the same invariant which is defined as follows:

$$H^{\text{std},\square}_{A}\; gc\; \hat{W} \overset{def}{=} \left\{ (n, ms_S, ms_T) \;\middle|\; \begin{array}{l} \text{dom}(ms_S) = \text{dom}(ms_T) = A \wedge \\ \exists S : A \to \text{World}.\, \xi(\hat{W}) = \oplus_{a \in A} S(a) \wedge \\ \quad \forall a \in A.\, (n, (ms_S(a), ms_T(a))) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(S(a)) \end{array} \right\}$$

The standard region invariant requires the memory segment pairs to have a specific address space A. Further, the two memory segments must contain words from the untrusted value relation. The memory segments may contain linear capabilities, so we must distribute the ownership of the world between each memory cell which the function S takes care of. Note that the invariant takes a $\hat{W}$ from Wor as argument which means that we must apply the isomorphism ξ before the world can be used. Using this invariant, we define the standard shadow and spatial regions as follows:

$$\iota^{\text{std},v}_{A,gc} \overset{def}{=} (v, H^{\text{std},\square}_{A}\; gc),\; v \in \{\text{shadow}, \text{spatial}\}$$

and the standard shared regions as follows:

$$\iota^{\text{std,shared}}_{A,gc} \overset{def}{=} (\text{shared}, H^{\text{std},\square}_{A}\; gc, \lambda\_\,\_.\, \emptyset)$$

Note that the standard shared region has an empty seal invariant and thus puts no requirements on seals.

Sometimes we need to know that the contents of a memory segment stay the same. For instance, the contents of encapsulated stack frames do not change which we need to be able to rely on. To express this, we define a *static region*. The static region is parameterised with a memory segment pair which is the only memory segment pair the region accepts. The memory invariant is defined as follows:

$$H^{\text{sta},\square}_{(ms_S,ms_T)}\; gc\; \hat{W} \overset{def}{=}$$
$$\left\{ \left(n, \left(ms'_S, ms'_T\right)\right) \;\middle|\; \begin{array}{l} \left(ms'_S, ms'_T\right) = (ms_S, ms_T) \wedge \text{dom}(ms_S) = \text{dom}(ms_T) \wedge \\ \exists S : \text{dom}(ms_S) \to \text{World}.\, \xi(\hat{W}) = \oplus_{a \in \text{dom}(ms)} S(a) \wedge \\ \quad \forall a \in \text{dom}(ms_S).\, (n, (ms_S(a), ms_T(a))) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(S(a)) \end{array} \right\}$$

The region also requires the static memory to contain words from the untrusted value relation. This means that the stack should not be used to store return seals, closure seals, and code pointers for trusted code. With the memory invariant, we define the static region as follows:

$$\iota^{\text{sta},v,\square}_{(ms_S,ms_T),gc} \overset{def}{=} (v, H^{\text{sta},\square}_{(ms_S,ms_T)}\; gc),\; v \in \{\text{shadow}, \text{spatial}\}$$

A shared static region can be defined in a similar fashion to that of the standard region.

In our result, we assume well-formed components which puts certain syntactic constraints on the components. We also have the semantic assumption that trusted components are reasonable. Both assumptions need to be captured in the logical relation in order for us to rely on them. To this end, we define a *code region* which captures the syntactic and semantic assumptions we make on components. The memory invariant of the code region is defined as

$$H^{\text{code}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ (T_A, \_, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}})\ \hat{W} =$$
$$\left\{ \left(n, \begin{pmatrix} code \uplus ms_{\text{pad}}, \\ code \uplus ms_{\text{pad}} \end{pmatrix}\right) \;\middle|\; \begin{array}{l} \exists tst.\ \text{dom}(code) = [b, e] \wedge ([b-1, e+1] \subseteq T_A \wedge \\ \overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \overline{\sigma_{\text{clos}}} \subseteq \overline{\sigma_{\text{glob\_clos}}} \wedge tst = \text{trusted}) \vee \\ \quad ([b-1, e+1] \# T_A \wedge \overline{\sigma_{\text{ret}}} = \emptyset \wedge tst = \text{untrusted}) \wedge \\ ms_{\text{pad}} = [b-1 \mapsto 0] \uplus [e+1 \mapsto 0] \wedge \\ \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} code \wedge \\ \forall a \in \text{dom}(code). \\ \quad (n, (code(a), code(a))) \in \mathcal{V}^{\square, gc}_{tst}(sharedPart(\xi(\hat{W}))) \end{array} \right\}$$

The code region is more restrictive than the standard region. It only allows one memory segment, namely *code* padded with zeroes that make sure that two capabilities cannot be spliced to cause unintended control-flow. We use the relation to reason about trusted components (well formed and reasonable) as well as untrusted components (well formed). The assumptions we can make on the code depend on whether it is part of a trusted or untrusted component. This is captured by requiring the contents of the code memory to be in the trusted or untrusted value relation depending on the trustworthiness of the code. That is, if all the code memory addresses are in the trusted address space and the seals are from the global seals, then the component is trusted. On the other hand, if the code memory addresses are disjoint from the trusted addresses and there are no return seals, then the component is untrusted. In either case, the words should be in the value relation with respect to the *sharedPart* of the world which means that the code memory cannot contain linear capabilities.

StkTokens rely on proper seal usage to guarantee WB and local state encapsulation. This means that components must use return and closure seals for their intended purpose for StkTokens to work. The code region has a seal invariant $H^{\text{code},\square}_{\text{seal}}$ to guarantee that the return and closure seals of the region are used correctly. The seal invariant is displayed in Figure 20. The return seals $\overline{\sigma_{\text{ret}}}$ in a code region should only be used to seal return pointers. That is on oLCM, the return seals should only be used to seal ret-ptr-code and ret-ptr-data. If we allowed any ret-ptr-code to be sealed, then we could not be sure that the ret-ptr-code came from a call even though it should only be possible to get a return pointer from a call. For this reason, we require that the oLCM return pointer actually points to the first address after a call. For a LCM capability related to a oLCM code return pointer, we require it to point to the first address of the return code, not the first address after the call, as the return instructions must be executed.

For sealed data return pointers, we need to know that the world contains a region that governs the local stack frame. That is, there should be a static region with the contents of the stack frame. The fact that it is static signifies that the contents will remain the same. The region that governs the stack frame must come from the call-stack sub-world which means that it is paired with a return address. The return address should correspond to an actual return address of a call in *code*.

$$
\begin{array}{l}
H^{\text{code},\square}_{\text{seal}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ (T_A, \text{stk\_base}, \_, \overline{\sigma_{\text{glob\_ret}}})\ \sigma\ \hat{W} \overset{def}{=} \\
\left\{ \begin{array}{l}
\left(n, \begin{pmatrix} \text{ret-ptr-code}(b, e, a' + \text{call\_len}), \\ ((\text{RX}, \text{normal}), b, e, a) \end{pmatrix}\right) \Bigg| \\
\overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\text{dom}(code) \subseteq T_A \wedge \\
decode(code([a', a' + \text{call\_len} - 1])) = \overline{\texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2} \wedge \\
a = a' + \text{ret\_pt\_offset} \wedge \\
code(a' + off_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma = \sigma_b + off_{\sigma} \in \overline{\sigma_{\text{ret}}} \wedge \\
[a', a' + \text{call\_len} - 1] \subseteq [b, e]
\end{array} \right\} \cup \\
\left\{ \begin{array}{l}
\left(n, \begin{pmatrix} \text{ret-ptr-data}(b, e), \\ ((\text{RW}, \text{linear}), b, e, b - 1) \end{pmatrix}\right) \Bigg| \\
\overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\text{dom}(code) \subseteq T_A \wedge \\
\exists r \in \text{dom}(\xi(\hat{W}).\text{call\_stk}). \\
\quad \xi(\hat{W}).\text{call\_stk}(r) \overset{n}{=} (\iota^{\text{sta},\text{spatial},\square}_{(ms_S, ms_T)}\ (T_A, \text{stk\_base}), a' + \text{call\_len}) \wedge \\
\quad \text{dom}(ms_S) = \text{dom}(ms_T) = [b, e] \wedge \\
\quad decode(code([a', a' + \text{call\_len} - 1])) = \overline{\texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2} \wedge \\
\quad code(a' + off_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma = \sigma_b + off_{\sigma} \in \overline{\sigma_{\text{ret}}}
\end{array} \right\} \\
\hfill \text{for } \sigma \in \overline{\sigma_{\text{ret}}}
\end{array}
$$

$$
\begin{array}{l}
H^{\text{code},\square}_{\text{seal}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ (T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_clos}}}, \overline{\sigma_{\text{glob\_ret}}})\ \sigma\ \hat{W} \overset{def}{=} \\
\left\{ \begin{array}{l}
\big(n, (sc, sc')\big) \Big| \\
(\text{dom}(code) \# T_A \wedge \big(n, (sc, sc')\big) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}\ \xi(\hat{W})) \vee \\
(\text{dom}(code) \subseteq T_A \wedge \overline{\sigma_{\text{clos}}} \subseteq \overline{\sigma_{\text{glob\_clos}}} \wedge \overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\quad ((executable(sc) \wedge \big(n, (sc, sc')\big) \in \mathcal{V}^{\square,gc}_{\text{trusted}}\ \xi(\hat{W})) \vee \\
\quad\ (nonExec(sc) \wedge \big(n, (sc, sc')\big) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}\ \xi(\hat{W}))))
\end{array} \right\} \\
\hfill \text{for } \sigma \in \overline{\sigma_{\text{clos}}}
\end{array}
$$

Fig. 20. The seal invariant for code regions.

Unlike return seals, both trusted and untrusted components can have closure seals. For untrusted components (components with their code address space disjoint from the trusted address space), we allow everything in the untrusted value relation to be sealed. Intuitively, untrusted components are assumed to have access to words from the untrusted value relation, and we cannot know how the words are used, so we need to assume that an untrusted component may seal untrusted words. Trusted components only use closure seals for sealed capability pairs that represent actual closures. The code capability for a closure must point to the code memory because it is the only part of memory that is executable. Untrusted components cannot safely have a capability for a trusted component's code (it could be used to read return capabilities or start execution in the middle of a call), so capabilities for the code memory of a trusted component are in the trusted value relation. While it is not safe to give a bare capability for a trusted components code memory, it can be perfectly safe to give a sealed capability for a trusted components code. For this reason, the seal invariant allows executable capabilities from the trusted value relation to be sealed with a closure seal.

When it comes to the data capability of a closure, we just require that it comes from the untrusted value relation because the trusted value relation contains nothing that makes sense to seal as the data capability (we return to the specific contents of the two value relations later in this section).

With the memory invariant and seal invariant in hand, we define the code region as follows:

$$\iota^{\text{code}}_{\overline{\sigma_{\text{ret}}},\overline{\sigma_{\text{clos}}},code,gc} \stackrel{def}{=} (\text{shared}, H^{\text{code},\square}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ gc, H^{\text{code}}_{\text{seal}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ gc)$$

The code region is shared because it needs to contain a seal invariant and because we assume that code pointers are normal capabilities.

Permission based conditions. The safe capabilities will be defined by the value relation. However, the safety requirements for a capability depend on the authority the capability gives. Therefore, rather than bundling everything into the value relation, we first present a number of *permission-based conditions* that each spell out what the requirements are for each permission.

