StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities

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

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.

CCS Concepts: • **Security and privacy** → *Formal security models*; *Systems security*; • **Theory of computation** → *Program reasoning*;

Additional Key Words and Phrases: fully abstract compilation, secure compilation, capability machines, linear capabilities, well-bracketed control flow, stack frame encapsulation, fully abstract overlay semantics

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

1 INTRODUCTION

Secure compilation is an active topic of research (e.g. [Abate et al. 2018; Devriese et al. 2017; Juglaret et al. 2016; New et al. 2016; Patrignani and Garg 2017]), but a real secure compiler is yet to be made. 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, but not in the assembly languages they are compiled to, are well-bracketed control flow and encapsulation of local state.

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. At the assembly level, this property does not hold immediately. Invoked functions get direct 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, a function may invoke return pointers from other stack frames than its own: either frames higher in the call stack or ones that no longer exist as they have already returned.

Authors' addresses: Lau Skorstengaard, Aarhus University, Denmark, lask@cs.au.dk; Dominique Devriese, Vrije Universiteit Brussel, Belgium, dominique.devriese@vub.be; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2019/1-ART19
https://doi.org/10.1145/3290332

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 immediately. 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 that can be used to prevent untrusted code from misbehaving without imposing too much overhead on well-behaved code. The virtual-memory based security primitives 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. 2015b]. These processors use tagged memory to enforce a strict separation between integers and *capabilities*: pointers that carry authority. Capabilities come in different flavours. 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. Each of the two capabilities are sealed with a common seal which make them opaque. The hardware transparently unseals the pair upon invocation [Watson et al. 2015a, 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. 2015a], 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.

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. 2018]. The hardware prevents these capabilities from being duplicated. We propose to make

stack and return pointers linear and require components to hand them out in cross-component invocations and to return them in returns. The non-duplicability of linear capabilities together with some security checks allow us to guarantee WBCF and LSE without large overhead on boundary crossings and in particular without the need for clearing large blocks of memory.

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 a second operational semantics for our target language with a native well-bracketed call stack and primitive ways to do calls and returns. This 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 iff 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 paper is accompanied by a technical report [Skorstengaard et al. 2018b] with the elided details and proof.

2 A CAPABILITY MACHINE WITH SEALING AND LINEAR CAPABILITIES

In this section, we introduce a simple but representative capability machine with linear capabilities, that we call LCM (Linear Capability Machine). LCM is mainly inspired by CHERI [Watson et al.

2015b] with linear capabilities as the main addition. For simplicity, LCM assumes an infinite address space and unbounded integers.

The concept of a capability is the cornerstone of any capability machine. In its simplest form, a capability is a permission and a range of authority. The permission dictates the operations the capability can be used for, and the range of authority specifies the range of memory it can act upon. The capabilities on LCM are of the form $((perm, lin), base, end, addr)$ (defined in Figure 2 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 1. In addition to the permission and range, capabilities also have a current address *addr* and a linearity *lin*. 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 non-duplicability of linear capabilities means that a linear capability cannot become aliased if it wasn't to begin with.

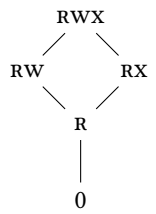


Fig. 1. Permission hierarchy

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 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. 2016, 2015b]. 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. Syntactically, the sealed capability is represented as $\text{sealed}(\sigma, sc)$ where *sc* is a sealable and σ is the seal used to seal the capability. In order to seal a sealable with a seal σ, it is necessary to have the right to do so. The right to make sealed capabilities is represented by a sets of seals $\text{seal}(\sigma_{base}, \sigma_{end}, \sigma_{current})$. A set of seals is a capability that represents the authority to seal sealables with seals in the range $[\sigma_{base}, \sigma_{end}]$. In spirit of memory capabilities, a set of seals has an active seal $\sigma_{current}$ that is selected for use in the next seal operation. As we will see later, sealed capabilities can be unsealed with 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 of the pair) available to the invoked code. The combination of sealed capabilities and `xjmp` gives (1) and (2).

Words on LCM are capabilities and data (represented by integers ($\mathbb{Z}$)). We assume a finite set of register names RegName containing at least pc, r_{rdata}, r_{rcode}, r_{stk}, r_{data}, r_{t1}, and r_{t2}, and we define register files as functions from register names to words. Complete memories map all addresses to words and memory segments map some addresses to words (i.e. 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`),

$$
\begin{array}{rrcl}
\text{a, base} \in & \text{Addr} & \stackrel{def}{=} & \mathbb{N} \\
\text{end} \in & & & \text{Addr} \cup \{\infty\} \\
\text{perm} \in & \text{Perm} & ::= & \textsc{rwx} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{r} \mid 0 \\
sc \in & \text{Sealables} & ::= & ((\text{perm}, \text{l}), \text{base}, \text{end}, \text{a}) \mid \text{seal}(\sigma_{\text{base}}, \sigma_{\text{end}}, \sigma) \\
c \in & \text{Cap} & ::= & \text{Sealables} \mid \text{sealed}(\sigma, sc) \\
r \in & \text{RegName} & ::= & \text{pc} \mid \text{r}_{\text{rdata}} \mid \text{r}_{\text{rcode}} \mid \text{r}_{\text{stk}} \mid \text{r}_{\text{data}} \mid \text{r}_{\text{t1}} \mid \text{r}_{\text{t2}} \mid \ldots \\
reg \in & \text{RegFile} & \stackrel{def}{=} & \text{RegName} \to \text{Word} \\
ms \in & \text{MemSeg} & \stackrel{def}{=} & \text{Addr} \rightharpoonup \text{Word} \\
& \text{Conf} & \stackrel{def}{=} & \text{ExecConf} \cup \{\text{failed}\} \cup \{\text{halted}\}
\end{array}
\qquad
\begin{array}{rrcl}
\sigma_{\text{base}}, \sigma \in & \text{Seal} & \stackrel{def}{=} & \mathbb{N} \\
\sigma_{\text{end}} \in & & & \text{Seal} \cup \{\infty\} \\
\text{l} ::= & & & \text{linear} \mid \text{normal} \\
w \in & \text{Word} & \stackrel{def}{=} & \mathbb{Z} \uplus \text{Cap} \\
mem \in & \text{Memory} & \stackrel{def}{=} & \text{Addr} \to \text{Word} \\
\Phi \in & \text{ExecConf} & \stackrel{def}{=} & \text{Memory} \times \text{RegFile}
\end{array}
$$

