StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities

ANONYMOUS AUTHOR(S)

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.

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

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. Particularly, 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.

Local state encapsulation (LSE) is the guarantee that when a function invokes another function, its local variables (saved on its stack frame) will not have been read or modified when 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 varies 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].

To enforce WBCF and LSE on a capability machine, there are essentially two approaches. A first approach uses separate stacks for distrusting components, and a central, trusted stack manager component 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 scales poorly to large amounts of distrusting components because of the need to reserve separate stack space for all components. Also in the presence of higher-order values (e.g. function pointers, objects etc.), 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 this can be done 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. [2018] has 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. 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. 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. [2018] has 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. 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.

A second contribution of this work is the way in which we formulate these two properties using a technique we call fully abstract overlay semantics. Formulations in previous work are either partial and not suitable for reasoning [Abadi et al. 2005a] or lacked evidence of generality [Skorstengaard et al. 2018]. Our new formulation 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 accommodate this, 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. that components are

1Although they have been mentioned in some technical documents.
StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities

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.

Any reasonable capability machine needs a way to set up boundaries between security domains as well as a way to cross the boundaries in a way where each domain retains their authority. On LCM we have CHERI-like sealed capabilities to achieve this [Watson et al. 2015b]. Syntactically,
sealed capabilities add two types of capabilities. A set of seals $\text{seal}(\sigma_{base}, \sigma_{end}, \sigma)$ consists of a range of available seals $[\sigma_{base}, \sigma_{end}]$ and the current seal $\sigma$, all represented by a natural number. A sealed capability $\text{seal}(\sigma, sc)$ consists of a sealable capability $sc$ and the seal $\sigma$ it has been sealed with. Sealable capabilities $sc \in \text{Sealables}$ are sets of seals or memory capabilities.

Words on LCM are capabilities and data (represented by $\mathbb{Z}$). We assume a finite set of register names $\text{Name}$ containing at least $\text{pc}$, $r\text{data}$, $r\text{code}$, $r\text{stk}$, $r\text{data}$, $r_{t1}$, and $r_{t2}$, and we define register files as functions from register names to words. We have complete memories mapping all addresses to words and memory segments mapping 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 machine as well as capability-related instructions. The standard instructions are: unconditional and conditional jump ($\text{jmp}$ and $\text{jnz}$), copy between registers ($\text{move}$), instructions that load from memory and store to memory ($\text{load}$ and $\text{store}$), and arithmetic operations ($\text{plus}$, $\text{minus}$, and $\text{lt}$). The simplest of the capability instructions simply inspect the properties of capabilities: type ($\text{gettype}$), linearity ($\text{getl}$), range ($\text{getb}$ and $\text{gete}$), current address or seal ($\text{geta}$) or permission ($\text{getp}$). The current address (or seal) of a capability (or set of seals) can be shifted by an offset ($\text{cca}$) or set to the base address ($\text{seta2b}$). The $\text{restrict}$ 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 ($\text{split}$). The reverse is possible using $\text{splice}$. Sealables can be sealed using $\text{cseal}$ and pairs of sealed capabilities can be unsealed by crossing security boundaries ($\text{xjmp}$, see below). Finally, LCM has instructions to signal whether an execution was successful or not ($\text{halt}$ and $\text{fail}$).

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 $\Phi$ which results in a new executable configuration or one of the two terminated configurations.
\[
\Phi(\text{pc}) = ((p, \_), b, e, a) \\
b \leq a \leq e \quad p \in \{\text{rw}x, \text{rx}\} \\
\Phi \rightarrow \lbrack\text{decode}(\Phi.\text{mem}(a))\rbrack(\Phi) \\
\lnot(\Phi \rightarrow \Phi') \quad \Phi' \neq \text{failed} \\
\Phi \rightarrow \text{failed}
\]

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

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

\[
x\text{jmpRes}(c_1, c_2, \Phi) = \begin{cases} 
\Phi[\text{reg}.\text{pc} \mapsto c_1][\text{reg}.\text{rdata} \mapsto c_2] & \text{nonExec}(c_2) \\
\text{failed} & \text{otherwise}
\end{cases}
\]