The world can be seen as an authority specification which means that it dictates what kind of capabilities can address a certain part of memory. Specifically, linear capabilities can only address memory governed by a spatial region, and normal capabilities can only address memory governed by a shared region. All the permission-based conditions we define project the regions that the capability may address from the world. The addressable *addressable* function takes care of the projection:

$$addressable(l, W) \stackrel{def}{=} \begin{cases} \{r \mid W(r) = (\text{shared}, \_)\} & \text{if } l = \text{normal} \\ \{r \mid W(r) = (\text{spatial}, \_)\} & \text{otherwise (i.e. } l = \text{linear)} \end{cases}$$

We capture the essence of what it means for a capability with read permission to be safe in the condition *readCondition*. The main purpose of *readCondition* is to make sure that only safe words can be read from the memory governed by a read capability. This is done by putting an upper bound on what requirements an invariant can impose on the memory segments governed by the capability. In particular, a region that governs the memory a read capability has access to can at most allow safe values to be read. Without this requirement, a read capability could potentially be used to break memory invariants if it were used to read capabilities that has the authority to break memory invariants. The read condition is defined as follows

$$readCondition^{\square,gc}(l, W) = \left\{ (n, A) \;\middle|\; \begin{array}{l} \exists S \subseteq addressable(l, W.\text{heap}). \\ \quad \exists R : S \to \mathscr{P}(\mathbb{N}). \\ \quad\quad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \quad\quad (l = \text{linear} \Rightarrow \forall r \in S.\ |R(r)| = 1) \wedge \\ \quad\quad \forall r \in S.\ W.\text{heap}(r).H \overset{n}{\subseteq} \iota^{\text{std,shared}}_{R(r),gc}.H \end{array} \right\}$$

The *readCondition* is compatible with all the operations that can be performed on capabilities. This means that if two capabilities for which *readCondition* holds are spliced together, we can establish that *readCondition* holds for the resulting capability. To support this, we require the presence of a set of regions S that governs the addresses the capability has authority over rather than just a single region. If we need to establish the *readCondition* after a splice, we can simply use the union of the regions that witnessed the *readCondition* of the two individual capabilities. We also need to support splitting which is no problem for normal capabilities as the same shared region can be used to establish the *readCondition* for multiple normal capabilities. On the other hand, a spatial region can only be used to establish the *readCondition* for one linear capability because the ownership of a spatial region can only go to one world when splitting the ownership. Nonetheless, we need to support arbitrary splitting of linear capabilities, which means that *readCondition* must make sure that the necessary regions are in the world to argue that the result of a split preserves *readCondition*. This is why *readCondition* requires all regions to only govern one address when the capability is linear. This means that after a split, the authority of the regions for the bottom half of the split can go to one capability and the remaining regions can go to the top half.

A safe read capability only gives authority to read safe words. The invariant on the memory a read capability gives access to may be even more restrictive than just requiring safe words. For instance, the invariant may require a flag to stay unchanged. We express the fact that a region may be more restrictive by making the standard region $\iota^{\text{std,shared}}_{R(r),gc}$, which permits all memory segments with safe words, the upper bound of what a region may require when it governs a memory segment that can be accessed through a safe read capability.

Similarly to *readCondition*, we define a condition that captures the essence of what it means for a capability with writer permission to be safe. We call this condition *writeCondition*. A capability with write permission can be used to write to memory. The question is what can we safely allow to be written to memory without any memory invariants being broken. The answer to this is anything – even words that are unsafe. Say, you manage to write something that can break memory invariants, then it would not be possible to read it back again as write permission, generally speaking, does not entail read permission. If the capability had read permission, then *readCondition* would make sure that the word would have to be safe.[8] It should always be possible to write safe values, so we impose this as a lower bound.

A safe write capability must respect the memory invariant of the region that governs the memory the capability gives access to. Now consider the case where the invariant permits two memory segments that differ in two or more addresses. In this case, the write capability cannot be used to transform the memory from one memory segment to the other because only one memory address can be updated at a time. If an adversary had such a capability, then it should be possible for them to transform the memory in a way that is consistent with the region. In other words, the adversarial code should be able to transform the memory segment to any memory segment permitted by the region. This is captured by address stratification (Definition 15) which basically says that if a region permits two memory segments, then all the intermediate memory segments you may end up with when transforming one memory segment to the other must be permitted as well.

[8] It should not be possible to obtain a capability that can be used to break invariants. After all, if such a capability was obtained, memory invariants could be broken. However, the *writeCondition* tries to capture the essence of safety and in principle it is safe to write an unsafe capability that cannot be read back.

Definition 15. *We say that a region $\iota = (\_, H, \_)$ is address stratified iff*

$$\begin{array}{l} \forall n, ms_S, ms_T, ms'_S, ms'_T, s, \hat{W}. \\ \quad (n, (ms_S, ms_T)), \left(n, \left(ms'_S, ms'_T\right)\right) \in H\ \hat{W} \wedge \\ \quad \mathrm{dom}(ms_S) = \mathrm{dom}(ms_T) = \mathrm{dom}(ms'_S) = \mathrm{dom}(ms'_T) \\ \quad \Rightarrow \\ \qquad \forall a \in \mathrm{dom}(ms_S).\ \left(n, (ms_S[a \mapsto ms'_S(a)], ms_T[a \mapsto ms'_T(a)])\right) \in H\ \hat{W} \end{array}$$

With address stratification defined, we define the write condition.

Definition 16.

$$\mathit{writeCondition}^{\square,gc}(l, W) \stackrel{def}{=} \left\{ (n, A) \;\middle|\; \begin{array}{l} \exists S \subseteq \mathit{addressable}(l, W.\mathrm{heap}). \\ \quad \exists R : S \to \mathscr{P}(\mathbb{N}) \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad (l = \mathrm{linear} \Rightarrow \forall r \in S.\ |R(r)| = 1) \wedge \\ \qquad \forall r \in S.\ W.\mathrm{heap}(r).H \overset{n}{\supseteq} \iota^{\mathrm{std,shared}}_{R(r),gc}.H \wedge \\ \qquad\quad W.\mathrm{heap}(r) \textit{ is address-stratified} \end{array} \right\}$$

The definition of *writeCondition* is very similar to *readCondition*. Support for split and splice is done in the same way, and the bound is defined in terms of the standard region.

The *readCondition* and *writeCondition* specifically use the heap sub-world which means that it can only be used for heap capabilities. This means that we cannot use it for stack capabilities. To take care of stack capabilities, we define two more conditions: a *stackReadCondition* and *stackWriteCondition*. The two new conditions are essentially the same as the *readCondition* and *writeCondition* except that they use the free stack sub-world and assume that the capability is linear as all stack capabilities are linear. Note that we do not have any condition that talks about the stack-frames sub world because we should never have a capability that allows us to directly read from or write to that part of memory.

Definition 17.

$$\mathit{stackReadCondition}^{\square,gc}(W) = \left\{ (n, A) \;\middle|\; \begin{array}{l} \exists S \subseteq \mathit{addressable}(\mathrm{linear}, W.\mathrm{free\_stk}). \\ \quad \exists R : S \to \mathscr{P}(\mathbb{N}). \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad \forall r \in S.\ |R(r)| = 1 \\ \qquad \forall r \in S.\ W.\mathrm{free\_stk}(r).H \overset{n}{\subseteq} \iota^{\mathrm{std,shared}}_{R(r),gc}.H \end{array} \right\}$$

Definition 18.

$$\mathit{stackWriteCondition}^{\square,gc}(W) = \left\{ (n, A)) \;\middle|\; \begin{array}{l} \exists S \subseteq \mathit{addressable}(\mathrm{linear}, W.\mathrm{free\_stk}). \\ \quad \exists R : S \to \mathscr{P}(\mathbb{N}) \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad \forall r \in S.\ |R(r)| = 1 \wedge \\ \qquad \forall r \in S.\ W.\mathrm{free\_stk}(r).H \overset{n}{\supseteq} \iota^{\mathrm{std,shared}}_{R(r),gc}.H \wedge \\ \qquad\quad W.\mathrm{free\_stk}(r) \textit{ is address-stratified} \end{array} \right\}$$

The final permission we define conditions for is the execute permission. We define two conditions *executeCondition* and *readXCondition*. The *executeCondition* captures what operations an execute capability can be used for, i.e. execution. The *readXCondition* captures some additional read assumptions we can make on a capability when we know the capability is executable.

The *executeCondition* intuitively says that an execute capability is safe when any capability that can be derived from it is safe as a program counter now and in the future. We later define the $\mathscr{E}$-relation which captures what it means for a word to be safe as a program counter, but for now it suffices to think of it as a program counter that causes an execution that does not break memory invariants. An executable capability can have its range of authority shrunk or its current address changed which changes what instructions are executed and thus potentially whether the code respects memory invariants. For this reason, the condition requires that any executable capability with a derived range of authority and a current address in that range is safe to use for execution. The *executeCondition* is quantified over all future worlds of the *sharedPart* of W. We do not know when the executable capability will be used, so it should be safe even in the future when the memory has changed. The *sharedPart* function turns the spatial regions of a world into shadow copies. This means that the capability cannot depend on linear capabilities and thus the contents of the stack. When we define the logical relation, we even require the executable capability to not be linear. Linear executable capabilities would likely not be useful because they cannot be moved from the pc-register without crashing the execution. This may sound like an ideal primitive for constructing something that can be executed once; however, most programs rely on loading other capabilities or seal sets using the program counter capability which is not possible when the program counter is linear.

$$executeCondition^{\square,gc}(W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq sharedPart(W).\, \forall b', e'.\, \forall a \in [b', e'] \subseteq A. \\ \left(n', \begin{pmatrix} ((\mathrm{RX}, \text{normal}), b', e', a), \\ ((\mathrm{RX}, \text{normal}), b', e', a) \end{pmatrix}\right) \in \mathscr{E}^{\square,gc}(W') \end{array} \right\}$$

The *readCondition* condition by itself allows many different regions and thus potentially many different memory segments. However, when we have a read capability with execute permission, we know that the capability must point to a piece of code memory. For this reason, we define the *readXCondition* to capture the additional assumptions that we can make when a capability is executable.

$$readXCondition^{\square,gc}(W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \exists r \in addressable(\text{normal}, W.\text{heap}), code. \\ \quad W.\text{heap}(r) \overset{n}{=} \iota^{\text{code}}_{\_,\_,code,gc} \wedge \\ \quad \text{dom}(code) \supseteq A \end{array} \right\}$$

The *readXCondition* requires that the memory segment an executable capability has authority over is governed by a code region. Note that we do not define *executeCondition* and *readXCondition* for the stack because the stack is not executable.