$$
\begin{array}{rcl}
r \in \text{RegisterName} & & rn ::= r \mid \mathbb{N} \\
\text{Instr} ::= & \multicolumn{2}{l}{\texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \texttt{minus}\ r\ rn\ rn \mid} \\
& \multicolumn{2}{l}{\texttt{lt}\ r\ rn\ rn \mid \texttt{gettype}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid} \\
& \multicolumn{2}{l}{\texttt{cca}\ r\ nrn \mid \texttt{seta2b}\ r \mid \texttt{restrict}\ r\ rn \mid \texttt{cseal}\ r\ r \mid \texttt{xjmp}\ r\ r \mid \texttt{split}\ r\ r\ r\ rn \mid} \\
& \multicolumn{2}{l}{\texttt{splice}\ r\ r\ r \mid \texttt{fail} \mid \texttt{halt}}
\end{array}
$$

Fig. 2. The syntax of our capability machine with seals and linear capabilities.

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 reducing the range of authority of a capability. Because LCM has linear capabilities, the instruction for this splits the capability in two rather than reducing the range of authority (`split`). 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 Figure 3. 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 Figure 3. 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 non-duplicable linear capabilities. When a linear capability is moved, the source register must be cleared, so the capability is not duplicated. 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

$$\frac{\Phi(\mathrm{pc}) = ((p, \_), b, e, a) \qquad b \le a \le e \qquad p \in \{\textsc{rwx}, \textsc{rx}\}}{\Phi \to [\![\mathit{decode}(\Phi.\mathit{mem}(a))]\!](\Phi)} \qquad \frac{\forall \Phi' \neq \mathrm{failed}.\ \Phi \not\rightsquigarrow \Phi'}{\Phi \to \mathrm{failed}}$$

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

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

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

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

Fig. 3. An excerpt of the operational semantics of LCM

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 either write or read depending on the operation, 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 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.

The operational side of the sealing in LCM consists of two instructions: `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 r_{data} register, but only if they are sealed with the same seal and the data capability (the one placed in r_{data}) is non-executable. 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 capability, 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. 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 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 what a capability can do before it is passed away. 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 non-duplicable. 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 seal sets. 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 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 introduce

$$s \in \text{Symbol} \qquad \overline{\text{import}} ::= a \hookleftarrow s \qquad \text{export} ::= s \mapsto w$$
$$comp_0 ::= (ms_{\text{code}}, ms_{\text{data}}, \overline{\text{import}}, \overline{\text{export}}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}})$$
$$comp ::= comp_0 \mid (comp_0, c_{\text{main},c}, c_{\text{main},d})$$

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

$$\frac{comp_0'' = comp_0 \bowtie comp_0'}{(comp_0'', c_{\text{main},c}, c_{\text{main},d}) = comp_0 \bowtie (comp_0', c_{\text{main},c}, c_{\text{main},d}) = (comp_0, c_{\text{main},c}, c_{\text{main},d}) \bowtie comp_0'}$$

Fig. 4. Components and linking of components.

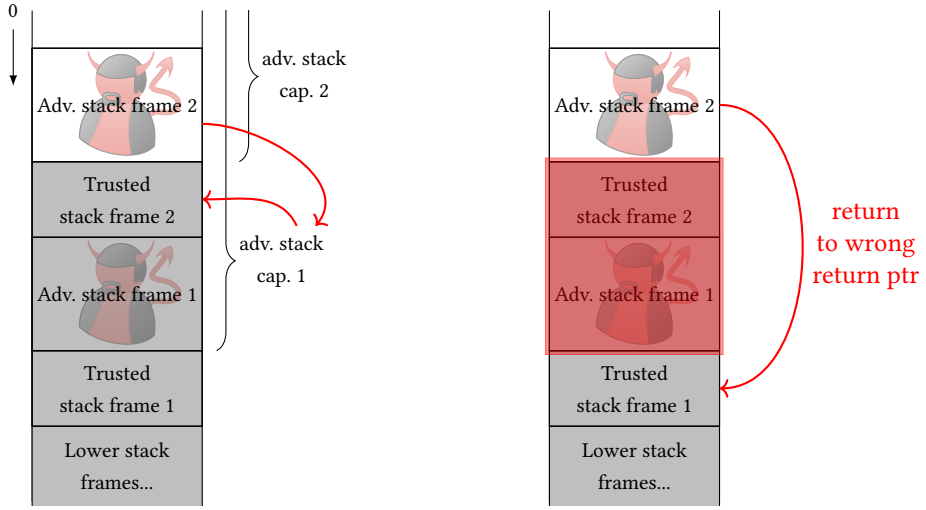
notions of components and programs from which we construct executable configurations. A component (defined in Figure 4) is basically a program with entry points in the form of imports that need to be linked. It has exports that can satisfy the imports of other components. A base component $comp_0$ consists of a code memory segment, a data memory segment, a list of imported symbols, a list of exported symbols, two lists specifying the available seals[1], 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. The exports are words each associated with a symbol. A component is either a library component (without a main entry point) or an incomplete program with a main 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 is an incomplete program with a main. 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 updated to be the exports of the two components.

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

Definition 1 (Programs and Contexts). *A program is a component $(comp_0, c_{\text{main},c}, c_{\text{main},d})$ 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 initialised to create an executable configuration, is discussed in Section 4. Some simplifications have been made in this presentation of LCM. See Skorstengaard et al. [2018b] for details.