### Table

<table>
<thead>
<tr>
<th>(i \in \text{Instr})</th>
<th>(<a href="%5CPhi">i</a>)</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>halted</td>
<td></td>
</tr>
<tr>
<td>fail</td>
<td>failed</td>
<td></td>
</tr>
<tr>
<td>move r (\text{rn})</td>
<td>\text{updPc}(\Phi[\text{reg}.\text{rn} \mapsto w_1])</td>
<td>(r_n \in \text{RegName}) and (w_1 = \Phi(\text{rn})) and (w_2 = \text{linClear}(\text{rn}))</td>
</tr>
<tr>
<td>load (r_1 \text{to} r_2)</td>
<td>\text{updPc}(\Phi[\text{reg}.r_1 \mapsto w_1])</td>
<td>(\Phi(r_2) = ((p, _), b, e, a) \land b \leq a \leq e) and (p \in {\text{rw}x, \text{rw}, \text{rx}, \text{r}}) and (w_1 = \Phi.\text{mem}(a)) and \text{isLinear}(\text{w_1}) \Rightarrow p \in {\text{rw}x, \text{rw}}) and (w_2 = \text{linClear}(\text{w_1}))</td>
</tr>
<tr>
<td>store (r_1 \text{to} r_2)</td>
<td>\text{updPc}(\Phi[\text{reg}.r_2 \mapsto w_2])</td>
<td>(\Phi(r_1) = ((p, _), b, e, a) \land p \in {\text{rw}x, \text{rw}}) and (b \leq a \leq e) and (w_2 = \text{linClear}(\Phi(r_2)))</td>
</tr>
<tr>
<td>geta (r_1 \text{to} r_2)</td>
<td>\text{updPc}(\Phi[\text{reg}.r_1 \mapsto w])</td>
<td>\text{If} (\Phi(r_2) = ((_), _, _, _)) or (\Phi(r_2) = \text{seal}(_, _, _)), (w = a) and (w = -1)</td>
</tr>
<tr>
<td>cca (r \text{to} \text{rn})</td>
<td>\text{updPc}(\Phi[\text{reg}.r \mapsto w])</td>
<td>(\Phi(r_n) = 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</td>
</tr>
<tr>
<td>jmp (r)</td>
<td>(\Phi[\text{reg}.r, \text{pc} \mapsto w, \Phi(r)])</td>
<td>(w = \text{linClear}(\Phi(r)))</td>
</tr>
<tr>
<td>xjmp (r_1 \text{to} r_2)</td>
<td>(\Phi')</td>
<td>(\Phi(r_1) = \text{seal}(\sigma, c_1)) and (\Phi(r_2) = \text{seal}(\sigma, c_2)) and (w_1 = \text{linClear}(\text{c_1})) and (w_2 = \text{linClear}(\text{c_2})) and (\Phi' = x\text{jmpRes}(c_1, c_2, \Phi[\text{reg}.r_1, r_2 \mapsto w_1, w_2]))</td>
</tr>
<tr>
<td>split (r_1 \text{to} r_2 \text{to} r_3 \text{to} \text{rn})</td>
<td>\text{updPc}(\Phi[\text{reg}.r_3 \mapsto w])</td>
<td>(\Phi(r_3) = ((p, l), b, e, a)) and (\Phi(\text{rn}) = n \in \mathbb{N}) and (b \leq n &lt; e) and (c_1 = ((p, l), b, n, a)) and (c_2 = ((p, l), n + 1, e, a)) and (w = \text{linClear}(\Phi(r_1)))</td>
</tr>
<tr>
<td>splice (r_1 \text{to} r_2 \text{to} r_3)</td>
<td>\text{updPc}(\Phi[\text{reg}.r_2 \mapsto w_2])</td>
<td>(\Phi(r_2) = ((p, l), b, n, _)) and (\Phi(r_3) = ((p, l), b, n, e, a)) and (b \leq n &lt; e) and (c = ((p, l), b, e, a)) and (w_2, w_3 = \text{linClear}(\Phi(r_2), \Phi(r_3)))</td>
</tr>
<tr>
<td>cseal (r_1 \text{to} r_2)</td>
<td>\text{updPc}(\Phi[\text{reg}.r_1 \mapsto \text{sc}])</td>
<td>(\Phi(r_1) \in \text{Sealables}) and (\Phi(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma)) and (\sigma_b \leq \sigma \leq \sigma_e) and (\text{sc} = \text{seal}(\sigma, \Phi(r_1)))</td>
</tr>
</tbody>
</table>

... otherwise

---

Fig. 3. An excerpt of the operational semantics of LCM.
The executed instruction is determined by the capability in pc register, i.e. $\Phi(pc)$ (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 things are satisfied, then the word pointed to by the capability is decoded to an instruction which is interpreted relatively to $\Phi$. The interpretation 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 two approaches. In the first one, we use the function $linClear$ that returns 0 for linear capabilities and is the identity for all other words. When a word $w$ is transferred on the machine, then the source of $w$ is overwritten with $linClear(w)$ which clears the source if $w$ was linear and leaves it unchanged otherwise. In the case of move, the source register $r_2$ is overwritten with $linClear(\Phi(r_2))$. We see the other approach later.

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, and they clear the source of a linear capability transfer. 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 geta instruction projects the current address (or seal) from a capability (or set of seals), and returns $-1$ for data and sealed capabilities. $cca$ changes the current address or seal of a capability or set of seals, respectively, by a given offset. It handles linearity by updating the capability in-place, i.e. the source register is used as the target register. This is the second approach we use to uniformly handle linearity. The jmp instruction is a simple jump that just sets the pc register.

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 $\sigma$ is within the range of available seals, the $cseal$ instruction seals sc with $\sigma$. 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 it only becomes 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

---

At page 6, Anon. (2018). 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.
StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities

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 one with \([base, n]\) as its range of authority and the other with \([n + 1, end]\) as its range 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.

At this time, the executable configuration is our only way to specify the state of the capability machine. We would, however, much prefer a notion of programs that only describes the essential parts of an entity on the machine and only later instrument the program to make it into an executable configuration. In later sections, we describe different entities on the capability machine that interact with each other. In order for them to interact, we need to be able to link them together, and to this end we introduce a notion of components. A component is basically a program with open ends in the form of imports that need to be satisfied. A component also has exports allowing other components to link with it. We define the notion of a component in Figure 4. A base component \(comp_0\) consists of the following: a code memory segment, a data memory segment, a list of imported symbols, a list of exported symbols, two lists specifying the available seals\(^2\), and a set of all the linear addresses. 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 base component can be seen as library implementations because it does not specify a main entry point of execution. A component is either a library component or a base component with a main in the form of a pair of sealed capabilities which can be seen as a program that still needs to be linked with libraries. Components are combined into new components by linking them together. Libraries can be linked together and incomplete programs can be linked with libraries, but two incomplete programs cannot be linked with each other. Two components can be linked when their memories, seals, and linear addresses 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, in the sense that they have the same symbol, 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. Components and linking are defined in Figure 4.

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

**Definition 1 (Programs and Contexts).** We define a program to be a component \((comp_0, c_{main,c}, c_{main,d})\) with an empty import list. A context for a component \(comp\) is another component \(comp'\) such that \(comp \Rightarrow comp'\) is a program.