The *executeCondition* handles normal jumps, but it does not cover the case of `xjmp`. Executable capabilities can be used on their own, whereas sealed capabilities must be jumped to in pairs. However, we do not need to consider arbitrary pairs: given a sealed capability we only have to consider the capabilities permitted by the relevant seal invariant. Just like the $\mathscr{E}$ relation captures what it means for a word to be safe as a program counter, we later define $\mathscr{E}_{\text{xjmp}}$ that defines what it means for a code and data capability pair to be safe together as program counter and data capability, respectively. A sealed capability is safe when it can be paired with any sealed capability from the seal invariant such that the pair is in the $\mathscr{E}_{\text{xjmp}}$ relation. Just like safe executable capabilities, a sealed capability may be stored, so it should also be safe to use in future worlds. The condition for sealed capabilities is defined by *sealedCondition*.

$$\mathit{sealedCondition}^{\square,gc}(W, H_{\text{seal}}) = \left\{ (n, (\sigma, sc_S, sc_T)) \;\middle|\; \begin{array}{l} \forall W' \sqsupseteq W, W_o, n' < n, \left(n', \left(sc'_S, sc'_T\right)\right) \in H_{\text{seal}}\ \sigma\ \xi^{-1}(W_o). \\ \quad \left(n', sc_S, sc'_S, sc_T, sc'_T\right) \in \mathscr{E}^{\square,gc}_{\text{xjmp}}(W' \oplus W_o)) \end{array} \right\}$$

The untrusted value relation. The untrusted value relation $\mathscr{V}_{\text{untrusted}}$ relates all the words that untrusted components can safely possess. That is words that cannot be used to break any memory invariants. The relation is displayed in Figure 21.

The untrusted value relation has five cases: data, capabilities, stack pointers, sealed capabilities, seal sets, and stack pointers. In the following, we will give some intuition about why it is safe to give these words to untrusted code as well motivate the conditions they are safe under.

The first case is data. Data grant no authority, so data are always safe. Further unlike capabilities, it is always possible to construct a new integer with the move instruction.

Next we have capabilities that do not have a special representation on OLCM, i.e. all capabilities but stack pointers and return pointers. For two capabilities to be related, they should be syntactically equal. That is, they should have the same range of authority, linearity and so one. Generally speaking, untrusted components should not have direct access to a trusted components code, so we require that capabilities must have a range of authority outside the trusted address space if they are to be related. The safety of a capability also depends on the world and whether the capability can be used to break the memory invariants of the world. For instance, if a capability has read-permission, then it should not be possible to read something unsafe, i.e. something that can break memory invariants. This condition and conditions for the other permissions are captured by the permission-based conditions, so we use them to express the necessary conditions. That is, if a capability has read permission, then *readCondition* must be satisfied, if it has write permission, then *writeCondition* must be satisfied, and if it has execute permission, then *executeCondition* and *readXCondition* must be satisfied. If the capability has execute permission, then it must also be a normal capability. Finally, the capability cannot have read/write/execute permission because that would break the write-XOR-execute assumption, i.e. the code memory in nonwritable and data memory is nonexecutable.

Stack pointers on OLCM are represented with the special token stack-ptr(p, b, e, a). The corresponding capability on LCM is a linear capability with the same permission and addresses. We assume that the stack is nonexecutable, so the permission for a stack pointer cannot have execute permission. Similarly to the normal capabilities, we use the permission based conditions for the stack to ensure that the stack capability is safe to use.

$$
\begin{aligned}
\mathcal{V}^{\square,gc}_{\text{untrusted}}(W) = {} & \{(n,(i,i)) \mid i \in \mathbb{Z}\} \cup \\
& \left\{ \left(n, \begin{pmatrix} ((p,l),b,e,a), \\ ((p,l),b,e,a) \end{pmatrix}\right) \;\middle|\; \begin{array}{r} [b,e] \,\#\, T_A \wedge p \neq \mathrm{RWX} \wedge \\ p \in \mathit{readAllowed} \Rightarrow (n,[b,e]) \in \mathit{readCondition}^{\square,gc}(l,W) \wedge \\ p \in \mathit{writeAllowed} \Rightarrow (n,[b,e]) \in \mathit{writeCondition}^{\square,gc}(l,W) \wedge \\ p = \mathrm{RX} \Rightarrow (n,[b,e]) \in \mathit{executeCondition}^{\square,gc}(W) \wedge \\ (n,[b,e]) \in \mathit{readXCondition}^{\square,gc}(W) \wedge \\ l = \text{normal} \end{array} \right\} \cup \\
& \left\{ \left(n, \begin{pmatrix} \text{stack-ptr}(p,b,e,a), \\ ((p,\text{linear}),b,e,a) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} p \notin \{\mathrm{RX},\mathrm{RWX}\} \wedge \\ p \in \mathit{readAllowed} \Rightarrow (n,[b,e]) \in \mathit{stackReadCondition}^{\square,gc}(W) \wedge \\ p \in \mathit{writeAllowed} \Rightarrow (n,[b,e]) \in \mathit{stackWriteCondition}^{\square,gc}(W) \end{array} \right\} \cup \\
& \left\{ \left(n, \begin{pmatrix} \text{sealed}(\sigma, sc_S), \\ \text{sealed}(\sigma, sc_T) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} (\mathit{isLinear}(sc_S) \text{ iff } \mathit{isLinear}(sc_T)) \wedge \\ \exists r \in \mathrm{dom}(W.\text{heap}), \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, ms_{\text{code}}. \\ \quad W.\text{heap}(r) = (\text{shared}, \_, H_{\text{seal}}) \wedge \\ \quad H_{\text{seal}}\ \sigma \overset{n}{=} H^{\text{code},\square}_{\text{seal}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ ms_{\text{code}}\ gc\ \sigma \wedge \\ \quad \big(n', (sc_S, sc_T)\big) \in H_{\text{seal}}\ \sigma\ \xi^{-1}(W) \text{ for all } n' < n \wedge \\ \quad \mathit{isLinear}(sc_S) \Rightarrow (n,(\sigma, sc_S, sc_T)) \in \\ \qquad\qquad \mathit{sealedCondition}^{\square,gc}(W, H_{\text{seal}}) \wedge \\ \quad \mathit{nonLinear}(sc_S) \Rightarrow (n,(\sigma, sc_S, sc_T)) \in \\ \qquad\qquad \mathit{sealedCondition}^{\square,gc}(\mathit{purePart}(W), H_{\text{seal}}) \end{array} \right\} \cup \\
& \left\{ \left(n, \begin{pmatrix} \text{seal}(\sigma_b, \sigma_e, \sigma), \\ \text{seal}(\sigma_b, \sigma_e, \sigma) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} [\sigma_b, \sigma_e] \,\#\, (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}}) \wedge \\ \forall \sigma' \in [\sigma_b, \sigma_e].\ \exists r \in \mathrm{dom}(W.\text{heap}). \\ \quad W.\text{heap}(r) = (\text{shared}, \_, H_{\text{seal}}) \wedge \\ \quad H_{\text{seal}}\ \sigma' \overset{n}{=} (\mathcal{V}^{\square,gc}_{\text{untrusted}} \circ \xi) \end{array} \right\}
\end{aligned}
$$

Fig. 21. The untrusted value relation relates all the words on oLCM to all the words on LCM that are safe for non-trusted components to posses.

A sealed capability encapsulates the authority of the underlying capability, and the authority is only released when the sealed capability is used in an `xjmp`. The `xjmp` takes a pair of sealed capabilities, so the authority of a sealed capability depends on what other sealed capabilities it might be used with. The seal invariant specifies the capabilities that may be sealed with a given seal and thus the capabilities that may be used together as a sealed pair. As discussed, closure and return seals must be used in specific ways which is captured in the code region seal invariant. In order for a pair of oLCM and LCM sealed capabilities to be in the untrusted value relation, they must be sealed with the same seal σ and related in the appropriate seal invariant. Further, they should satisfy the *sealedCondition* which means that they can safely be paired up with any other pair of capabilities from the seal invariant and used safely for execution.

$$
\begin{aligned}
&\mathcal{V}^{\square,gc}_{\text{trusted}}(W) = \\
&\quad \mathcal{V}^{\square,gc}_{\text{untrusted}}(W) \cup \\
&\quad \left\{ \left(n, \begin{pmatrix} \mathrm{seal}(\sigma_b, \sigma_e, \sigma), \\ \mathrm{seal}(\sigma_b, \sigma_e, \sigma) \end{pmatrix} \right) \middle| \begin{array}{l} gc = (T_A, \mathrm{stk\_base}, \overline{\sigma_{\mathrm{glob\_ret}}}, \overline{\sigma_{\mathrm{glob\_clos}}}) \wedge \\ \exists r \in \mathrm{dom}(W.\mathrm{heap}). \\ \quad W.\mathrm{heap}(r) \overset{n}{=} \iota^{\mathrm{code}}_{\overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, code, gc} \wedge \\ \quad \mathrm{dom}(code) \subseteq T_A \wedge \\ \quad [\sigma_b, \sigma_e] \subseteq (\overline{\sigma_{\mathrm{ret}}} \cup \overline{\sigma_{\mathrm{clos}}}) \wedge \\ \quad \overline{\sigma_{\mathrm{ret}}} \subseteq \overline{\sigma_{\mathrm{glob\_ret}}} \wedge \overline{\sigma_{\mathrm{clos}}} \subseteq \overline{\sigma_{\mathrm{glob\_clos}}} \end{array} \right\} \cup \\
&\quad \left\{ \left(n, \begin{pmatrix} ((p, \mathrm{normal}), b, e, a), \\ ((p, \mathrm{normal}), b, e, a) \end{pmatrix} \right) \middle| \begin{array}{l} p \sqsubseteq \mathrm{RX} \wedge \\ gc = (T_A, \mathrm{stk\_base}, \overline{\sigma_{\mathrm{glob\_ret}}}, \overline{\sigma_{\mathrm{glob\_clos}}}) \wedge \\ [b, e] \subseteq T_A \wedge \\ (n, [b, e]) \in \mathit{readXCondition}^{\square,gc}(W) \end{array} \right\}
\end{aligned}
$$

Fig. 22. The trusted value relation $\mathcal{V}_{\text{trusted}}$ relates all the words that are safe for trusted components to contain. A trusted component may contain untrusted words (Figure 21), return seals and trusted closure seals, and code pointers for trusted code.

For sets of seals to be related in the untrusted relation, they must be syntactically equal. Further, the seals in the set should be disjoint from the return seals and trusted closure seals ($\overline{\sigma_{\mathrm{glob\_ret}}}$ and $\overline{\sigma_{\mathrm{glob\_clos}}}$) because the trusted code relies on having the sole access to them. We do not know what an adversary may seal or what seal they may use, so, for every seal in the seal set, we require the seal invariant to be essentially equal to the untrusted value relation.

When we give a word to untrusted code, we can make no assumptions on when they will use it. For instance, they may store it in memory and use it in a later call. This means that a safe word must not only be safe now but also at any point in the future. The untrusted value relation ensures this as it is monotone with respect to future worlds.

Lemma 1 (Untrusted value relation monotonicity). *For all integers n, words w_1 and w_2, and worlds $W' \sqsupseteq W$, if $(n, (w_1, w_2)) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(W)$, then $(n, (w_1, w_2)) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(W')$.*

The trusted value relation. The trusted value relation $\mathcal{V}_{\text{trusted}}$ relates everything safe for a trusted component to have without breaking memory invariants. For the most part, we allow them to contain the same words as the untrusted components, but we also need to allow them to have seal sets with trusted closure seals and return seals which we cannot allow untrusted components to have. Further, we need to allow trusted components to have capabilities for the trusted code which, again, is something that we cannot allow untrusted components to have. The trusted value relation is defined in Figure 22.