[1]We will return to the seals in Section 4.



(a) An adversary uses a previous stack frame's stack pointer.

(b) An adversary jumps to a previous stack frame's stack pointer.

Fig. 5. Possible ways to abuse stack and return capabilities.

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. it does not cover the caller's stack frame. Return pointers, on the other hand, are replaced by a pair of sealed return capabilities. They form an opaque closure that the callee can only jump to, and the caller's data becomes 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[2]) can store the provided capabilities on the heap in order to use them later. Figure 5 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 alternating between ours and theirs. The figure on the left (Figure 5a) 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 5b), we have given the adversary two pairs of sealed return capabilities, one in each of the

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

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.

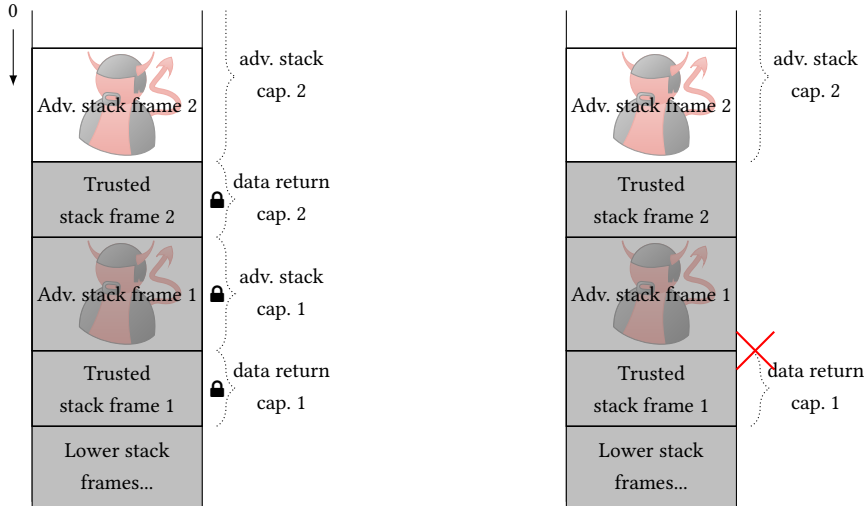
As the examples illustrate, this naive use of standard memory and object capabilities does not provide sufficient guarantees to enforce LSE and WBCF. The problem is essentially that the stack and return pointers that a callee receives from a caller remain in effect after their intended lifetime: either when the callee has already returned or when they have themselves invoked other code. Linear capabilities offer a form of revocation[3] 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 would need the stack capability produced by the caller in order to access their local data. The caller gives this capability to the adversary only in a sealed form, rendering it opaque and unusable. This is illustrated in Figure 6a and prevents the issue illustrated in Figure 5a.

In a traditional calling convention with a single stack, the stack serves as a call stack keeping track of the order calls where 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 returning from an arbitrary call on the call stack. This is exactly what the adversary does in Figure 5b 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 provide 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 capability back on return, they need to check that this token is actually spendable, i.e. check whether their stack frame is on top of the call stack or not. They do this by attempting to restore the stack capability from before the call by splicing the return token with the stack capability for the local stack frame which at this point has been unsealed again. 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 6b, it prevents the issue in Figure 5b 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 non-empty. We cannot allow empty stack frames on the call stack, because then it would be impossible to tell whether the topmost non-empty stack frame has an empty stack frame on top of it. Non-empty 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. This means that a caller using StkTokens needs to take care that their stack frame is non-empty in order

[3]Revocation in the sense that if we hand out a linear capability and later get it back, then the receiver no longer has it or a copy of it as it is non-duplicable.



(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.

Fig. 6. Abuse of stack and return capabilities prevention.

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 non-empty: They need a bit 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 callee may even 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 7: In Figure 7a, 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 7b). The trusted caller calls the adversary back using the precautions we have described so far (Figure 7c). At this point (Figure 7b), 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 downwards, 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, which was the base address for the return token they gave to the callee. In the scenario we just sketched, the caller would be alerted to the attempt to break WBCF when the base address check of the return token fails in Figure 7d.

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 never be leaked to adversarial code as this would allow them to unseal the local stack frame of a caller breaking 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 capability 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 vs 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 are going to assume that both trusted and adversarial components are initially syntactically well-formed (described in more detail in Section 4.2) which basically just restrict adversarial components to not break machine guarantees initially (e.g. no aliases for linear capabilities or access to seals of other components). This means that any component can ensure WBCF and LSE by employing StkTokens.

To summarise, 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 non-empty.
(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 for the data part of the return pair.
(4) Create code return capability on call: Seal the old program pointer.
(5) Reasonable use of seals: Return seals are only used to seal old program pointers, every return seal is only used for one call site, and they are not leaked.

Item 1-4 are captured by the code in Figure 8 , 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 well-bracketed control flow (WBCF) and local state encapsulation (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.

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.

