2016 IEEE 29th Computer Security Foundations Symposium

Beyond Good and Evil

Formalizing the Security Guarantees of Compartmentalizing Compilation

Yannis Juglaret[1,2] Cătălin Hriţcu[1] Arthur Azevedo de Amorim[4] Boris Eng[1,3] Benjamin C. Pierce[4]

[1]Inria Paris [2]Université Paris Diderot (Paris 7) [3]Université Paris 8 [4]University of Pennsylvania

***Abstract*—Compartmentalization is good security-engineering practice. By breaking a large software system into mutually distrustful components that run with minimal privileges, restricting their interactions to conform to well-defined interfaces, we can limit the damage caused by low-level attacks such as control-flow hijacking. When used to defend against such attacks, compartmentalization is often implemented cooperatively by a compiler and a low-level compartmentalization mechanism. However, the formal guarantees provided by such *compartmentalizing compilation* have seen surprisingly little investigation.**

We propose a new security property, *secure compartmentalizing compilation (SCC)*, that formally characterizes the guarantees provided by compartmentalizing compilation and clarifies its attacker model. We reconstruct our property by starting from the well-established notion of fully abstract compilation, then identifying and lifting three important limitations that make standard full abstraction unsuitable for compartmentalization. The connection to full abstraction allows us to prove SCC by adapting established proof techniques; we illustrate this with a compiler from a simple unsafe imperative language with procedures to a compartmentalized abstract machine.

1 Introduction

Computer systems are distressingly insecure. Visiting a website, opening an email, or serving a client request is often all it takes to be subjected to a control-hijacking attack. These devastating low-level attacks typically exploit memory-safety vulnerabilities such as buffer overflows, use-after-frees, or double frees, which are abundant in large software systems. Various techniques have been proposed for guaranteeing memory safety [12], [19], [23], [42], [49]–[53], but the challenges of efficiency [52], [53], precision [67], scalability [71], backwards compatibility [15], and effective deployment [12], [19], [22], [23], [42], [49]–[51] have hampered their widespread adoption.

Meanwhile, new mitigation techniques have been proposed to deal with the most onerous consequences of memory unsafety—for instance, techniques aimed at preventing control-flow hijacking even in unsafe settings [2], [3], [24], [64]. Unfortunately, these defenses often underestimate the power of the attackers they may face [17], [24], [26], [27], [30], [63]—if, indeed, they have any clear model at all of what they are protecting against. Clarifying the precise security properties and attacker models of practical mitigation techniques is thus an important research problem—and a challenging one, since a good model has to capture not only the defense mechanism itself but also the essential features of the complex world in which low-level attacks occur.

In this paper we focus on the use of *compartmentalization* [13], [31], [66] as a strong, practical defense mechanism against low-level attacks exploiting memory unsafety. The key idea is to break up a large software system into mutually distrustful components that run with minimal privileges and can interact only via well-defined interfaces. This is not only good software engineering; it also gives strong security benefits. In particular, control-hijacking attacks can compromise only specific components with exploitable vulnerabilities, and thus only give the attacker direct control over the privileges held by these components. Also, because compartmentalization can be enforced by more coarse-grained mechanisms, acceptable efficiency and backwards compatibility are generally easier to achieve than for techniques enforcing full-blown memory safety.

When used as a defense mechanism against memory unsafety, compartmentalization is often achieved via cooperation between a compiler and a low-level compartmentalization mechanism [15], [31], [37], [40], [58], [66], [69]. In this paper we use *compartmentalizing compilation* to refer to cooperative implementations of this sort. The compiler might, for instance, insert dynamic checks and cleanup code when switching components and provide information about components and their interfaces to the low-level compartmentalizing mechanism, which generally provides at least basic isolation. Two such low-level compartmentalization technologies are already widely deployed: process-level privilege separation [13], [31], [38] (used, e.g., by OpenSSH [59] and for sandboxing plugins and tabs in modern web browsers [60]) and software fault isolation [65] (provided, e.g., by Google Native Client [68]); many more are on the drawing boards [12], [15], [32], [58], [66].