The words in $\mathcal{V}_{\text{trusted}}$ but not in the $\mathcal{V}_{\text{untrusted}}$ have the potential to break the system invariants StkTokens rely on. We can only let trusted components have words from $\mathcal{V}_{\text{trusted}}$ because the trusted component promises to not break the invariant by behaving reasonably. This promise is expressed formally in $\mathcal{V}_{\text{trusted}}$ by requiring the presence of a code region in both cases specific to $\mathcal{V}_{\text{trusted}}$. As explained previously, the code region essentially captures the requirements put on components by well formedness and the reasonability condition which constitutes the promise to use seals and trusted code pointers in a way that does not break invariants.

The trusted closure seals and return seals serve a specific purpose, namely, they must be used for return pointers and closures. To make sure this is the case, there must be a code region in the world that governs the code. The code region contains a seal invariant that makes sure that the seals are only used for their intended purpose. This is why the trusted value relation only relates seal sets of trusted closure seals and return seals when the world contains an appropriate code region.

Two capabilities are related as trusted code pointers if they are normal, have a permission derivable from read-execute, and have a range of authority within the trusted address space, T_A. Further, we need to know that the capabilities actually point to a piece of code which is why we require the *readXCondition* to be satisfied. This makes sure that the region that governs the memory the capability points to is a code region. Note that even though the capability has read permission, we do not require the read condition to hold. The code memory contains trusted closure seals and return seals that we cannot let untrusted code have and the read condition requires everything to be in $\mathcal{V}_{\text{untrusted}}$, so the read condition would not hold. However, trusted code can have access to such seals because we expect the trusted code to treat the seals reasonably. Like the untrusted value relation, the trusted value relation is monotone. Intuitively, the two relations are monotone for the same reason; words are potentially used at any point in time. If words are safe now (in the current world), then they should also be safe to use later (in any possible future world).

Lemma 2 (Trusted value relation monotonicity)**.** *For all integers n, words w_1 and w_2, and worlds $W' \sqsupseteq W$, if $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W)$, then $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W')$.*

Another, perhaps unsurprising property, of the trusted value relation it that nonlinear words do not depend on the spatial regions that may be in the world. This is unsurprising as normal capabilities do not necessarily reference memory uniquely.

Lemma 3 (Nonlinear words are independent of spatial regions)**.** *If $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W)$ and either $nonLinear(w_1)$ or $nonLinear(w_2)$, then*

$$(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(sharedPart(W)).$$

This is similar to Lemma 3 for the untrusted value relation.

5.3.3 Register file relation

The register file relation relates oLCM register files to LCM register files. Intuitively, two register files are related when they only contain safe words, i.e. words from the value relation. This raises the question "which value relation?" We only use the register file relation to relate register files for components we do not trust, so the answer is the untrusted value relation. The definition of the register-file relation is straightforward. It distributes

$$\mathcal{R}^{\square,gc}(R)(W) = \left\{ \left(n, \left(reg_S, reg_T\right)\right) \middle| \begin{array}{l} \exists S : (\mathrm{RegName} \setminus (\{\mathrm{pc}\} \cup R)) \to \mathrm{World}. \\ \quad W = \bigoplus_{r \in (\mathrm{RegName} \setminus (\{\mathrm{pc}, \mathrm{r}_{\mathrm{data}}\} \cup R))} S(r) \wedge \\ \quad \forall r \in \mathrm{RegName} \setminus (\{\mathrm{pc}\} \cup R). \\ \qquad \left(n, (reg_S(r), reg_T(r))\right) \in \mathcal{V}^{\square,gc}_{\mathrm{untrusted}}(S(r)) \end{array} \right\}$$

Fig. 23. The register file relation relates register files. Two register files are related when their content is related.

the authority of the world among the registers and requires each of the registers to contain a safe word with respect to the authority it is given. The register file never takes into account the pc and it can leave out further registers. We use this to not relate register content that will be overwritten anyway. We write $\mathcal{R}(W)$ to mean $\mathcal{R}(\emptyset)(W)$. That is, if we do not need to exclude additional registers, then we simply omit that argument. The register file relation is defined in Figure 23.

5.3.4 Expression relations

The expression relation $\mathcal{E}$ defines when two capabilities can be used in the pc-register to produce related executions, i.e. the capabilities can be used to construct configurations in the observation relation. The $\mathcal{E}$ relation can be used to reason about the safety of an executable capability, i.e. capabilities that can change the control flow during execution when a `jmp` instruction is executed. In the setting of OLCM and LCM, we can also use sealed capabilities to change the control flow by using the `xjmp` instruction. The `xjmp` instruction updates the pc register and the $\mathrm{r}_{\mathrm{data}}$ register; however, the $\mathcal{E}$ relation only updates the *pc*-register, so we cannot use $\mathcal{E}$ to reason about sealed capabilities. Instead, we define the relation $\mathcal{E}_{\mathrm{xjmp}}$ which relates two pairs of capabilities when they are safe to use with the `xjmp` instruction.

Executions are related when the observable effects of the executions are permissible. The permissible observations are defined by the observation relation, so we define the expression relation in terms of the observation relation. However, the observation relation relates configurations, not capabilities. We lift the capabilities to configurations simply by plugging the two capabilities into the pc-register of two configurations. We cannot pick arbitrary configurations because an arbitrary configuration may contain words that can be used to break memory invariants and thus create unacceptable observable effects. Instead, we need to pick configurations made out of related components, i.e. related register files and related memories that respect linearity. The type of execution captured by $\mathcal{E}$ corresponds to a normal jump. When a `jmp` r instruction is interpreted, the pc-register is replaced with the contents of register r, i.e. the current configuration is plugged with a new pc. The $\mathcal{E}$ relation is defined in Figure 24

The $\mathcal{E}_{\mathrm{xjmp}}$ relation looks very much like the $\mathcal{E}$ relation. It takes related memories and register files (ignoring the $\mathrm{r}_{\mathrm{data}}$ register) and combines them into two configurations. Each of the configurations is plugged with a code and data capability just like the `xjmp` instruction would and requires the resulting configurations to be in the $\mathcal{O}$ relation. The $\mathcal{E}_{\mathrm{xjmp}}$ relation is defined in Figure 24.

$$\mathcal{E}^{\square,gc}(W) = \left\{ (n, (v_{c,S}, v_{c,T})) \middle| \begin{array}{l} \forall n' \leq n, W_{\mathcal{R}}, W_{\mathcal{M}}, reg_S, reg_T, ms_S, ms_T, ms_{stk}, stk. \\ \quad (n', (reg_S, reg_T)) \in \mathcal{R}^{\square,gc}(W_{\mathcal{R}}) \wedge \\ \quad ms_S, stk, ms_{stk}, ms_T :_{n'}^{gc} W_{\mathcal{M}} \\ \quad \Phi_S = (ms_S, reg_S, stk, ms_{stk}) \\ \quad \Phi'_S = \Phi_S[reg.\mathrm{pc} \mapsto v_{c,S}] \\ \quad \Phi_T = (ms_T, reg_T) \\ \quad \Phi'_T = \Phi_T[reg.\mathrm{pc} \mapsto v_{c,T}] \\ \quad W \oplus W_{\mathcal{R}} \oplus W_{\mathcal{M}} \\ \quad \Rightarrow (n', (\Phi'_S, \Phi'_T)) \in \mathcal{O}^{\square,gc} \end{array} \right\}$$

$$\mathcal{E}^{\square,gc}_{\mathrm{xjmp}}(W) = \left\{ \begin{array}{r} (n, (v_{c,S}, v_{d,S}, \\ v_{c,T}, v_{d,T})) \end{array} \middle| \begin{array}{l} \forall n' \leq n, reg_S, reg_T, ms_S, ms_T, ms_{stk}, stk. \\ \quad \forall W_{\mathcal{R}}, W_{\mathcal{M}}. \\ \quad\quad (n', (reg_S, reg_T)) \in \mathcal{R}^{\square,gc}(\{\mathrm{r_{data}}\})(W_{\mathcal{R}}) \wedge \\ \quad\quad ms_S, stk, ms_{stk}, ms_T :_{n'}^{gc} W_{\mathcal{M}} \wedge \\ \quad\quad \Phi_S = (ms_S, reg_S, stk, ms_{stk}) \wedge \\ \quad\quad \Phi_T = (ms_T, reg_T) \wedge \\ \quad\quad W \oplus W_{\mathcal{R}} \oplus W_{\mathcal{M}} \text{ is defined} \\ \quad\quad \Rightarrow \exists \Phi'_S, \Phi'_T. \\ \quad\quad\quad \Phi'_S = xjumpResult(v_{c,S}, v_{d,S}, \Phi_S) \wedge \\ \quad\quad\quad \Phi'_T = xjumpResult(v_{c,T}, v_{d,T}, \Phi_T) \wedge \\ \quad\quad\quad (n', (\Phi'_S, \Phi'_T)) \in \mathcal{O}^{\square,gc} \end{array} \right\}$$

Fig. 24. The expression relation relates capabilities that can safely be used for execution. The xjmp expression relation relates capabilities that are safe as sealed capabilties.

5.4 Fundamental theorem

An important lemma in our proof of full abstraction of the embedding of oLCM into LCM is the fundamental theorem of logical relations (FTLR). The name indicates that it is an instance of a general pattern in logical relations proofs, but is otherwise unimportant.

Theorem 3 (FTLR)**.** *For all n, W, l, b, e, a, If*

- $(n, [b, e]) \in readXCondition^{\square,gc}(W)$

and one of the following sets of requirements holds:

- $[b, e] \subseteq T_A$ *and* $(((\mathrm{RX}, \mathrm{normal}), b, e, a)$ *behaves reasonably up to* n *steps (see Section 4.2).*
- $[b, e] \# T_A$

then

$$(n, (((\mathrm{RX}, \mathrm{normal}), b, e, a), ((\mathrm{RX}, \mathrm{normal}), b, e, a))) \in \mathcal{E}^{\square,gc}(W)$$

Roughly speaking, this lemma says that under certain conditions, executing any executable capability under oLCM and LCM semantics will produce the same observable behavior. The conditions require that the capability points to a memory region where code is loaded and that code must be either trusted and behave reasonably (i.e. respect the restrictions that StkTokens relies on, see Section 4.2) or untrusted (in which case, it cannot have WBCF or LSE expectations, see Section 4.2).

The proof of the lemma consists of a big induction where each possible instruction is proven to behave the same in source and target in related memories and register files. After that first step, the induction hypothesis is used for the rest of the execution.

5.5 Related components

In order to show full abstraction (Theorem 1), we need not only to relate the words on oLCM with words on LCM we also need to relate oLCM components with LCM components. Specifically, we say that two components are related when they are syntactically equal, after all, an oLCM component is in some sense the same as a LCM as we only see the difference during execution when a call happens and when we lift a component to a configuration where we need to introduce a stack pointer. However, we cannot take arbitrary components as they could potentially break memory invariants. For related base components, we require that if the imports are satisfied with related words, then the resulting memory should be safe. Further, related components must have safe exports. Components with a main are related when the base components are related and the main capabilities are in the public interface, that is, they must be in the exports.