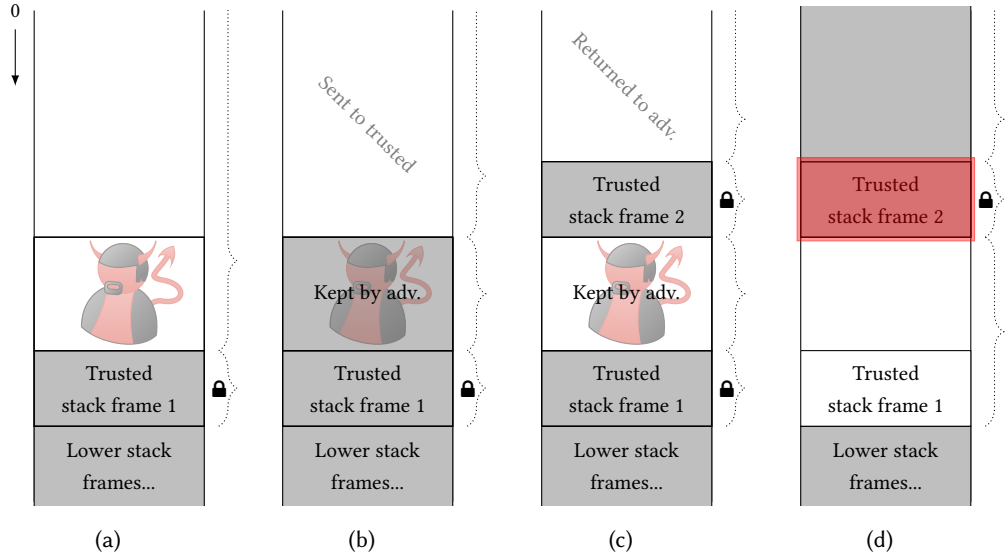


Fig. 7. Partial return token used to return out of order.

```
// 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 r_1 r_2
// 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 :  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. 8. The instructions for a $\texttt{call}^{off_{pc}, off_\sigma}\ r_1\ r_2$ with off_{pc} the offset from line 1 of the call to the set of seals it uses and off_σ the offset in the set of seals to the call seal. 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 non-empty stack. Line 7: −5, offset from line 6 (where pc was copied into r_{t1}) to line 1. Line 12: 5, offset to the return address. Line 19: 5, offset to fail. Line 21: offset to address after fail.

(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 multi-threading 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 formalise 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 run-time 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-in to 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 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 iff 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 well-behaved components that respect the basic rules of the calling convention. For example, the system should be set up such 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

judgement 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 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 9). 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 claimed by a call and thus can be used at this point of time. 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 the code and data return pointers. The data return pointer corresponds to some stack pointer (which in turn corresponds to a linear capability), and the code return pointer corresponds to some capability with read-execute permission. Syntactically, the return capabilities contain just enough information to reconstruct what they correspond to on the underlying machine. On oLCM, return pointers are generated by calls from the capabilities they correspond to on LCM, and they are turned back to the capabilities they correspond to upon return.

The opaque nature of the return pointers is reflected in the interpretation of the instructions common to both LCM and oLCM 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 10. 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 the overlay semantics has on the underlying machine. In other words, 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. Only instructions at these addresses can be interpreted as the oLCM native call and push frames to the call stack which guarantees LSE and WBCF. The constant T_A is a parameter of the oLCM step relation. Similarly, 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.

$$
\begin{array}{rcl}
\text{Sealables} & ::= & \text{Sealables} \mid \text{stack-ptr(perm, base, end, a)} \mid \\
& & \text{ret-ptr-data(base, end)} \mid \text{ret-ptr-code(base, end, a)} \\
\text{StackFrame} & \stackrel{def}{=} & \text{Addr} \times \text{MemSeg} \qquad\qquad \text{Stack} \stackrel{def}{=} \text{StackFrame}^* \\
\text{ExecConf} & \stackrel{def}{=} & \text{Memory} \times \text{RegFile} \times \text{Stack} \times \text{MemSeg} \\
\text{Instr} & ::= & \text{Instr} \mid \texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r\ r \qquad\qquad off_{\text{pc}}, off_{\sigma} \in \mathbb{N}
\end{array}
$$

Fig. 9. 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.

Apart from the step relation of LCM, oLCM has one overlay step that takes precedence over the others. This step is shown in Figure 10, and it is different from the others in the sense that it interprets a sequence of instructions rather than one. The sequence of instructions have to correspond to a call, i.e. the instructions in Figure 8 ($\texttt{call}_i^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2$ corresponds to the i'th 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 $\texttt{call}^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2$ is also shown in Figure 10 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 r_{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 sealed return capabilities. The return capabilities are sealed with the seal designated for the call.

The return capabilities, ret-ptr-code and ret-ptr-data are sealed and 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 doesn't 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 call. 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 (well-bracketed control flow) 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-Formedness and Reasonability

The judgement $T_A \vdash comp$ specifies what components are well formed, i.e. satisfy the initial syntactic requirements necessary to be able to rely on system guarantees. We elide the details of the judgement here and describe the main points: For components with a main pair, the main pair must come from the exports and the remainder of the component must be well-formed. As a reminder, a base component looks like this: (ms_{code}, ms_{data}, $\overline{import}$, $\overline{export}$, $\overline{\sigma_{\text{ret}}}$, $\overline{\sigma_{\text{clos}}}$, A_{linear}). The return seals $\overline{\sigma_{\text{ret}}}$ are the seals supposed to be used to seal return pointers and the closure seals $\overline{\sigma_{\text{clos}}}$ all the other seals in a component. If a component is adversarial, i.e. the domain of the code memory ms_{code} is disjoint from the set of trusted addresses T_A, then there should be no return seals. The code memory may contain sets of seals but only for the seals in $\overline{\sigma_{\text{ret}}}$ and $\overline{\sigma_{\text{clos}}}$. Other than that, code memory only contains instructions in the form of integers. When a sequence of instructions in the code memory corresponds to a call (Figure 8), the call must have access to the return seal it specifies. The

$$\frac{\Phi(\text{pc}) = ((p, \_), b, e, a) \qquad [a, a + \text{call\_len} - 1] \subseteq T_A \qquad [a, a + \text{call\_len} - 1] \subseteq [b, e] \\ p \in \{\text{RWX}, \text{RX}\} \qquad \Phi.mem(a, \ldots, \text{a} + \text{call\_len} - 1) = \texttt{call}_0^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2 \cdots \texttt{call}_{\text{call\_len}-1}^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2}{\Phi \rightarrow^{T_A, \text{stk\_base}} [\![\texttt{call}^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2]\!](\Phi)}$$

$i \in$ 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 $p \in \{\text{RWX}, \text{RW}\}$ and $b \le a \le e$ and $w_2 = linClear(\Phi(r_2))$ and $a \in \text{dom}(ms_{stk})$
`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 $w = \text{stack-ptr}(p, b, e, a + n)$