In Section 4, we will get back to how a program is instrumented to create an executable configuration.

Some simplifications has been made in this presentation of LCM. See TR [2018] for the details.

\(^2\)We will return to the seals in Section 4.
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 take and motivate them with the attacks they prevent.
**StkTokens**: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities

StkTokens is based on a traditional single stack, shared between all components. As we are on a capability machine, it is natural to add some extra protection to stack and return pointers. 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. The caller provides the adversary with a stack capability and a return pair that they are supposed to use for the call, but the adversary 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 some 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 a component we do not trust 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 two calls to the adversarial component. The adversary may have stored the pair of sealed return capabilities from the first call in order to use it in the second call where they are not allowed to use it. 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, the standard memory and object-like capabilities (here sealed capabilities) do 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 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, such that they have a capability for their local stack frame and a capability for the unused part of the stack. The stack capability for their local stack frame is sealed and used as the data part of the sealed return pair, and the stack 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 will need the capability the caller produced in order to access their local data. While the caller gives the capability to the adversary, it is only after the caller seals it which makes the capability opaque. 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 enforcement mechanism that allows us to make sure that the order of the call stack is kept. One way to enforce this would be to

---

3Revocation 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.

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. 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. If we could have empty stack frames on the call stack, 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 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 does not make sure 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 later can 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 is fixed as a compile-time constant. 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, we ensure that at the start of execution the stack memory is only referenced by a single linear stack capability. Because of this, we can assert validity of the return token simply by checking its base address and splicing it with the caller’s stack frame. There is no need to check linearity because we know that 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 of course sealed with the same seal. All call points need to be associated with a unique seal (a return seal) that is only used to seal the capabilities in the return pair 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 caused the stack frame to be pushed on the call stack which would break 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, such as leaving a seal in a register or writing it to adversarial memory, as well as indirect leaks, where a capability for reading a return seal (either directly or indirectly) from memory is leaked.

In the description of the StkTokens calling scheme, it has sometimes been 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. This means that any component can ensure WBCF and LSE by employing StkTokens.

In this section, we have described our proposed calling convention, StkTokens. The measures taken are:

1. Check the base address of the stack capability before and after calls
2. Make sure that local stack frames are non-empty

4The stack grows downwards in the address space
(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 stack capability for your local stack frame 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, all return seals are only used for one call site, and return seals are not leaked.

Item 1–4 are captured by the code in Figure 8 with the exception of checking stack base before calls. We do not include this check because it only needs to happen once between two calls which means that the check after a call suffices if the base of the stack pointer is not changed subsequently.

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

Fig. 8. The instructions that constitute a call. The variable \( \text{off}_{\text{pc}} \) is the offset from the first instruction of the call to the set of seals it uses. The variable \( \text{off}_{\sigma} \) is the offset in the set of seals. The constant \( \text{stk\_base} \) is the globally agreed on stack base. There are a number of “magic” numbers in the code: line 1: 42, garbage data to ensure a non-empty stack. Line 7: −5, offset to the first instruction (from the point the capability was moved out of the pc-register). Line 12: 5, offset to the return address. Line 19: 5, offset to fail. Line 21: offset to address after fail.
4 FORMULATING SECURITY WITH A FULLY ABSTRACT OVERLAY SEMANTICS

A central contribution of this paper is to prove that StkTokens actually guarantees LSE and WBCF. Our proof uses 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 formulated as the 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 the corresponding components in the well-behaved overlay semantics. In other words, they cannot do anything that doesn’t correspond to something that a well-behaved component, respecting LSE and WBCF, can also do. More formally, our full-abstraction result states that two trusted components are indistinguishable to arbitrary other components in the regular semantics if they are indistinguishable to arbitrary other components in the overlay semantics.

Our formal results are complicated by the fact that they only hold for well-behaved components that respect the basic rules of the calling convention, and also on a sane initial configuration of the system. For example, the system should be setup such that seals that components use for constructing return pointers should not be shared with other components. We envision distributing seals as a job for the linker, so this means our results depend on a linker to do this properly. As another example, a seal that is used to construct a return pointer, can be used again to construct other return pointers for the same return point, but a different seal must be used for others. Such seals should also never be passed to other components. Such 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.

4.1 Overlay semantics

The overlay semantics we define for LCM (oLCM) 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 other calls. A stack frame contains encapsulated stack memory as well as the program point execution is supposed to return to. The free stack memory is the 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. Return pointers are generated by calls from the capabilities they correspond to, and they are turned back to the capabilities they correspond to upon return. Even though the return pointers both correspond to capabilities with read permission, neither can be used for reading. Say, we allowed reads through a data return pointer, then that would correspond to reading from some stack memory that is encapsulated in a stack frame which would break LSE.

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. 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, it should not be possible to have something else than a stack pointer for that part of memory.

As discussed earlier, our formal results only provide guarantees for components that respect the calling convention. However, there is no such assumption on the untrusted other components they interact with. To formalize this distinction, oLCM has a set of trusted addresses $T_A$. Only instructions at these addresses are allowed to perform calls that push stack frames to the call stack, i.e. calls for which oLCM provides native LSE and WBCF guarantees. The constant $T_A$ is passed around as a parameter of the LCM 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.