Definition 19 (Component relation)**.**

$$
\begin{aligned}
&\mathcal{C}^{\square,gc}(W) = \\
&\quad \left\{ \begin{array}{l}
(n, comp, comp) \mid \\
\quad comp = (ms_{\text{code}}, ms_{\text{data}}, \overline{a_{\text{import}} \hookleftarrow s_{\text{import}}}, \overline{s_{\text{export}} \mapsto w_{\text{export}}}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}) \textit{ and} \\
\quad \textit{For all } W' \sqsupseteq W. \\
\qquad \textit{If } \overline{\left(n', (w_{\text{import}}, w_{\text{import}})\right)} \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(\textit{sharedPart}(W')) \textit{ for all } n' < n \\
\qquad \textit{and } ms'_{\text{data}} = ms_{\text{data}}[\overline{a_{\text{import}} \mapsto w_{\text{import}}}] \\
\qquad \textit{then } \left(n, (\overline{\sigma_{\text{ret}}} \uplus \overline{\sigma_{\text{clos}}}, ms_{\text{code}} \uplus ms'_{\text{data}}, ms_{\text{code}} \uplus ms'_{\text{data}})\right) \\
\qquad\qquad\qquad\qquad \in \mathcal{H}(W.\text{heap})(W') \textit{ and} \\
\qquad \overline{\left(n, (w_{\text{export}}, w_{\text{export}})\right)} \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(\textit{sharedPart}(W'))
\end{array} \right\} \\
&\quad \cup \left\{ \begin{array}{l}
\left(n, (comp_0, c_{\text{main},c}, c_{\text{main},d}), (comp_0, c_{\text{main},c}, c_{\text{main},d})\right) \mid \\
\qquad \left(n, (comp_0, comp_0)\right) \in \mathcal{C}^{\square,gc}(W) \textit{ and} \\
\qquad\qquad \{(\_ \mapsto c_{\text{main},c}), (\_ \mapsto c_{\text{main},d})\} \subseteq \overline{w_{\text{export}}}
\end{array} \right\}
\end{aligned}
$$

5.6 Related execution configuration

The full abstraction theorem (Theorem 1) is stated in terms of contextual equivalence. Contextual equivalence (Definition 7) plugs two components into a context and requires equitermination of the resulting executable configurations. This means that we need to lift relatedness one step further than the components, namely to the level of execution configurations. To this end, we define $\mathcal{EC}$.

Definition 20 (Related execution configuration)**.**

$$\mathcal{EC}^{\square,gc}(W) = \left\{ \begin{array}{l} \big(n, \big((ms_S, reg_S, stk, ms_{stk}), (ms_T, reg_T)\big)\big) \mid \\ \quad gc = (T_A, \text{stk\_base}) \wedge \\ \quad \exists W_M, W_R, W_{\text{pc}}.\, W = W_M \oplus W_R \oplus W_{\text{pc}} \wedge \\ \quad \big(n, ((reg_S(\text{pc}), reg_S(\text{r}_{\text{data}})), (reg_T(\text{pc}), reg_T(\text{r}_{\text{data}})))\big) \in \mathcal{E}^{\square,gc}_{\text{xjmp}}(W_{\text{pc}}) \wedge \\ \quad reg_S(\text{pc}) \neq \text{ret-ptr-code(\_)} \wedge reg_S(\text{r}_{\text{data}}) \neq \text{ret-ptr-data(\_)} \wedge \\ \quad nonExec(reg_S(\text{r}_{\text{data}})) \wedge nonExec(reg_T(\text{r}_{\text{data}})) \\ \quad ms_S, ms_{stk}, stk, ms_T :^{gc}_{n} W_M \wedge \\ \quad \big(n, (reg_S, reg_T)\big) \in \mathcal{R}^{\square,gc}(\{\text{r}_{\text{data}}\})(W_R) \end{array} \right\}$$

Definition 20 essentially says that two executable configurations are related when they are made out of related components. That is, the authority of the world must be distributed such that the code and data pointer pairs are safe for execution, i.e. the contents of the pc and r_{data} registers are related by the $\mathcal{E}_{\text{xjmp}}$ relation. Further, the two memories and the two register files should be related. This means that the executable configuration only contains words that respect memory invariants.

5.7 Full abstraction proof sketch

Using Lemma 3, we can now proceed to proving Theorem 1 (full abstraction).

Using Lemma 3 and the definitions of the logical relations, we can then prove the following two lemmas. The first is a version of the FTLR for components, stating that all components are related to themselves if they are either (1) well-formed and untrusted or (2) well-formed, reasonable, and trusted.

Lemma 4 (FTLR for components)**.** *If comp is a well-formed component in* LCM, *i.e.* $\vdash$ *comp. Then consider the same comp in* oLCM. *If either* $\text{dom}(comp.ms_{\text{code}}) \subseteq T_A$ *and comp is a reasonable component in* oLCM*; or* $\text{dom}(comp.ms_{\text{code}}) \# T_A$*, then there exists a* W *such that* $(n, (comp, comp)) \in \mathcal{C}^{\square,gc}(W)$.

Another lemma then relates the component relation and context plugging: plugging-related components into related contexts produces related execution configurations.

Lemma 5. *If* $(n, (\mathcal{C}_S, \mathcal{C}_T)) \in \mathcal{C}^{\square,gc}(W_1)$ *and* $\big(n, \big(comp_S, comp_T\big)\big) \in \mathcal{C}^{\square,gc}(W_2)$ *and* $W_1 \oplus W_2$ *is defined, then* $\mathcal{C}_S[comp_S]$ *terminates iff* $\mathcal{C}_T[comp_T]$ *terminates.*

Finally, we use these two lemmas to prove Theorem 1.

Proof (Proof of Theorem 1) Both directions of the proof are similar, so we only show the right direction. To show the LCM contextual equivalence, assume w.l.o.g a well-formed context $\mathcal{C}$ such that $\mathcal{C}[comp_1]\Downarrow$. The proof is sketched in Figure 25. By the statement of Theorem 1, we may assume that the trusted components $comp_1$ and $comp_2$ are well formed and reasonable. We prove arrow (1) in the figure by using the mentioned assumptions about $comp_1$ and $\mathcal{C}$ along with Lemma 4 and 5. Now we know that $\mathcal{C}[comp_1]\Downarrow$, so by the assumption that $comp_1$ and $comp_2$ are contextually equivalent on oLCM we get

$$\begin{tikzcd}
{} \arrow[dddd, Rightarrow, "\text{Contextual equivalence preservation}"'] & & & C_1 \approx_{\mathrm{ctx}} C_2 & & \\
& & \mathscr{C}[C_1]\Downarrow^{gc} \arrow[rr, Rightarrow, "(2)"'] & & \mathscr{C}[C_2]\Downarrow^{gc} \arrow[dd, Rightarrow, "(3)"'] & \\
& \begin{array}{c}\mathscr{C}\cong\mathscr{C}\\ C_1\cong C_1\end{array} & & & & \begin{array}{c}\mathscr{C}\cong\mathscr{C}\\ C_2\cong C_2\end{array} \\
& & \mathscr{C}[C_1]\Downarrow \arrow[uu, Rightarrow, "(1)"'] \arrow[rr, Rightarrow, "?"] & & \mathscr{C}[C_2]\Downarrow & \\
{} & & & C_1 \overset{?}{\approx}_{\mathrm{ctx}} C_2 & &
\end{tikzcd}$$

Fig. 25. Proving one direction of fully abstract compilation (contextual equivalence preservation).

$\mathscr{C}[comp_2]\Downarrow$, i.e. arrow (2) in the figure. To prove arrow (3), we again apply Lemma 4, 5; but this time, we use the assumption that $comp_2$ is well formed and reasonable and that $\mathscr{C}$ is well formed. ■

6 Discussion

With the technical results established, it is worth taking a step back and discuss our work from a more high-level perspective. In the next sections, we specifically give some more thoughts on our use of full abstraction to express WBCF and LSE and the practical usability of StkTokens.

6.1 Full abstraction

Our formulation of WBCF and LSE using a fully abstract overlay semantics has an important advantage. Imagine that you are implementing a fully abstract compiler for a high-level language, i.e. a secure compiler that enforces high-level abstractions when interacting with untrusted target-language components. Such a compiler would need to perform many things and enforce other high-level properties than just WBCF and LSE.

If such a compiler uses the StkTokens calling convention, then the security proof should not have to reprove security of StkTokens. Ideally, it should just combine security proofs for the compiler's other functionality with our results about StkTokens. We point out that our formulation enables such reuse. Specifically, the compiler could be factored into a part that targets oLCM, followed by our embedding into LCM. If the authors of the secure compiler can prove full abstraction of the first part (relying on WBCF and LSE in oLCM) and they can also prove that this first part generates well formed and reasonable components, then full abstraction of the whole compiler follows by our result and transitivity of fully abstract compilation. Perhaps other reusable components of secure compilers could be formulated similarly using some form of fully abstract overlay semantics, to obtain similar reusability of their security proofs.

A compiler is secure when it enforces the properties of high-level languages that begs the question what properties we should enforce. When it comes to fully-abstract compilation, then the answer is that all properties in the high-level language (or at least the equivalences they imply[9]) must be preserved in the target language, either by translating them to similar abstractions in the target language or by using the available target language features to enforce the source language abstraction. In our case, oLCM semantics offers a native stack with LSE and WBCF, but this abstraction does not natively exist in LCM, so StkTokens can be used to enforce it.

Perhaps a more subjective question is what kind of high-level language we would like. StkTokens ensures a standard call-return control flow, but if we want a different kind of control flow, for instance call/cc or C's `longjmp`, then we need to come up with a different enforcement scheme. Further, many high-level languages have exceptions as another form of control-flow not (yet) supported by StkTokens. This goes to show how we must consider what high-level language we want in order to answer the question of what properties we must enforce to get a fully abstract compiler.

We have not investigated support for continuations and exceptions in StkTokens thoroughly, but we expect such support could be added. For exceptions, one approach would let callers provide callees with an additional capability for exceptional returns. This second capability would be similar to the code part of the return capability pair and signed with the same seal. The callee would be able to invoke it to signal that an exception has been thrown after which the caller's code would handle the exception, either by executing an exception handler or by unwinding its stack frame and passing the exception on to its own caller. Essentially, this would mean every function would be made responsible for unwinding its own stack frame. Continuations are more complicated but could perhaps be treated using similar ideas. Alternatively, it might also be possible to have a piece of central trusted code that does the stack unravelling for all stack frames. To do this, the trusted code would need to receive a copy of all return seals from the linker.

Full abstraction of a compiler is a property that takes into account the whole source and target languages. In other words, a full abstraction proof must consider all features of a language to make sure that the features do not interact in a way that breaks language abstractions. If we have a fully abstract compiler and add a feature to the source or target language, then the new compiler is not necessarily fully abstract anymore. Full abstraction would have to be proven for the new compiler to make sure that the new feature does not break existing abstractions in the language. Our proof of full abstraction for StkTokens targets a simple capability machine that may not be able to enforce the high-level language abstractions we want, e.g. address hiding. In other words, the full-abstraction proof cannot be reused immediately. However, StkTokens is still a good candidate for the enforcement mechanism for WBCF and local-state encapsulation in a real fully abstract compiler. Generally speaking, it is worth investigating enforcement mechanisms for full abstraction in a simple setting that allows to quickly try out ideas and verify that the enforcement works.