$\texttt{call}^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2$	$xjmpRes(c_1, c_2, \left(\begin{array}{l}\Phi[reg.r_1, r_2 \mapsto w_1, w_2] \\ [reg.\text{r}_{\text{rcode}} \mapsto s_c] \\ [reg.\text{r}_{\text{rdata}} \mapsto s_d] \\ [reg.\text{r}_{\text{stk}} \mapsto c_{stk}] \\ [ms_{stk} \mapsto ms_{stk,rest}] \\ [stk \mapsto stk']\end{array}\right))$	$ms_{stk,local}, c_{local}, ms_{stk,rest}, c_{stk} =$ $splitStack(\Phi.reg(\text{r}_{\text{stk}}), \Phi.ms_{stk})$ and $opc, c_{opc} = setupOpc(\Phi.reg(\text{pc}))$ and $stk' = (opc, ms_{stk,local}) :: \Phi.stk$ and $\sigma = getCallSeal($ $\Phi.reg(\text{pc}), \Phi.mem, off_{\text{pc}}, off_\sigma)$ and $s_c, s_d = sealReturnPair(\sigma, c_{opc}, c_{local})$ and $w_1, w_2 = linClear(\Phi.reg(r_1, r_2))$ and $\Phi.reg(r_1, r_2) = \text{sealed}(\sigma', c_1), \text{sealed}(\sigma', c_2)$
…		
_	failed	otherwise

$xjmpRes(c_1, c_2, \Phi) =$

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

Fig. 10. An excerpt of the operational semantics of oLCM (some details omitted). Auxiliary definitions are found in Figure 11.

return seal also needs to be unique to that call; that is, no other call can specify the same seal as its return seal. The data memory ms_{data} may contain data, capabilities, and sealed capabilities. The capabilities in the data memory can only have authority over the data memory itself. This allows components to have initial data structures. In order to respect Write-XOR-Execute, the capabilities in data memory cannot have execute permission. The capabilities also need to respect linearity which means that linear capabilities cannot be aliased by any other capability, and they must be for a range of the predetermined linear addresses A_{linear}. The sealed capabilities must be sealed with a closure seal, and the sealable can be anything that is allowed to reside in data memory. The exports

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

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

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

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

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

Fig. 11. Auxiliary definitions used in the operational semantics of oLCM.

$\overline{export}$ can be anything non-linear allowed to reside in data memory or a sealed capability for the code memory sealed with one of the closure seals.

The static guarantees given by $T_A \vdash comp$ makes sure that components initially don't undermine the security measures needed for StkTokens, but it does not prevent a component from doing something silly during execution that undermines StkTokens. In order for StkTokens to provide guarantees for a component, we expect it to not shoot itself in the foot and perform certain necessary checks not captured by the call code (Figure 8). 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 often would be redundant. (2) It uses the return seals only for calls and the closure seals in an appropriate way which means that they should only be used to seal executable capabilities for code that behaves reasonably or non-executable things that do not undermine the security mechanisms StkTokens relies on. (3) It does not leak return and closure seals or means to retrieve them. This means that sets of seals 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 does not store return and closure seals or means to get them. 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 to begin with. We elide the details here and refer to Skorstengaard et al. [2018b]

In our result, we assume that adversarial components are well-formed, but not necessarily reasonable. The well-formedness 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. 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

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

Fig. 12. The judgement $prog \rightsquigarrow \Phi$, which defines the initial execution configuration Φ for executing a program *prog*.

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.

4.3 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 judgement $comp \rightsquigarrow \Phi$ in Figure 12 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.

Definition 2 (Plugging a component into a context)**.** *When comp′ is a context for component comp and* $comp' \bowtie comp \rightsquigarrow \Phi$*, then we write comp′[comp] for the execution configuration* Φ*.* ■

Definition 3 (LCM and oLCM contextual equivalence)**.**

On oLCM *, we define that* $comp_1 \approx_{\text{ctx}} comp_2$ *iff*

$$\forall \mathscr{C}.\, \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[comp_1]\Downarrow_{-}^{T_{A,1},\text{stk\_base}_1} \Leftrightarrow \mathscr{C}[comp_2]\Downarrow_{-}^{T_{A,2},\text{stk\_base}_2}$$

with $T_{A,i} = \text{dom}(comp_i.ms_{\text{code}})$.

On LCM *, we define that* $comp_1 \approx_{\text{ctx}} comp_2$ *iff*

$$\forall \mathscr{C}.\, \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[comp_1]\Downarrow_{-} \Leftrightarrow \mathscr{C}[comp_2]\Downarrow_{-}$$

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

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*

$$comp_1 \approx_{\text{ctx}} comp_2 \quad \Leftrightarrow \quad comp_1 \approx_{\text{ctx}} comp_2$$ ■

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 any property the source language has must somehow be there after compilation whether or not it is a property of the target language. If the source language has a property that the target language doesn't have, then a compiled source program must use the available target language features to emulate the source language property in a way that it behaviorally matches exactly. In our case, LSE and WBCF was built into the semantics of oLCM, but they are not properties of LCM. In order to enforce these properties, components on LCM use StkTokens. Theorem 1 proves that StkTokens enforces these 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 will 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 iff the other does. The hard part is in defining when components are related. In the next section, we give an informal overview of the relation we define, and then we sketch the full-abstraction proof in Section 5.4.

5.1 Kripke Worlds

The relation between oLCM and LCM components is non-trivial: 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 if it is sealed with a seal that is only used for return pointers to the same code location. To accomodate such conditional relatedness, we construct our relation as a step-indexed Kripke logical relation with recursive worlds. Space constraints prevent us from explaining this in full detail in this paper. Instead, we will only highlight specific important parts of the logical relation in this section, and we refer to Skorstengaard et al. [2018b] for details and to Skorstengaard et al. [2018a] for a more comprehensive description of a logical relation for a capability machine. Additionally, for presentation, we will omit details, like step indexing, that are important for correctness but otherwise uninteresting.