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 ($\text{call}_{\text{off} \cdot \text{off} \cdot \text{r}} \cdot \text{r}_1 \cdot \text{r}_2 \text{corresponds to the } i\text{'th instruction in the figure and call_len} \leq 26$, i.e. the number of instructions). Calls are only executed

\[
\begin{align*}
\text{Sealables} & := \text{Sealables} \mid \text{stack.ptr}(\text{perm}, \text{base}, \text{end}, a) \mid \\
& \quad \text{ret.ptr-data}(\text{base}, \text{end}) \mid \text{ret.ptr-code}(\text{base}, \text{end}, a) \\
\text{StackFrame} & \overset{\text{def}}{=} \text{Addr} \times \text{MemSeg} \\
\text{Stack} & \overset{\text{def}}{=} \text{StackFrame}^* \\
\text{ExecConf} & \overset{\text{def}}{=} \text{Memory} \times \text{RegFile} \times \text{Stack} \times \text{MemSeg} \\
\text{off}_{\text{pc}} \cdot \text{off}_{\sigma} & \in \mathbb{N} \\
\text{Instr} & := \text{Instr} \mid \text{call}_{\text{off} \cdot \text{off} \cdot \text{r}} \cdot \text{r} \cdot \text{r}
\end{align*}
\]

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.
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 a call is also shown in Figure 10, and it may look daunting due to a fair bit of bookkeeping, but it essentially does the following: Given a stack pointer in the stack register, 42 is pushed to the stack to ensure that the stack frame will be non-empty. Based on the stack pointer in $r_{stk}$, a stack pointer for the callee is created (this corresponds to the return token of StkTokens) as well as a return pointer which governs the private part of the stack, i.e. addresses $[a_{stk}, e_{stk}]$ which is 42 and everything (conceptually) below on the stack. A new stack frame is pushed to the stack. The stack frame consists of the local stack (addresses $[b_{stk}, a_{stk} - 1]$) as well as the address where execution is supposed to continue on return. The return address in the stack frame is an important piece of meta data as it connects the stack frame to the return point and thus allows us on return to check whether we returned to the right place. The return pointers are sealed with the seal dictated by $off_{pc}$ and $off_{\sigma}$ and placed in registers $r_{code}$ and $r_{data}$, so the callee can use them to return. Finally, control is transferred in the same fashion xjmp does it.

The return pair can only be used to return with. They only exists on oLCM in sealed capabilities, so it is only natural that x jmp takes care of returning. To this end, we update the semantics of x jmp by redefining $xjmpRes$. The function remains the same except when $c_1$ and $c_2$ are code and data return pointers, respectively. In this case, the top stack frame is popped from the stack and added to the free stack. The execution continues at the return point with a stack pointer in $r_{stk}$ for the free stack that now includes the memory of the popped stack frame. However, the execution only continues if the code return pointer actually points to the correct return address, the base of the returned stack pointer is stk_base, and the returned stack pointer is adjacent to the data return pointer (which corresponds to checking the validity of the return token in StkTokens).

oLCM supports tail calls. A tail call is a call from a caller that is done executing and thus don’t need to be returned to. 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 x jmp. The callee needs to return to the caller’s caller, so the caller leaves the return pair they received for the callee to use.

By inspecting the operational semantics of oLCM, we can see that it does not allow reads from inactive stack frames on the call stack. Further, the built-in call and returns make sure that a stack frame is pushed on call, and a return is to the address in the topmost stack frame on the call stack.

In other words by inspection of the operational semantics, oLCM guarantees LSE and WBCF. It is important to point this out as StkTokens guarantee exactly the LSE and WBCF we have on oLCM.

### 4.2 Well-formedness and reasonability

The judgement $T_A \vdash \text{comp}$ specifies what components are well formed, i.e. satisfy the 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: $(ms_{code}, ms_{data}, \text{import}, \text{export}, \sigma_{ret}, \sigma_{clos}, A_{linear})$. The return seals $\sigma_{ret}$ are the seals supposed to be used to seal return pointers and the closure seals $\sigma_{clos}$ all the other seals in a component. If a component is not trusted, i.e. the domain of the code memory $ms_{code}$ is disjointed 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 the seals in $\sigma_{ret}$ and $\sigma_{clos}$. 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), it must have access to the return seal specified by that call. The return seal also needs to be unique to that call, so no other call can specify the same seal.
\[\Phi(\text{pc}) = ( (p, \_), b, e, a) \quad \text{[a, a + call\_len - 1] \subseteq T_A} \quad \text{[a, a + call\_len - 1] \subseteq [b, e]}\]
\[p \in \{\text{rwx}, \text{rx}\} \quad \Phi.\text{mem}(a, \ldots, a + \text{call\_len} - 1) = \text{call\_off}_{\text{pc}, \text{off}}^a r_1 r_2, \ldots, \text{call\_off}_{\text{pc}, \text{off}}^a r_1 r_2\]
\[\Phi \rightarrow T_A, \text{stk\_base} \left[ \text{call\_off}_{\text{pc}, \text{off}}^a r_1 r_2 \right]\]

\[
\begin{array}{ccc}
\text{i} \in \text{Instr} & |i| (\Phi) & \text{Conditions} \\
halt & \text{halted} & \\
\end{array}
\]