Our full-abstraction theorem, Theorem 1 only applies to components that are reasonable and well-formed. In other words, if we were to define a compiler phase that targets oLCM, then we would also have to show that every program it generates is well formed and reasonable in order to use the full-abstraction result. Without the reasonability assumption,

[9] See Section 7 for a discussion of secure compilation properties that require preservation of more general language properties than equivalences.

StkTokens would have to enforce reasonability instead. That is, StkTokens would have to dynamically ensure that no return seals or means to obtain them are passed in calls. Such checks would impose a performance overhead and are in principle unnecessary for correct code. We have instead chosen to assume that we are given reasonable code that never exhibits the unreasonable behavior. A compiler phase where more information about the original program is available should be able to ensure the assumption with relative ease. The syntactic requirement of well-formedness is similar, except that it should be ensured by the compiler and linker together.

One challenge in full-abstraction proofs is to construct source language contexts that emulate the behavior of an arbitrary target language context. This construction is known as a back-translation (Devriese *et al.*, 2017), i.e. a translation from the target language to the source language. When we use an overlay semantics, the back-translation becomes trivial because the source and target language are syntactically the same, so the identity can be used as the back-translation. If we have native call and return instructions in the source language, then the source language would be different from the target language, and we would have to use a non-trivial back-translation. Specifically, the back-translation would need to distinguish sequences of instructions that are the translation of a call from sequences of instructions that just look like a call. With overlay semantics, this is not a concern because everything that looks like a call is interpreted as a call.

6.2 Linear capabilities in reality

We believe there are good arguments for practical applicability of StkTokens. The strong security guarantees are proven in a way that is reusable as part of a bigger proof of compiler security. Its costs are

- a constant and limited amount of checks on every boundary crossing.
- possibly a small memory overhead because stack frames must be of nonzero length

The main caveat is that we rely on the assumption that capability machines like CHERI can be extended with linear capabilities in an efficient way.

Although this assumption can only be discharged by demonstrating an actual implementation with efficiency measurements, the following notes are based on private discussions with people from the CHERI team as well as our own thoughts on the matter. We also refer to the latest CHERI ISA report, which contains a concurrently developed, unimplemented design for linear capabilities, albeit without `splice` and `split` (Watson *et al.*, 2020, Section D.7). As we understand it, the main problems for adding linear capabilities to a capability machine like CHERI are related to the move semantics for instructions like `move`, `store`, and `load`. Processor optimizations like pipelining and out-of-order execution rely on being able to accurately predict the registers and memory that an instruction will write to and read from. Our instructions are a bit clumsy from this point-of-view because, for example, `move` or `store` will zero the source register resp. memory location if the value being written is linear. A solution for this problem could be to add separate instructions for moving, storing, and loading linear registers at the cost of additional opcode space. Adding splice and split will also consume some opcode space.

Another problem is caused by the move semantics for `load` in the presence of multiple hardware threads. In this setting, zeroing out the source memory location must happen atomically to avoid race conditions where two hardware threads end up reading the same linear capability to their registers. This means that a `load` of a linear capability should behave atomically, similar to a primitive compare-and-swap instruction. This is in principle not a problem except that atomic instructions are significantly slower than a regular `load` (on the order of 10x slower or more). When using StkTokens, loads of linear capabilities happen only when a thread has stored its return data capability on the stack and loads it back from there after a return. Because the stack is a region of memory with very high thread affinity (no other hardware thread should access it, in principle), and which is accessed quite often, well-engineered caching could perhaps reduce the high overhead of atomic loads of linear capabilities. The processor could perhaps also (be told to) rely on the fact that race conditions should be impossible for loads from linear capabilities (which should be nonaliased) and just use a nonatomic load in that case.

6.3 Support for source language idioms and features

Our calling convention supports all source language features that can be expressed in oLCM. This includes relatively basic features such as optimized tail calls (see Section 4.1) and stack-allocated variable-size arrays. Some other idioms require a bit more thought, such as the control flow primitives we discussed above.

Also interesting is the idiom of passing stack references in calls, as is common in programming languages with a C-like calling convention. StkTokens supports stack references but with a couple of caveats. First of all, the stack capability is linear, so all references to the stack have to be linear. This means that the callee has to respect this linearity when using the reference. Next, like the stack capability, the stack references must be given back to the caller on return, so they can reconstruct their original stack capability (allowing them to return). Finally, the encapsulated local stack frame should be a continuous piece of memory (because it has to be addressable by a single capability: the data part of the return capability pair). Because of this, stack-allocated objects for which references are passed to callees must be allocated at the top or bottom of the caller's stack frame. An escape analysis could be used to statically determine where to put allocations and, in principle, the allocations could be reordered dynamically before a call. In summary, support for passing stack references as arguments to callees could be added to StkTokens, but this would probably require some changes in the compiler and, more importantly, would require the callee to take special care when manipulating such references. We are unsure whether it is realistic to apply this approach for existing C code.

StkTokens relies on an important property: the fact that return seals should be uniquely associated to a single call-site. If we were to use a single return seal for multiple call sites, that would allow an adversary to combine a valid return data capability with a return code capability for the wrong call-site, thus breaking WBCF guarantees. This property is currently enforced statically in the well-formedness property for components' code memory (see Section 4.2), specifically in the `C-Instr` rule. For any call instruction in trusted components' code memory, it is required that the return seal used is exclusively owned by this call-site. To guarantee that this property when the system is initialized, but continues to hold during execution, the well-formedness judgment additionally enforces a Write-Xor-Execute policy for components: writable capabilities are not allowed for code memory and executable capabilities are not allowed for data memory. Although this approach works, it has the side effect of disallowing dynamic code generation. This is unfortunate, because we believe that in principle, StkTokens is compatible with dynamic code generation, as long as the generated code respects the unique allocation of return seals to call sites.

To lift this restriction, an alternative approach would be to enforce the unique allocation of return seals to trusted call sites semantically rather than statically. More concretely, the semantic reasonability requirement for components could additionally require that during any execution, a component uses a return seal only for a single call-site. Such an approach would, we think, obviate the need for the Write-Xor-Execute policy and the static enforcement of proper use of return seals. Thus, we think dynamic code generation could be allowed without losing StkTokens guarantees.

7 Related work

In this final section, we discuss related work on securely enforcing control flow correctness and/or local state encapsulation or the linear capabilities we use to do it. We do not repeat the work we discussed in Section 1.

Capability machines originate with Dennis & Van Horn (1966), and we refer to Levy (1984) and Watson *et al.* (2015a) for an overview of previous work. The capability machine formalized in Section 2 is modelled after CHERI (Woodruff *et al.*, 2014; Watson *et al.*, 2015a). This is a recent, relatively mature capability machine that combines capabilities with a virtual memory approach in the interest of backward compatibility and gradual adoption. For simplicity, we have omitted features of CHERI that were not needed for StkTokens (e.g. local capabilities, virtual memory).

Plenty of other papers enforce WBCF 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.*, 2005a). This technique prevents certain classes of attacks by sanitizing addresses before direct and indirect jumps based on static control graph information and a protected shadow stack. Contrary to StkTokens, CFI can be implemented on commodity hardware rather than capability machines. However, its attacker model is different, and its security goals are weaker. They assume an attacker that is unable to execute code but can overwrite arbitrary data at any time during execution (to model buffer overflows). In terms of security goals, the technique does not enforce local stack encapsulation. Also, it only enforces a weak form of control flow correctness saying that jumps stay within the program's static control flow graph (Abadi *et al.*, 2005b). Such a property ignores temporal properties and seems hard to use for reasoning. There is also more and more evidence that these partial security properties are not enough to prevent realistic attacks in practice (Evans *et al.*, 2015; Carlini *et al.*, 2015; van der Veen *et al.*, 2017). A related approach relies on CPU extensions with support for micropolicies on memory Roessler & DeHon (2018). They have a similar attacker model, but propose policies which enforce stronger properties.

More closely related to our work are papers that use separate per-component stacks, a trusted stack manager, and some form of memory isolation to enforce control-flow correctness as part of a secure compilation result (Patrignani *et al.*, 2016; Juglaret *et al.*, 2016). Our work differs from theirs in that we use a different 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 per-component stacks or a trusted stack manager but a single shared stack and a decentralized calling convention based on linear capabilities. Both prove a secure compilation result from a high-level language which clearly implies a general form of control-flow correctness, but that result is not separated from the results about other aspects of their compiler.

CheriBSD applies a similar approach with separate per-component stacks and a trusted stack manager on a capability machine (Watson *et al.*, 2015a). The authors use local capabilities to prevent components from accidentally leaking their stack pointer to other components, but there is no actual capability revocation in play. They do not provide many details on this mechanism and it is, for example, not clear if and how they intend to deal with higher order interfaces (C function pointers) or stack references shared across component boundaries.

The use of return pointers or capabilities in low-level languages is in many ways similar to lambda calculus terms in continuation passing style. When applying a CPS-translation to lambda terms, well-bracketedness can also be violated if continuations are invoked more than once. Enforcing well-bracketedness for CPS terms means enforcing that continuations are only used a single time and only in the right invocation, i.e. a form of linear (or more accurately, affine) usage of continuations must be enforced. One way to enforce this is by assigning well-chosen types to CPS-translated terms and researchers have proven effectiveness for two ways of doing this, in the form of a full-abstractness result for the CPS-translation. In both cases, correct usage of continuations is enforced by assigning well-chosen types to CPS-translated terms. In the case of Laird (2005), an affine type system is used to do this, while Ahmed & Blume (2011) use a form of answer-type polymorphism instead. These results are related to ours because they enforce well-bracketedness using a form of linearity of continuations or return capabilities, but there are also quite a few differences: the static nature of the enforcement, the low-level versus high-level nature of the languages, the fact that LSE comes naturally in such a setting while we also have to enforce it. Additionally, Ahmed & Blume consider total languages and use a back-translation in two steps using an intermediate multilanguage and Laird proves full abstraction using a fully abstract denotational semantics for source and target languages.

The fact that our full abstraction result only applies to reasonable components (see Section 4) makes it related to full abstraction results for unsafe languages. In their study of compartmentalization primitives, Juglaret *et al.* (2016) discuss the property of Secure Compartmentalizing Compilation (SCC): a variant of full abstraction that applies to unsafe source languages. Essentially, they modify standard full abstraction so that preservation and reflection of contextual equivalence are only guaranteed for components that are *fully defined*, which means essentially that they do not exhibit undefined behavior in any fully defined context. If we see reasonable behavior as defined behavior, then our full abstraction result can be seen as an application of this same idea. In follow-up work, Abate *et al.*

(2018) extend this approach to components with undefined behavior by proving secure compilation guarantees for the execution until the undefined behavior happens.