Assumptions about the system that relatedness is predicated on are gathered in (Kripke) worlds. We use a type of worlds tailored to our purposes. They consist of three sub-worlds: $\text{Wor} = \text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}}$, capturing assumptions about the heap[4], the inactive and the active part of the stack, respectively. Sub-worlds consist of a finite mapping from region names to regions which come in two forms (spatial and shared):

$$\begin{aligned}
\text{World}_{\text{heap}} &= \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} + \text{Region}_{\text{shared}}) \\
\text{World}_{\text{call_stack}} &= \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \times \text{Addr}) \\
\text{World}_{\text{free_stack}} &= \text{RegionName} \rightharpoonup \text{Region}_{\text{spatial}}
\end{aligned}$$

[4]Actually, we use the term heap to describe all memory except the stack, including, for example, code memory.

Different parts of the world can contain different types of regions: heap-related assumptions can be either spatial or shared (see below), while stack-related regions must be spatial. Additionally, regions for inactive parts of the stack additionally include an address specifying the return address for that stack frame.

Regions in $\text{Region}_{\text{shared}}$ specify the presence of an invariant in the system, shared with the rest of the system. They are tagged with the syntactic token pure and may prescribe two different types of requirements:

$$\text{Region}_{\text{shared}} = \begin{cases} \{\text{pure}\} \times (\text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{MemSeg})) \times \\ \quad (\text{Seal} \rightharpoonup \text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{Sealables} \times \text{Sealables})) \end{cases}$$

First, they may require the presence of oLCM and LCM memory segments satisfying a given relation in $\text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{MemSeg}^2)$ (readers unfamiliar with Kripke and step-indexed logical relations may read $\text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{MemSeg}^2)$ as the set of functions from Wor to relations on memory segments). A region might, for example, require the presence of a certain list of instructions at a certain set of memory addresses in both oLCM and LCM. If a memory segment is owned by a given region, then that memory segment must be disjoint from the memory segments owned by all other regions. Second, a shared region may also contain a partial function from seals to relations on pairs of sealable capabilities: $\text{Seal} \rightharpoonup \text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{Sealables} \times \text{Sealables})$. When the region defines such a relation for a given seal, then no other region in the world can do the same, and any value signed with that seal will be required to satisfy the registered relation.

Spatial regions are similar to shared regions, but they are tagged as spatial or spatial_owned and may not specify seal invariants. Additionally, a spatial region may also be revoked:

$$\text{Region}_{\text{spatial}} = \{\text{spatial}, \text{spatial\_owned}\} \times (\text{Wor} \xrightarrow{mon,\ ne} \text{URel}(\text{MemSeg}^2)) \cup \{\text{revoked}\}$$

The difference between spatial and shared regions is related to linearity and ownership. For example, a LCM linear capability to a piece of memory is related to its oLCM counterpart, but only if no other linear capability overlaps with it. We will model such an assumption of exclusive ownership by making the relatedness rely on the presence of a spatial_owned region that only one value in the system may rely on. More concretely, we will define how to combine worlds W_1 and W_2 into a combined world $W_1 \oplus W_2$, on the condition that they represent compatible assumptions: W_1 and W_2 must contain the same regions except that they must respect exclusive ownership: pure regions must be present in both worlds, but a spatial_owned region can only be present in one and must be spatial in the other. When $W_1 \oplus W_2$ is defined, we say that worlds W_1 and W_2 are compatible and we refer to W_1 and W_2 as compatible partitions of the combined world.

The reason that we have spatial regions (in addition to spatial_owned ones) is for defining when oLCM and LCM memories are related. We need them to contain suitable memory segments for all regions in the system, even for regions owned by values that live outside the memory (for example register values). Those regions will be in the world for the memory, but only as spatial, i.e. they may not actually be referenced from within memory, but we still require them to be backed by suitable memory contents. We also use a relation $W_2 \sqsupseteq W_1$ that defines when a world W_2 is a future world of a world W_1. Relations that hold with respect to W_1, will then generally continue to hold in W_2. Our future world relation is fairly standard: the future world must contain all the previous world's regions, except that spatial regions are allowed to become spatial_owned (i.e. gaining ownership of a region will never break relatedness) or revoked (i.e. revoking spatial regions will never break relatedness).

Attentive readers may have noticed that our definition of worlds is actually cyclic: worlds in Wor contain regions in $\text{Region}_{\text{shared}}$, but those contain partial functions from the set Wor to something

else. Such recursive worlds are in fact common in Kripke models, and we use the method of Birkedal and Bizjak [2014]; Birkedal et al. [2011] (essentially an advanced form of step-indexing) to construct the set Wor and rigorously resolve the circularity.

In our proof, we use only a few different types of regions. For space reasons, we do not go into their definitions (see the Skorstengaard et al. [2018b]), but we give a brief overview here. First, we have the code region $\iota^{\text{code}}_{\overline{\sigma_{\text{ret}}},\overline{\sigma_{\text{clos}}},ms_{\text{code}}}$. This is a shared region that represents the assumption that memory segment ms_{code} is loaded in heap memory at a certain location, it is well-formed and it uses return seals $\overline{\sigma_{\text{ret}}}$ and closure seals $\overline{\sigma_{\text{clos}}}$. It takes ownership of those seals and registers appropriate invariants on the capabilities that may be signed with them.