... (the operational semantics of LCM)

\[
\begin{align*}
\Phi(r_1) & = \text{stack\_ptr}(p, b, e, a) \\
p & \in \{\text{rwx}, \text{rw}\} \quad \text{and} \quad b \leq a \leq e \quad \text{and} \\
w_2 & = \text{linClear}(\Phi(r_2)) \quad \text{and} \quad a \in \text{dom}(\text{ms}.)
\end{align*}
\]

\[
\begin{align*}
\Phi(r_1) & = \text{sealed}(\sigma, c_1) \\
\Phi(r_1) & = \text{sealed}(\sigma', c_2) \\
\Phi(r_{\text{stk}}) & = \text{stack\_ptr}(\text{rw}, b, e, a) \\
b_{\text{stk}} & < a_{\text{stk}} \leq e_{\text{stk}} \\
\text{ms}_{\text{stk}, \text{priv}} & = \text{ms}_{\text{stk}}[ \text{stk\_base} ][a_{\text{stk}} \mapsto 42]
\end{align*}
\]

\[
\begin{align*}
\text{opc} & = a + \text{call\_len} \\
\text{c}_{\text{opc}} & = \text{ret\_ptr\_code}(b, e, \text{opc}) \\
\text{stk}' & = (\text{opc}, \text{ms}_{\text{stk}, \text{priv}}) \\
\text{mem}(a + \text{call\_len}) & = \text{ret\_ptr\_data}(\text{stk}) \\
\Phi(r_{\text{stk}}) & = \text{stack\_ptr}(\text{rw}, \text{stk\_base}, a_{\text{stk}} - 1, a_{\text{stk}} - 1)
\end{align*}
\]

\[
\begin{align*}
\text{w}_1, w_2 & = \text{linClear}(\Phi(r_1), \Phi(r_2)) \\
\text{reg}' & = \text{reg}(\text{w}_1, \text{w}_2) \\
\text{reg}' & = \text{reg}(\text{w}_1, \text{w}_2)
\end{align*}
\]

\[
\begin{align*}
\text{failure} & = \text{reg}(\text{stk\_base}) \\
\text{reg}' & = \text{reg}(\text{stk\_base}) \\
\text{reg}' & = \text{reg}(\text{stk\_base}) \\
\text{reg}' & = \text{reg}(\text{stk\_base})
\end{align*}
\]

Fig. 10. An excerpt of the operational semantics of oLCM.

\[
\begin{align*}
\text{callConditions}(r_1, r_2, \text{off}_{\text{pc}}, \text{off}_{\sigma}) & = (\text{mem}, \text{reg}, \text{stk}, \text{ms}_{\text{stk}}) \quad (\text{mem}, \text{reg}', \text{stk}', \text{ms}_{\text{stk}, \text{rest}})
\end{align*}
\]

\[
\begin{align*}
\text{xjmpRes}(c_1, c_2, \Phi) =
\end{align*}
\]

\[
\begin{align*}
\text{nonExec}(c_2) \quad \text{and} \quad c_1 & \neq \text{ret\_ptr\_code}(\_.) \quad \text{and} \quad c_2 \neq \text{ret\_ptr\_data}(\_.)
\end{align*}
\]

\[
\begin{align*}
\text{ failure} & = \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
\text{data memory } \text{ms}_{\text{data}} \text{ may contain data, capabilities, and sealed capabilities. The capabilities in the data memory can only have authority over the data memory itself. By allowing this, a component}
\end{align*}
\]

, Vol. 1, No. 1, Article . Publication date: July 2018.
can be constructed with predetermined data structures. In order to respect Write-XOR-Execute, the capabilities can at most have a read-write permission. The capabilities also need to respect linearity which means that a linear capability cannot have any aliases, and it must be for a range of the predetermined linear addresses \( \mathcal{A}_{\text{linear}} \). 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 \( \text{export} \) 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. Finally, the addresses of the imports \( \text{import} \) must be addresses in the data memory.

The static guarantees given by \( T_A \vdash \text{comp} \) 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). We more precisely 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 TR [2018]

### 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 \( \text{comp} \), the judgement \( \text{comp} \rightsquigarrow \Phi \) defines an initial execution configuration that can be executed. It works almost the same on LCM and oLCM. 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 \( r_{\text{stk}} \). 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 \( r_{\text{data}} \) 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 \( \text{comp}' \) is a context for component \( \text{comp} \) and \( \text{comp}' \rightsquigarrow \text{comp} \rightsquigarrow \Phi \), then we write \( \text{comp}'[\text{comp}] \) for the execution configuration \( \Phi \).

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

On oLCM, we define that \( \text{comp}_1 \approx_{\text{ctx}} \text{comp}_2 \) iff

\[
\forall C . \emptyset \vdash C \iff C[\text{comp}_1] \parallel T_{A_1, \text{stk base}_1} \leftrightarrow C[\text{comp}_2] \parallel T_{A_2, \text{stk base}_2}
\]

with \( T_{A_i} = \text{dom} (\text{comp}_i, ms_{\text{code}}) \).
On LCM, we define that $\comp_1 \approx_{\text{ctx}} \comp_2$ iff

$$\forall C. \emptyset \vdash C \Rightarrow C[\comp_1]_\downarrow \iff C[\comp_2]_\downarrow$$

where $\Phi \downarrow T_{A, \text{stk}_\text{base}} \iff \Phi \rightarrow T_{A, \text{stk}_\text{base}}$ halted and $\Phi \downarrow T_{A, \text{stk}_\text{base}} \overset{\text{def}}{=} \exists i. \downarrow T_{A, \text{stk}_\text{base}}$

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

$\comp_1 \approx_{\text{ctx}} \comp_2 \iff \comp_1 \approx_{\text{ctx}} \comp_2$

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

The backbone of the full-abstraction proof is a cross-language, step-indexed, Kripke logical relation. It relates the observable behavior of oLCM to the observable behavior of LCM. In the full-abstraction proof, it allows us to transfer termination assumptions between the two machines which enables the use of a contextual equivalence assumption. We first give an informal presentation of the logical relation before we return to a sketch of the full-abstraction proof.

### 5.1 Logical relation

In this section, we present the logical relation. Unfortunately, for space reasons, we do not give a comprehensive description of it. Instead we highlight specific parts to the logical relation and refer to TR [2018] for details and to Skorstengaard et al. [2018] for a more comprehensive description of a logical relation for a capability machine. Finally, for presentation, we will sometimes omit details, like step indexing, that are important for correctness but otherwise not interesting.