Another interesting relation between our work and that of Abate *et al.* (2018) is that they consider dynamic compromise scenarios, where some components start out as trusted until they are compromised and are treated as adversarial after that. Our full abstraction result does not apply to those scenarios because it is intended to be used in the verification of a secure compiler where such scenarios are not relevant. Nevertheless, we believe our Fundamental Theorem of Logical Relations (see Section 3) does apply in such scenarios because it is step-bounded. To deal with the evolving classification of components as trusted or adversarial, we can simply apply the theorem several times with different choices of T_A and taking n small enough that the compromise of T_A has not yet happened until that point. Because the theorem only requires reasonability up to n steps, we get well-bracketedness guarantees up to the right step in the execution. This can be seen as a step-bounded version of the idea explained in (Patrignani *et al.*, 2016). If confirmed by a more careful analysis, it would be an interesting observation that modular secure compilation, combined with step-bounded reasoning, is sufficient to cover dynamic compromise scenarios without the need to build in dynamic compromise scenarios into the definition of secure compilation like Abate *et al.* (2018).

Local capabilities can be used to ensure WBCF and local-state encapsulation as demonstrated by Skorstengaard *et al.* (2018a, 2019a). Recently, Tsampas *et al.* (2019) demonstrated that an extension of local capabilities with multiple linearly ordered colors can be used to enforce the life time of stack references. Specifically, a stack reference should not be able to outlive the stack frame it points to. If StkTokens was extended with stack references, then it would also enforce reference life times. Specifically in order to return from a call, we must use the return token, i.e. the stack. The stack is linear, so if there are references to it, aside from the stack capability itself, then we cannot have a complete return token. This means that we have to splice all the stack references together with the stack capability to complete the return token in order to return. Tsampas *et al.* (2019) allow (almost) normal references that can be stored in multiple places at the same time. This means that their approach is closer to C than StkTokens. As explained in Section 1, these approaches have the downside that they require stack clearing (including unused parts) on boundary crossings. Very recently, Georges *et al.* (2021) have proposed a version of Skorstengaard *et al.*'s calling convention which succesfully avoids the need for stack clearing by relying on a newly proposed form of capabilities called uninitialized capabilities and they have proven soundness of the approach using Coq and the Iris program logic (Jung *et al.*, 2018).

In addition to the already-mentioned work on linear capabilities, Van Strydonck *et al.* (2019) have recently used them in a secure (fully abstract) compiler for a C-like language with separation logic contracts. A form of linear capabilities was also used in the SAFE machine developed within the CRASH/SAFE project (Azevedo de Amorim *et al.*, 2015, 2016). Abate *et al.* (2018) used micro-policy enforced linear return capabilities to ensure a cross-component stack discipline. Their linear capabilities were designed specifically to enforce the stack discipline but behave similarly to ours with the notable difference that their linear return pointers are destroyed in a jump.

There are other notions of secure compilation than full-abstraction (Abadi, 1998). Abate *et al.* (2019) present an overview of trace-based secure compilation properties. Full abstraction is only one, relatively weak, property in their hierarchy. It would be interesting to investigate if our compiler, i.e. the embedding function from OLCM into LCM, also satisfies some of their other properties. While our current result implies that we can prove contextual equivalences in LCM components using STKTOKENS, by proving them instead in the more well-behaved OLCM semantics, such stronger properties would imply that we can also prove robust preservation of other (hyper-)properties in a similar manner. As our back-translation works for arbitrary programs, we expect that, in addition to full abstraction, our embedding also satisfies Robust Relational Hyperproperty Preservation (RrHP, the strongest property in the hierarchy of Abate *et al.*) and that a large part of our current proof (the back-translation and the logical relation) could be reused to establish this. However, to do this, we would first need to extend our semantics with some form of traces and we have not investigated how best to do this.

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 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).

Conflicts of Interest

None.

References

Abadi, M. (1998) Protection in programming-language translations: Mobile object systems. In European Conference on Object-Oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 291–291.

Abadi, M. (1999) Protection in programming-language translations. In *Secure Internet Programming*. Springer-Verlag, pp. 19–34.

Abadi, M., Budiu, M., Erlingsson, Ú. & Ligatti, J. (2005a) Control-flow integrity. In Conference on Computer and Communications Security. ACM, pp. 340–353.

Abadi, M., Budiu, M., Erlingsson, Ú. & Ligatti, J. (2005b) A theory of secure control flow. In *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer, pp. 111–124.

Abate, C., Azevedo de Amorim, A., Blanco, R., Evans, A. N., Fachini, G., Hritcu, C., Laurent, T., Pierce, B. C., Stronati, M. & Tolmach, A. (2018) When good components go bad: Formally secure compilation despite dynamic compromise. In Computer and Communications Security. CCS'18. ACM.

Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M. & Thibault, J. (2019) Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In Computer Security Foundations Symposium. IEEE Computer Society Press.

Ahmed, A. & Blume, M. (2011) An equivalence-preserving CPS translation via multi-language semantics. In International Conference on Functional Programming. ACM, pp. 431–444.

Ahmed, A. J. (2004) *Semantics of Types for Mutable State*. PhD thesis, Princeton University.

America, P. & Rutten, J. J. M. M. (1989) Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.* **39**, 343–375.

Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B. C., Pollack, R. & Tolmach, A. (2016) A verified information-flow architecture. *J. Comput. Secur.* **24**(6), 689–734.

Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B. C., Spector-Zabusky, A. & Tolmach, A. (2015) Micro-policies: Formally verified, tag-based security monitors. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015, pp. 813–830.

Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D. & Trieu, A. (2019) Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* **4**(POPL), 7:1–7:30.

Birkedal, L. & Bizjak, A. (2014) A Taste of Categorical Logic — Tutorial Notes.

Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J. & Yang, H. (2011) Step-indexed Kripke models over recursive worlds. In POPL. ACM, pp. 119–132.

Carlini, N., Barresi, A., Payer, M., Wagner, D. & Gross, T. R. (2015) Control-flow bending: On the effectiveness of control-flow integrity. In USENIX Security. USENIX Association.

Dennis, J. B. & Van Horn, E. C. (1966) Programming semantics for multiprogrammed computations. *Commun. ACM* **9**(3), 143–155.

Devriese, D., Patrignani, M., Piessens, F. & Keuchel, S. (2017) Modular, fully-abstract compilation by approximate back-translation. *Logical Methods Comput. Sci.* **13**(10), 1–38.

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H. & Sidiroglou-Douskos, S. (2015) Control jujutsu: On the weaknesses of fine-grained control flow integrity. In Computer and Communications Security. ACM.

Georges, A. L., Guéneau, A., Van Strydonck, T., Timany, A., Trieu, A., Huyghebaert, S., Devriese, D. & Birkedal, L. (2021) Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* **5**(POPL), 6:1–6:30.

Joannou, A., Woodruff, J., Kovacsics, R., Moore, S. W., Bradbury, A., Xia, H., Watson, R. N. M., Chisnall, D., Roe, M., Davis, B., Napierala, E., Baldwin, J., Gudka, K., Neumann, P. G., Mazzinghi, A., Richardson, A., Son, S. & Markettos, A. T. (2017) Efficient tagged memory. In IEEE International Conference on Computer Design (ICCD). IEEE.

Juglaret, Y., Hriţcu, C., Azevedo de Amorim, A. & Pierce, B. C. (2016) Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In CSF. IEEE Computer Society Press.

Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L. & Dreyer, D. (2018) Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* **28**, 1–73.

Laird, J. (2005) Game semantics and linear CPS interpretation. *Theor. Comput. Sci.* **333**(1), 199–224.

Levy, H. M. (1984) *Capability-Based Computer Systems*. Digital Press.

New, M. S., Bowman, W. J. & Ahmed, A. (2016) Fully abstract compilation via universal embedding. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. ICFP 2016. ACM, pp. 103–116.

Patrignani, M., Ahmed, A. & Clarke, D. (2019) Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.* **51**, 17–30.

Patrignani, M., Devriese, D. & Piessens, F. (2016) On modular and fully abstract compilation. In Computer Security Foundations. IEEE.

Patrignani, M. & Garg, D. (2017) Secure compilation and hyperproperty preservation. In Computer Security Foundations. IEEE.

Roessler, N. & DeHon, A. (2018) Protecting the stack with metadata policies and tagged hardware. In 2018 IEEE Symposium on Security and Privacy (SP), pp. 478–495.

Scott, D. S. (1976) Data types as lattices. *SIAM J. Comput.* **5**, 522–587.

Skorstengaard, L. (2019) *Formal Reasoning about Capability Machines*. PhD thesis, Aarhus University.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2018a) Reasoning about a machine with local capabilities. In *Programming Languages and Systems*. Springer International Publishing, pp. 475–501.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2018b) *StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities*. Technical Report with Proofs and Details.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2019a) Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Trans. Program. Lang. Syst.* **42**(1), 5:1–5:53.

Skorstengaard, L., Devriese, D. & Birkedal, L. (2019b) StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.* **3**(POPL), 19:1–19:28.

Szabo, N. (1997) Formalizing and securing relationships on public networks. *First Monday* **2**(9).

Szabo, N. (2004) *Scarce Objects*.

Thamsborg, J. & Birkedal, L. (2011) A kripke logical relation for effect-based program transformations. In Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19–21, 2011.

Tsampas, S., Devriese, D. & Piessens, F. (2019) Temporal safety for stack allocated memory on capability machines. In 2019 IEEE 32nd Computer Security Foundations Symposium. IEEE Computer Society Press.

van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H. & Giuffrdia, C. (2017) The dynamics of innocent flesh on the bone: Code reuse ten years later. In ACM SIGSAC Conference on Computer and Communications Security. CCS'17. Association for Computing Machinery, pp. 1675–1689.

Van Strydonck, T., Piessens, F. & Devriese, D. (2019) Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.* **ICFP**. accepted.

Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., Davis, B., Eisen, L., Filardo, N. W., Grisenthwaite, R., Joannou, A., Laurie, B., Markettos, A. T., Moore, S. W., Murdoch, S. J., Nienhuis, K., Norton, R., Richardson, A., Rugg, P., Sewell, P., Son, S. & Xia, H. (2020) *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory.

Watson, R. N. M., Neumann, P. G., Woodruff, J., Roe, M., Anderson, J., Chisnall, D., Davis, B., Joannou, A., Laurie, B., Moore, S. W., Murdoch, S. J., Norton, R. & Son, S. (2015b) *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Technical report UCAM-CL-TR-876. University of Cambridge, Computer Laboratory.

Watson, R. N. M., Norton, R. M., Woodruff, J., Moore, S. W., Neumann, P. G., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Roe, M., Dave, N. H., Gudka, K., Joannou, A., Markettos, A. T., Maste, E., Murdoch, S. J., Rothwell, C., Son, S. D. & Vadera, M. (2016) Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro* **36**(5).

Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S. & Vadera, M. (2015a) CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In IEEE Symposium on Security and Privacy. IEEE, pp. 20–37.

Watson, R. N., Neumann, P. G., Woodruff, J., Anderson, J., Anderson, R., Dave, N., Laurie, B., Moore, S. W., Murdoch, S. J., Paeps, P., Roe, M. & Saidi, H. (2012) CHERI: A research platform deconflating hardware virtualization and protection. In Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE).

Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R. & Roe, M. (2014) The CHERI capability model: Revisiting RISC in an age of risk. In International Symposium on Computer Architecuture. IEEE, pp. 457–468.

Appendix A LCM instruction interpretation

In this section, we present the interpretation of the LCM instructions left out of Figures 3 and 4.