A second type of regions $\iota^{\text{std,pure}}_{A}$ and $\iota^{\text{std,spatial\_owned}}_{A}$ governs heap or stack memory at a set of addresses A. It simply requires the presence of a memory segment for those addresses, such that the memory contains values that themselves satisfy the relation between values that we will see below. The contents of the memory is allowed to change as long as the new contents are still valid. Finally, a third type of regions $\iota^{\text{sta,spatial\_owned}}_{(ms_S,ms_T)}$ requires the presence of two given memory segment ms_S and ms_T and does not allow them to change. This region is used, for example, to govern inactive parts of the stack whose contents is required to remain unmodified.

5.2 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 summarised 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:

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

These relations are defined using a set of mutually recursive equations, with cyclicity resolved through another use of step-indexing. For space reasons, we cannot show all of these definitions, but we will try to give an overview.

Note first how we have two value relations, whose definitions are sketched in Figure 13. The difference is that the untrusted value relation $\mathcal{V}_{\text{untrusted}}(W)$ does not just express that the two values are related, but also that they are safe to pass to an untrusted adversary, i.e. they cannot be used to break LSE and WBCF. The trusted value relation does not have the latter requirement and is a superset of the former.

Both relations trivially include numbers (i, i) which are always related to themselves. The untrusted value relation also includes stack pointers and the underlying linear capability (with the same (non-executable) permission, range of authority, and current address), as well as syntactically equal memory capabilities, seals and sealed values, all under certain conditions involving the world W and the capability's properties.

Details are in the [Skorstengaard et al. 2018b], but roughly, for stack capabilities, the omitted condition requires that the world contains a spatial_owned region governing this part of the stack.

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

Fig. 13. Sketches of the trusted and untrusted value relation.

$$
\mathcal{R}_{tst}(W) = \left\{ (reg_S, reg_T) \;\middle|\; \begin{array}{l} \exists S : (\text{RegName} \setminus \{\text{pc}\}) \to \text{World}. \\ \quad W = \bigoplus_{r \in (\text{RegName} \setminus (\{\text{pc}\} \cup R))} S(r) \wedge \\ \quad \forall r \in \text{RegName} \setminus \{\text{pc}\}. \; (reg_S(r), reg_T(r)) \in \mathcal{V}_{tst}(S(r)) \end{array} \right\}
$$

$$
\mathcal{E}(W) = \left\{ \begin{array}{l} (w_{c,S}, w_{c,T}) \mid \\ \quad \forall reg_S, reg_T, ms_S, ms_T, ms_{stk}, stk, W_{\mathcal{R}}, W_{\mathcal{M}}. \\ \qquad (reg_S, reg_T) \in \mathcal{R}_{\text{untrusted}}(W_{\mathcal{R}}) \text{ and } (ms_S, stk, ms_{stk}, ms_T) : W_{\mathcal{M}} \text{ and} \\ \qquad \Phi_S = (ms_S, reg_S, stk, ms_{stk}) \text{ and } \Phi'_S = \Phi_S[reg.\text{pc} \mapsto w_{c,S}] \text{ and} \\ \qquad \Phi_T = (ms_T, reg_T) \text{ and } \Phi'_T = \Phi_T[reg.\text{pc} \mapsto w_{c,T}] \text{ and} \\ \qquad W \oplus W_{\mathcal{R}} \oplus W_{\mathcal{M}} \text{ is defined} \\ \qquad\quad \Rightarrow (\Phi'_S, \Phi'_T) \in O \end{array} \right\}
$$

$$
O = \{(\Phi_S, \Phi_T) \mid \Phi_S \Downarrow_- \Leftrightarrow \Phi_T \Downarrow_-\}
$$

Fig. 14. Simplified sketches of the register file relation $\mathcal{R}_{\text{untrusted}}(W)$, the relation for *jmp* targets $\mathcal{E}(W)$ and the observation relation $O(W)$.

For memory capabilities $((p, l), b, e, a)$, a region in the world must govern memory $[b, e]$, either spatial_owned or pure, depending on the linearity l of the capability. If the capability is executable ($p \in \{\text{RX}, \text{RWX}\}$), then we additionally require that the governing region is a code region and that the two capabilities are related jmp targets, as expressed by the relation $\mathcal{E}(W)$, in any future world (see below).

Seals allocated to trusted code are related to themselves only by $\mathcal{V}_{\text{trusted}}(W)$, but other seals are in both value relations. Sealed values are in both relations essentially when the sealed values satisfy the relation that was registered for the seal in a region of the world. Additionally, when they are combined with any other pair of values related by that relation, they must be related as `xjmp` targets (i.e. in $\mathcal{E}_{\text{xjmp}}(W)$). Finally, capabilities to code memory are related to themselves in the trusted value relation ($\mathcal{V}_{\text{trusted}}(W)$) when there is an appropriate code region in the world. They are not in the untrusted value relation because the code memory contains copies of the return seals used by the code, which must not end up in the hands of an adversary.

In Figure 14, we show sketches of the register file relation $\mathcal{R}_{\text{untrusted}}$, the relation between `jmp` targets $\mathcal{E}(W)$ and the observation relation O. The trusted/untrusted register file relation simply requires that all registers except pc are in the corresponding value relation (in a compatible world partition). Two execution configurations are in the observation relation O if one terminates

whenever the other does[5]. The $\mathcal{E}(W)$ relation then includes any two words which can be plugged into related register files and memories (in compatible worlds), to obtain execution configurations in the observation relation.

5.3 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.

Lemma 1 (FTLR (roughly)). *If* $[b, e] \subseteq \mathrm{dom}(ms_{\mathrm{code}})$ *and* $W.\mathrm{heap}(r) = \iota^{\mathrm{code}}_{\overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, ms_{\mathrm{code}}}$, *and either* $[b, e] \subseteq T_A$ *and* ms_{code} *behaves reasonably (see Section 4.2) or* $[b, e] \# T_A$, *then*

$$(((\textsc{rx}, \mathrm{normal}), b, e, a), ((\textsc{rx}, \mathrm{normal}), b, e, a)) \in \mathcal{E}(W)$$ ■

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.

5.4 Full Abstraction Proof Sketch

Using Lemma 1, we can now proceed to proving Theorem 1 (full abstraction). First, we extend the logical relation into an omitted relation on components $(comp_S, comp_T) \in C(W)$. Using Lemma 1 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 2 (FTLR for components). *If comp is a well-formed component, i.e.* $\vdash comp$ *and either* $\mathrm{dom}(comp.ms_{\mathrm{code}}) \subseteq T_A$ *and comp is a reasonable component; or* $\mathrm{dom}(comp.ms_{\mathrm{code}}) \# T_A$, *then there exists a* W *such that* $(comp, comp) \in C(W)$. ■