#### 5.1.1 Worlds

Generally speaking, a world is used in a logical relation to model assumptions about memory. oLCM has three kinds of memory: the heap, the call stack, and the free stack, and our worlds model this by being triples of sub-worlds. All of the sub-worlds are partial maps from RegionName (modelled as natural numbers) to regions. The regions differ for each of the sub-worlds based on what they model. Generally, the regions consist of a region type and a world indexed relation that specifies the memories allowed by the region. The region types model linearity as they indicate how the memory can be referenced. We have four types of regions: pure, spatial, spatial_owned, and revoked. The pure regions can be referenced by normal capabilities, the spatial_owned regions by linear capabilities, and the spatial and revoked regions by nothing. To prevent duplication of linear capabilities, two linear capabilities cannot address the same spatial_owned, so ownership has to be distributed between worlds. To this end, we have the partial
operation $W_1 \oplus W_2$ that requires $W_1$ and $W_2$ to be the same except that they have disjoint ownership: pure regions can be present in both worlds, but a spatial-owned region can only be present in one. Such a region must then be spatial in the other world. This is important because the logical relation will require that valid memories are backed by a suitable memory segment even when the memory does not own them (for example, when a linear capability for it is in one of the registers).

Revocation of linear capabilities is supported in a certain way using the revoked region. Finally, pure regions can also claim ownership of a seal by defining a seal interpretation for it. The seal interpretation is a world-indexed relation that must hold for all sealables signed by this seal. We split the regions in two categories:

$$\text{Region}_{\text{shared}} = \{(\text{pure}) \times (\text{Wor} \xrightarrow{\text{mon}, \text{ne}} \text{URel(MemSeg)}^2)) \times \}
\quad (\text{Seal} \rightarrow \text{Wor} \xrightarrow{\text{mon}, \text{ne}} \text{URel(Sealables \times Sealables)})
$$

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

and define the sub-worlds as follows:

World$_{\text{heap}} = \text{RegionName} \rightarrow (\text{Region}_{\text{spatial}} + \text{Region}_{\text{shared}})
$

World$_{\text{call_stack}} = \text{RegionName} \rightarrow (\text{Region}_{\text{spatial}} \times \text{Addr})
$

World$_{\text{free_stack}} = \text{RegionName} \rightarrow \text{Region}_{\text{spatial}}
$

The world for the heap can contain every kind of region because it can be addressed by both normal and linear capabilities. The call stack and free stack are only ever addressed by linear capabilities, so their sub-worlds only allow Region$_{\text{spatial}}$. Regions in World$_{\text{call_stack}}$ correspond to a stack frame, so they contain the return address for this frame.

Define worlds as World = World$_{\text{heap}} \times$ World$_{\text{call_stack}} \times$ World$_{\text{free_stack}}$. They are recursive as they contain world indexed relations. Recursive worlds are common in Kripke models, and we use the method of Birkedal and Bizjak [2014]; Birkedal et al. [2011] to construct Wor which solves the circularity. The future world relation $\sqsubseteq$ is fairly standard. It is extensional and requires all but spatial regions to stay the same type in future worlds. The spatial regions can become spatial-owned or revoked.

### 5.1.2 Code region. The code region is a central region to the logical relation that models the code memory of a component. It is especially interesting because its seal interpretation function specifies what the return and closure seals in a component can seal. The code region is defined as follows:

$$H^{\text{code},[\square]}_{\sigma_{\text{ret}}, \sigma_{\text{clos}}, m_{\text{code}} : \text{gc}} \overset{\text{def}}{=} (\text{pure}, H^{\text{code},[\square]}_{\sigma_{\text{ret}} \sigma_{\text{clos}} m_{\text{code}} : \text{gc}}, H^{\text{code},[\square]}_{\sigma_{\text{ret}} \sigma_{\text{clos}} m_{\text{code}} : \text{gc}})
$$

The code region is pure which makes it persistent in future worlds and reflects the fact that it is addressed by normal capabilities. The memory relation $H^{\text{code},[\square]}(\ldots)$ only relates $m_{\text{code}}$ to itself, i.e. our component’s code memory must not be modified. $H^{\text{code},[\square]}(\ldots)$ requires $m_{\text{code}}$ to be well-formed and that its contents is non-linear and in the value relation. As we will see later, we have two value relations: a trusted and an untrusted. The value relation used depends on whether the code region governs trusted or untrusted code. For now it suffices to know that the untrusted value relation contains everything that untrusted code can safely have, and the trusted value relation additionally includes things that trusted components are allowed to have but not pass away, e.g. return seals.

---

\*\*We have cheated in the presentation and included the solution to the circularity already.*

---
The seal interpretation function $H^{\text{code, } \square}_c (\ldots)$ dictates what the return seals $\overline{\sigma}_{\text{ret}}$ and closure seals $\overline{\sigma}_{\text{clos}}$ can seal. The return seals can only be used to seal return pointers that corresponds to the return from a call in the code of the region. Code return pointers must point to the first address after a call, and the call must be associated with the return seal in question. The corresponding capability on LCM should point to the address just after the xjmp in the call code (Figure 8) as it needs to perform the safety checks that are built into the call on oLCM. The code return pointer corresponds to a normal capability which means that it can be reused but only to return from the same call. Because of this, the code pointer is not associated with any specific call frame as it can be used by any data return pointer generated from the same call point. Reuse of return pairs is prevented by the data return pointer being linear, so the seal interpretation function relates data return pointers to linear capabilities. It also requires the data return pointer to refer to a spatial_owned region for one of the stack frames on the call stack. The return address of this region in World$_{\text{call_stack}}$ must be the return address of the call that the return seal is affiliated with.