$i \in$ Instr	$[\![i]\!]\ (\Phi)$	Conditions
`getb` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_,\_), b, \_, \_)$ or$\Phi(r_2) = \mathrm{seal}(b, _, _)$, then $w = b$ and otherwise $w = -1$
`gete` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_,\_), \_, e, \_)$ or $\Phi(r_2) = \mathrm{seal}(\_, e, \_)$, then $w = e$ and otherwise $w = -1$
`gettype` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto w])$	$w = \mathit{encodeType}(\Phi(r_2))$[10]
`getl` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto w])$	If $\mathit{isLinear}(\Phi(r_2))$, then $w = \mathit{encodeLin}(\mathrm{linear})$, otherwise $w = \mathit{encodeLin}(\mathrm{normal})$
`getp` $r_1\ r_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto w])$	If $\Phi(r_2) = ((p,\_), \_, \_, \_)$, then $w = \mathit{encodePerm}(p)$ and otherwise $w = -1$
`jnz` $r\ rn$	$\Phi[\mathit{reg}.r \mapsto w][\mathrm{pc} \mapsto \Phi(r)]$	$\mathit{nonZero}(\Phi(rn))$ and $w = \mathit{linClear}(\Phi(r))$
`jnz` $r\ rn$	$\mathit{updPc}(\Phi)$	If not $\mathit{nonZero}(\Phi(rn))$
`plus` $r\ rn_1\ rn_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto n_1 + n_2])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$
`minus` $r\ rn_1\ rn_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto n_1 - n_2])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$
`lt` $r\ rn_1\ rn_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto 1])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$ and $n_1 < n_2$
`lt` $r\ rn_1\ rn_2$	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto 0])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$ and $n_1 \not< n_2$
`seta2b` r	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto c])$	$r \neq \mathrm{pc}$, $\Phi(r) = ((p,l), b, e, \_)$, and $c = ((p,l), b, e, b)$
`seta2b` r	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto c])$	$r \neq \mathrm{pc}$, $\Phi(r) = \mathrm{seal}(\sigma_b, \sigma_e, \_)$, and $c = \mathrm{seal}(\sigma_b, \sigma_e, \sigma_b)$
`restrict` $r\ rn$	$\mathit{updPc}(\Phi[\mathit{reg}.r \mapsto c])$	If $\Phi(r) = ((p,l), b, e, a)$ and $\Phi(rn) = n$ and $\mathit{decodePerm}(n) \leq p$ and $c = ((\mathit{decodePerm}(n), l), b, e, a)$
`split` $r_1\ r_2\ r_3\ rn$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto c_1][\mathit{reg}.r_2 \mapsto c_2])$	$\Phi(r_3) = \mathrm{seal}(\sigma_b, \sigma_e, \sigma)$ and $\Phi(rn) = \sigma_n \in \mathbb{N}$ and $\sigma_b \leq \sigma_n < \sigma_e$ and $c_1 = \mathrm{seal}(\sigma_b, \sigma_n, \sigma)$ and $c_2 = \mathrm{seal}(\sigma_n + 1, \sigma_e, \sigma)$
`splice` $r_1\ r_2\ r_3$	$\mathit{updPc}(\Phi[\mathit{reg}.r_1 \mapsto c])$	$\Phi(r_2) = \mathrm{seal}(\sigma_b, \sigma_n, \_)$ and $\Phi(r_3) = \mathrm{seal}(\sigma_n + 1, \sigma_e, \sigma)$ and $\sigma_b \leq \sigma_n < \sigma_e$ and $c = \mathrm{seal}(\sigma_b, \sigma_e, \sigma)$

[10]The *encodeType*() function differs for the two machines. Depending on the machine, the appropriate function is used.

Where *nonZero* is defined as follows

$$nonZero(w) \overset{def}{=} \begin{cases} \bot & w \in \mathbb{Z} \wedge w = 0 \\ \top & \text{otherwise} \end{cases}$$

Appendix B oLCM instruction interpretation

In this section, we present the interpretation of the oLCM instructions left out of Figures 11 and 12. For the instructions that only change slightly on oLCM compared to LCM, we include the oLCM specific things in blue and the things both have in common in black.

$i \in$ Instr	$[\![i]\!]\,(\Phi)$	Conditions
`store` $r_1\ r_2$	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[mem.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = ((p, \_), b, e, a)$ and $p \in \{\text{RWX}, \text{RW}\}$ and $b \le a \le e$ and $w_2 = linClear(\Phi(r_2))$ and $a \in \text{dom}(\Phi.mem)$
`load` $r_1\ r_2$	$updPc(\Phi[reg.r_1 \mapsto w_1]$ $[mem.a \mapsto w_a])$	$\Phi(r_2) = \text{stack-ptr}(p, b, e, a)$ and $b \le a \le e$ and $p \in \{\text{RWX}, \text{RW}, \text{RX}, \text{R}\}$ and $a \in \text{dom}(\Phi.ms_{stk})$ and $w_1 = \Phi.ms_{stk}(a)$ and $isLinear(w_1) \Rightarrow p \in \{\text{RWX}, \text{RW}\}$ and $w_a = linClear(w_1)$
`geta` $r_1\ r_2$	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, \_, a)$, then $w = a$ and otherwise $w = -1$
`getb` $r_1\ r_2$	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), b, \_, \_)$, then $w = b$ and otherwise $w = -1$
`gete` $r_1\ r_2$	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, e, \_)$, then $w = e$ and otherwise $w = -1$
`getp` $r_1\ r_2$	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((p, \_), \_, \_, \_)$, then $w = encodePerm(p)$ and otherwise $w = -1$
`seta2b` r	$updPc(\Phi[reg.r \mapsto c])$	$r \ne \text{pc}$, $\Phi(r) = \text{stack-ptr}(p, b, e, \_)$, and $c = \text{stack-ptr}(p, b, e, b)$
`restrict` $r\ rn$	$updPc(\Phi[reg.r \mapsto c])$	If $\Phi(r) = \text{stack-ptr}(p, b, e, a)$ and $\Phi(rn) = n$ and $decodePerm(n) \le p$ and $c = \text{stack-ptr}(decodePerm(n), lin),)$ b, e, a

`splice` $r_1\ r_2\ r_3$	$updPc(\Phi[reg.r_2 \mapsto 0]$ $[reg.r_3 \mapsto 0]$ $[reg.r_1 \mapsto c])$	$\Phi(r_2) = \text{stack-ptr}(p, b, n, \_)$ and $\Phi(r_3) = \text{stack-ptr}(p, n+1, e, a)$ and $b \leq n < e$ and $c = \text{stack-ptr}(p, b, e, a)$
`split` $r_1\ r_2\ r_3\ rn$	$updPc(\Phi[reg.r_3 \mapsto 0]$ $[reg.r_1 \mapsto c_1]$ $[reg.r_2 \mapsto c_2])$	$\Phi(r_3) = \text{stack-ptr}(p, b, e, a)$ and $\Phi(rn) = n \in \mathbb{N}$ and $b \leq n < e$ and $c_1 = \text{stack-ptr}(p, b, n, a)$ and $c_2 = \text{stack-ptr}(p, n+1, e, a)$

Appendix C World definitions

Definition 21. *Given* $W_{\text{free}} \in \text{World}_{\text{free\_stack}}$

$$\lfloor W_{\text{free}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{free}}(r).\text{v} \in S \\ \bot \end{cases}$$

Definition 22 (Private stack sub-world erasure)**.** *Given* $W_{\text{priv}} \in \text{World}_{\text{call\_stack}}$

$$\lfloor W_{\text{priv}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{priv}}(r).\text{region.v} \in S \\ \bot \end{cases}$$

Definition 23 (Heap sub-world erasure)**.** *Given* $W_{\text{heap}} \in \text{World}_{\text{heap}}$

$$\lfloor W_{\text{heap}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{heap}}(r).\text{v} \in S \\ \bot \end{cases}$$

Definition 24 (World erasure)**.** *Given world* $(W_{\text{heap}}, W_{\text{priv}}, W_{\text{free}})$ *define world erasure as*

$$\lfloor (W_{\text{heap}}, W_{\text{priv}}, W_{\text{free}}) \rfloor_{\{S\}} = \left(\lfloor W_{\text{heap}} \rfloor_{\{S\}}, \lfloor W_{\text{priv}} \rfloor_{\{S\}}, \lfloor W_{\text{free}} \rfloor_{\{S\}} \right)$$

$$\lfloor W_{\text{heap}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{heap}}(r).\text{v} \in S \\ \bot \end{cases}$$

$$\lfloor W_{\text{priv}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{priv}}(r).\text{region.v} \in S \\ \bot \end{cases}$$

$$\lfloor W_{\text{free}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{free}}(r).\text{v} \in S \\ \bot \end{cases}$$

The active function takes a world and filters away all the revoked regions, so

$$active(W) = \lfloor W \rfloor_{\{\text{shadow,spatial,shared}\}}$$

Appendix D Standard c.o.f.e. definitions

Definition 25 (Product c.o.f.e.)**.** *Given two c.o.f.e.'s* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ *and* $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ *define the product c.o.f.e. as* $\left(X \times Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ *where the equivalence family is defined as* *for* $(x, y), (x', y') \in X \times Y$

$$(x, y) \overset{n}{=} (x', y') \textit{ iff } x \overset{n}{=} x' \wedge y \overset{n}{=} y'$$

Definition 26 (Product preordered c.o.f.e.). *Given two c.o.f.e.'s* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ *and* $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}, \sqsupseteq_y\right)$*, define the product preordered c.o.f.e. as*

$$\left(X \times Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the preorder $\sqsupseteq$ *distributes to the underlying preorder, i.e.*

$$\textit{for } (x, y), (x', y') \in X \times Y, \quad (x', y') \sqsupseteq (x, y) \textit{ iff } x' \sqsupseteq_X x \wedge y' \sqsupseteq_Y y$$

and the family of equivalences distributes to the underlying families of equivalences, i.e.

$$\textit{for } (x, y), (x', y') \in X \times Y, \quad (x, y) \overset{n}{=} (x', y') \textit{ iff } x \overset{n}{=}_X x' \wedge y \overset{n}{=}_Y y'$$

Definition 27 (Union preordered c.o.f.e.). *Given two c.o.f.e.'s* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ *and* $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}, \sqsupseteq_y\right)$*, define the product preordered c.o.f.e. as*

$$\left(X \cup Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the preorder $\sqsupseteq$ *distributes to the underlying preorder, i.e.*

$$\textit{for } z, z' \in X \cup Y, \quad z' \sqsupseteq z \textit{ iff } \begin{cases} z, z' \in X \wedge z' \sqsupseteq_x z \vee \\ z, z' \in Y \wedge z' \sqsupseteq_Y z \end{cases}$$

and the family of equivalences distributes to the underlying families of equivalences, i.e.

$$\textit{for } z, z' \in X \cup Y, \quad z \overset{n}{=} z' \textit{ iff } \begin{cases} z, z' \in X \wedge z \overset{n}{=}_x z' \vee \\ z, z' \in Y \wedge z \overset{n}{=}_Y z' \end{cases}$$

Definition 28 (▶ preordered c.o.f.e.). *Given a preordered c.o.f.e* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ *define*

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

where

$$x \overset{n}{=}_{\blacktriangleright} x' \textit{ iff } \begin{cases} n = 0 \vee \\ x \overset{n-1}{=}_X x' \end{cases}$$