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

Lemma 3. *If* $(\mathscr{C}_S, \mathscr{C}_T) \in C(W_1)$ *and* $(comp_S, comp_T) \in C(W_2)$ *and* $W_1 \oplus W_2$ *is defined, then* $\mathscr{C}_S[comp_S]$ *terminates iff* $\mathscr{C}_T[comp_T]$ *terminates.* ■

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

Proof of Theorem 1. The proofs of both directions are similar, so we only show the right direction. To show the LCM contextual equivalence, assume w.l.o.g a well-formed context $\mathscr{C}$ such that $\mathscr{C}[comp_1]\Downarrow$. The proof is sketched in Figure 15. 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 $\mathscr{C}$ along with Lemma 2 and 3. Now we know that $\mathscr{C}[comp_1]\Downarrow$, so by the assumption that $comp_1$ and $comp_2$ are contextually equivalent on oLCM we get $\mathscr{C}[comp_2]\Downarrow$, i.e. arrow (2) in the figure. To prove arrow (3), we again

[5]The actual definition in the [Skorstengaard et al. 2018b] is complicated a bit by step-indexing and the fact that we actually use two separate observation relations for left- and right-approximation.

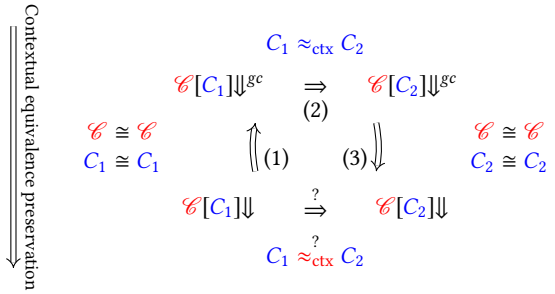


Fig. 15. Proving one direction of fully abstract compilation (contextual equivalence preservation).

apply Lemma 2, 3; but this time, we use the assumption that $comp_2$ is well-formed and reasonable and that $\mathscr{C}$ is well-formed. □

6 DISCUSSION

6.1 Full Abstraction

Our formulation of WBCF and LSE using a fully abstract overlay semantics has an important advantage with respect to others. 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 want to 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.

6.2 Practical Applicability

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 non-zero 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. As we understand it, the main problems to solve 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 in principle be non-aliased) and just use a non-atomic load in that case.

7 RELATED WORK

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

Capability machines originate with Dennis and Van Horn [1966] and we refer to Levy [1984] and Watson et al. [2015b] for an overview of previous work. The capability machine formalized in Section 2 is modelled after CHERI [Watson et al. 2015b; Woodruff et al. 2014]. This is a recent, relatively mature capability machine which combines capabilities with a virtual memory approach in the interest of backwards 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 well-bracketed control flow at a low level but most are restricted to preventing particular types of attacks and enforce only partial correctness of control flow. This includes particularly the line of work on control-flow integrity [Abadi et al. 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 [Carlini et al. 2015; Evans et al. 2015].

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 [Juglaret et al. 2016; Patrignani 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. 2015b]. 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 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. In follow-up work, Abate et al. [2018] extend this approach to scenarios where components only start to exhibit undefined behavior after a number of well-defined steps. If we see reasonable behavior as defined behavior, then our full abstraction result can be seen as an application of this same idea. Our results do not apply to dynamic compromise scenarios because they are intended to be used in the verification of a secure compiler where these scenarios are not relevant.

ACKNOWLEDGMENTS

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

REFERENCES

Martín Abadi. 1999. Protection in programming-language translations. In Secure Internet programming. Springer-Verlag, 19–34.

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

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005b. A Theory of Secure Control Flow. In Formal Methods and Software Engineering. Springer Berlin Heidelberg, 111–124.

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In Computer and Communications Security (CCS '18). ACM, 18. https://doi.org/10.1145/3243734.3243745

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

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

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In USENIX Security. USENIX Association.

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

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, fully-abstract compilation by approximate back-translation. Logical Methods in Computer Science 13 (10 2017). Issue 4. https://doi.org/10.23638/LMCS-13(4:2)2017

Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In Computer and Communications Security. ACM. https://doi.org/10.1145/2810103.2813646

Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In CSF. IEEE Computer Society Press.

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

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016). ACM, 103–116. https://doi.org/10.1145/2951913.2951941

Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully Abstract Compilation. In Computer Security Foundations. IEEE.

Marco Patrignani and Deepak Garg. 2017. Secure compilation and hyperproperty preservation. In Computer Security Foundations. IEEE.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018a. Reasoning About a Machine with Local Capabilities. In Programming Languages and Systems. Springer International Publishing, 475–501.

Lau Skorstengaard, Dominique Devriese, and Lau Birkedal. 2018b. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities - Technical Report with Proofs and Details. https://arxiv.org/abs/1811.02787

Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. First Monday 2, 9 (Sept. 1997). https://doi.org/10.5210/fm.v2i9.548

Nick Szabo. 2004. Scarce Objects. https://nakamotoinstitute.org/scarce-objects/

Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W Moore, Steven J Murdoch, Philip Paeps, and others. 2012. CHERI: A Research Platform Deconflating Hardware Virtualization and Protection. In Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE).

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. 2018. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory.

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, and Stacey Son. 2015a. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture. Technical Report UCAM-CL-TR-876. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-876.html

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

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

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