$$H^{\text{code, } \square}_c (\overline{\sigma}_{\text{ret}}, \overline{\sigma}_{\text{clos}}) m_{\text{code}} \sigma W =$$

$$\{ (n, (\text{ret-ptr-code}(b, e, a + \text{call_len}), ((\text{rx, normal}), b, e, a + \text{xjmp-off} + 1))) | \ldots \} \cup$$

$$\{ (n, (\text{ret-ptr-data}(b, e), ((\text{rw, linear}), b, e, b - 1))) | \ldots \}$$

if $\sigma \in \overline{\sigma}_{\text{ret}}$

Unlike return seals, closure seals are allowed in both trusted and untrusted components, but we cannot allow them to use closure seals to seal the same things. If we allowed this, then a trusted component could seal a read-only capability for its code (i.e. access to return seals) with an untrusted closure seal and give it to the untrusted component which can unseal it with xjmp. The closure seals in untrusted code should therefore only be used to seal things from the untrusted value relation, i.e. things that are safe to transfer to an untrusted component. The trusted closure seals can be used to seal things from the untrusted value relation. This allows trusted seals to seal capabilities for the trusted code which is necessary in order to create a new closure.

5.1.3 Expression relations and friends. The logical relation has two expression relations. They both relate capabilities that can safely be used for execution in configurations that satisfy a given world. The first one, $E$, is standard as logical relations for capability machines go: It takes related register files and related memories, plugs the capabilities in question into the program counter, and requires it to observably behave the same way. In our case, the only observable thing on the capability machines is termination. The observation relation $O$ (Figure 11) defines when a configuration observably approximates another. With sealed capabilities in the mix, we also need

$$O \preceq (T_{\text{stk_base}}, \ldots) = \{ (n, (\Phi_S, \Phi_T)) | \forall i \leq n. \Phi_S \parallel^{T_{\text{stk_base}}} \Rightarrow \Phi_T \downarrow \}$$

$$O \succeq (T_{\text{stk_base}}, \ldots) = \{ (n, (\Phi_S, \Phi_T)) | \forall i \leq n. \Phi_T \parallel_i \Rightarrow \Phi_S \parallel^{T_{\text{stk_base}}} \}$$

Fig. 11. The observation approximations.

6While we present this as a logical relation, it is in fact two logical approximations. However, the only difference between the two approximations is in the observation approximation.


\( \mathcal{E}_{xjmp} \). It relates two pairs of capabilities by plugging them into related register-files and memories using the \( xjmpRes \) function and requiring that the result is in the observation relation.

Register files are related either by a trusted or untrusted relation \( \mathcal{R}_{\text{def}}(W) \), and they are related when they contain words related in the appropriate value relation. The register-file relation splits the ownership of \( W \) between the registers as linear capability need a spatially owned region; but to prevent aliasing, the they cannot depend on the same spatially owned region. The memory on oLCM consists of a heap \( ms_S \), a call stack \( stk \) and a free stack \( ms_{stk} \). This relates to a memory \( ms_T \) on LCM in a world \( W \) if we can split \( ms_T \) into three parts that correspond with the view oLCM provides on the memory. Each of the three partitions of \( ms_T \) must be related to one of the three parts of oLCM memory under the relevant sub-world of \( W \). The heap \( ms_S \) relates to the heap partition of \( ms_T \) in the standard way, i.e. the two memories can be partitioned, so they satisfy all the regions of the heap world. The free stack \( ms_{stk} \) also relates to the free stack partition of \( ms_T \) in the standard way but with respect to the free stack world. The call stack \( stk \) relates to the call stack partition of \( ms_T \) if it can be split into parts related to each of the stack frames. Further, as each of the stack frames corresponds to a slot in the return order none of the stack frames should be empty. It is the physical location of a stack frame that determines its order in the call stack, so the order must be preserved by the partitions of \( ms_T \).

### 5.1.4 Value relations

The purpose of the value relations is twofold: It relates the words on oLCM to the words they correspond to on LCM, and it defines what pairs of words are safe, i.e. are unable to break the guarantees of the capability machines including the LSE and WBCF guaranteed by oLCM. As previously explained, we try to ensure LSE and WBCF for some trusted code that we expect to behave reasonably, e.g. by handling return seals in a safe way. This means that it is safe to give return seals to trusted code; but at the same time, we have some code that we do not trust and want to guarantee LSE and WBCF against. This means that there is a difference between what is safe to give to the trusted code and the untrusted code which we capture by having a trusted and an untrusted value relation. The two value relations are sketched in Figure 12.

The untrusted value relation relates words from the two machines that can safely be given to untrusted code. Data is always safe, and a piece of data relates to itself. A stack pointer is related to a linear capability with the same (non-executable) permission, range of authority, and current address. A safe stack pointer with read permission should only be able to produce safe words, i.e. words in \( V_{\text{untrusted}} \), and a safe stack pointer with write permission can at least write safe words. Regions model the contents of memory, so to determine safety of a stack pointer there must be an appropriate spatially owned region in the free stack sub-world. Syntactically equal memory capabilities are related in a similar way to stack capabilities when they have read (and write) permission. The difference is that memory capabilities point to the heap rather than the free stack, so the regions they depend on are picked from the heap sub-world. An executable memory capability should always point to code which means that there should be a code region governing the memory it references. The trusted return seals and closure seals cannot be given safely to untrusted code, so a set of seals is only in the untrusted value relation when it is disjoint from those, and they may only be used to seal things in \( V_{\text{untrusted}} \). When a piece of untrusted code is called, it is passed a return pair consisting of two sealed capabilities. Calling untrusted code should be safe, so sealed capabilities must be related in the untrusted value relation. The seal interpretation function decides what a seal can be used for, so for a pair \( (\text{sealed}(\sigma, sc_S), \text{sealed}(\sigma, sc_T)) \) to be in the untrusted value relation, \( (sc_S, sc_T) \) must be in the seal interpretation function for \( \sigma \). Further, it should be safe to use the sealed capability, i.e. jump to it with \( xjmp \), which is exactly what \( \mathcal{E}_{xjmp} \) expresses, so when paired up with another pair from the seal interpretation function for \( \sigma \) the quadruple must be in \( \mathcal{E}_{xjmp} \).
\[ V_{\text{untrusted}}^{\square, gc}(W) = \{(n, (i, i)) \mid i \in \mathbb{Z}\} \cup \]
\[ \{(n, \text{stack-\text{ptr}(p, b, e, a)}, ((p, \text{linear}), b, e, a)) \mid \ldots\} \cup \]
\[ \{(n, \text{seal}(\sigma_b, \sigma_e, \sigma), \text{seal}(\sigma_b, \sigma_e, \sigma)) \mid [\sigma_b, \sigma_e] \subseteq (\overline{\sigma_{\text{ret}}} \cup \overline{\sigma_{\text{dom}}} \cup \ldots) \cup \]
\[ \{(n, \text{sealed}(\sigma, sc_S), \text{sealed}(\sigma, sc_T)) \mid \ldots\} \cup \]
\[ \{(n, (((p, l), b, e, a), ((p, l), b, e, a))) \mid \ldots\} \]
\[ V_{\text{trusted}}^{\square, gc}(W) = V_{\text{untrusted}}^{\square, gc}(W) \cup \]
\[ \{(n, \text{seal}(\sigma_b, \sigma_e, \sigma), \text{seal}(\sigma_b, \sigma_e, \sigma)) \mid [\sigma_b, \sigma_e] \subseteq (\overline{\sigma_{\text{ret}}} \cup \overline{\sigma_{\text{dom}}} \cup \ldots) \cup \]
\[ \{(n, (((p, \text{normal}), b, e, a), ((p, \text{normal}), b, e, a))) \mid p \leq \text{rx} \land \ldots\} \]

Fig. 12. Sketches of the trusted and untrusted value relation.

The trusted value relation contains everything that the untrusted value relation contains and a few extra things. Trusted code uses the return seals and the closure seals (in a responsible way), and therefore sets of seals with trusted return seals and closure seals are in the trusted value relation. In order for trusted code to even execute, it needs to use an executable memory capability for the trusted code, so such capabilities are in the trusted value relation.\(^7\)

5.1.5 **Fundamental theorem.** With the logical relation defined, we state the fundamental theorem of logical relations (FTLR).

**Lemma 1 (FTLR).** If \((n, [b, e]) \in \text{readXCondition}^{\square, gc}(W)\) and either \([b, e] \subseteq T_A\) and \(((\text{rx, normal}), b, e, a)\) behaves reasonably up to \(n\) steps or \([b, e] \# T_A\), then

\[(n, (((\text{rx, normal}), b, e, a), ((\text{rx, normal}), b, e, a))) \in \mathcal{E}^{\square, gc}(W) \]

Roughly speaking, the FTLR says: any executable capability used for execution in a safe environment won’t break the capability machine guarantees. This is, however, only true for syntactically well-formed code, which the \text{readXCondition}^{\square, gc} entails, and for trusted code the semantic reasonability condition must also be satisfied. The proof of the FTLR is done by a nested induction where each instruction that could be executed is proven safe. The lemma is not directly used in the proof sketch we present in Section 5.2, but it is crucial to the proofs of the lemmas in that section.

5.2 **Full abstraction proof sketch**

The logical relation presented in Section 5.1 is lifted to a component relation \(C\) (we will also use \(\equiv\) to denote this relation) and an executable configuration relation \(\mathcal{E}C\) in a straight forward manner. The component relation relates components to themselves which makes sense as well-formed components should contain no oLCM specific constructs. We prove a number of lemmas about the relations. The first is a FTLR for components that says (1) all well-formed untrusted components are related to themselves in the component relation, and (2) all well-formed and reasonable trusted components are related to themselves in the component relation.

**Lemma 2 (FTLR for components).** If \(\text{comp}\) is a well-formed component, i.e.\(\text{comp}\) and either \(\text{dom}(\text{comp.msc_{code}}) \subseteq T_A\) and \(\text{comp}\) is a reasonable component; or \(\text{dom}(\text{comp.msc_{code}}) \# T_A\), then there exists a \(W\) such that \((n, (\text{comp}, \text{comp})) \in C_{\square, gc}(W)\)

\(^7\)Note that code capabilities for trusted code is not allowed in the untrusted value relation as it could be used to read return seals that are not in the untrusted value relation.
6.1 Full abstraction

Our formulation of WBCF and LSE using a fully abstract overlay semantics has an advantage with respect to others that we haven’t discussed yet. 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 every stack frame 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 will need to behave like an atomic operation similar to a compare-and-swap instruction. This is in principle not a problem except that a compare-and-swap is significantly slower than a regular load (on the order of 10x slower or more). When using StkTokens, loads of linear capabilities normally 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, we are hopeful that well-engineered caching could reduce the high overhead of atomic loads of linear capabilities. If such memory could be (mostly) kept exclusively locked in a cache close to the processor, the overhead of atomic loads in StkTokens might be significantly less than load’s worst case. 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 information about a program's control graph and a 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 static control flow graph of a program [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.
REFERENCES