So what security guarantees does compartmentalizing compilation provide, and what, exactly, is its attacker model? A good starting point for addressing these questions is the familiar notion of *fully abstract compilation* [1], [4]–[6], [8], [10], [11], [29], [33], [55]. A fully abstract compiler toolchain (compiler, linker, loader, and underlying architecture with its security mechanisms) protects the interactions between a compiled program and its low-level environment, allowing the programmer to reason soundly about the behavior of their code when it is placed in an arbitrary target-language context, by considering only its behavior in arbitrary source-language contexts. In particular, if we link the code produced by such a compiler against arbitrary low-level libraries—

DOI 10.1109/CSF.2016.11

perhaps compiled from an unsafe language or even written directly in assembly—the resulting execution will not be any less secure than if we had restricted ourselves to library code written in the same high-level language as the calling program.

(Why is it useful to restrict attention to attackers written in a high-level language? First, because reasoning about what attackers might do—in particular, what privileges they might exercise—is easier in a high-level language. And second, because by phrasing the property in terms of low- and high-level programs rather than directly in terms of attacker behaviors, specific notions of privilege, etc., we can re-use the same property for many specific languages.)

Since full abstraction works by partitioning the world into a program and its context, one might expect it to apply to compartmentalized programs as well: some set of components that are assumed to be subject to control-hijacking attacks could be grouped into the "low-level context," while some others that are assumed to be immune to such attacks would constitute the "high-level program." Full abstraction would then allow us to reason about the possible behaviors of the whole system using the simplifying assumption that the attacker's injected behavior for the compromised components can be expressed in the same high-level language as the good components. Sadly, this intuition does not withstand closer examination. Full abstraction, as previously formulated in the literature, suffers from three important limitations that make it unsuitable for characterizing the security guarantees of compartmentalizing compilation.

First, fully abstract compilation assumes that the source language itself is secure, so that it makes sense to define target-level security with respect to the semantics of the source language. However, compartmentalization is often applied to languages like C and C++, which do *not* have a secure semantics—the C and C++ standards leave most of the security burden to the programmer by calling out a large number of *undefined behaviors*, including memory-safety violations, that are assumed never to occur. Valid compilers for these languages are allowed to generate code that does literally *anything*—in particular, anything a remote attacker may want—when applied to inputs that lead to undefined behavior. There is no way to tell, statically, whether or not a program may have undefined behavior, and compilers do not check for this situation. (Indeed, not only do they not check: they aggressively exploit the assumption of no undefined behaviors to produce the fastest possible code for well-defined programs, often leading to easily exploitable behaviors when this assumption is broken.) The point of compartmentalizing compilation is to ensure that the potential effects of undefined behavior are limited to the compromise of the component in which it occurs: other components can only be influenced by compromised ones via controlled interactions respecting specified interfaces.

To characterize the security of compartmentalizing compilation, we therefore need a formal property that can meaningfully accommodate source languages in which components can be compromised via undefined behavior. Full abstraction as conventionally formulated does not fit the bill, because, in order to preserve equivalences of programs with undefined behavior, compilers must abandon the aggressive optimizations that are the reason for allowing undefined behaviors in the first place. To see this, consider C expressions `buf[42]` and `buf[43]` that read at different positions outside the bounds of a buffer `buf`. These two programs are equivalent at the source level: they both lead to arbitrary behavior. However, a real C compiler would never compile these expressions to equivalent code, since this would require runtime checks that many C programmers would deem too expensive.

Second, fully abstract compilation makes an *open world* assumption about the attacker context. While the context is normally required to be compatible with the protected program, for instance by respecting the program's typed interface, the structure and privilege of the context are unrestricted (the definition quantifies over *arbitrary* low-level contexts). This comes in direct contradiction with the idea of least privilege, which is crucial to compartmentalization, and which relies on the fact that even if a component is compromised, it does not immediately get more privilege. Compromised components cannot change the basic rules of the compartmentalization game. For instance, in this paper we consider a static compartmentalization setting, in which the breakup of the application into components is fixed in advance, as are the privileges of each component. A security property suitable for this setting needs to only consider contexts that conform to a fixed breakup into components with static privileges.[1]

Third, because the definition of full abstraction involves applying the compiler only to a program and not to the untrusted context in which it runs, a fully abstract compiler may choose to achieve its protection goals by introducing just a single barrier around the trusted part to protect it from the untrusted part [8], [43], [54], [55], [57]. Such compilation schemes force the programmer to commit in advance to a single compromise scenario, i.e., to a single static split of their application into a "good" trusted program and an "evil" untrusted context from which this program has to be protected. This is not realistic in the setting of compartmentalizing compilation, where we generally cannot predict which components may be vulnerable to compromise by control hijacking attacks, and instead must simultaneously guard against multiple compromise scenarios. Compartmentalizing compilers allow us to build more secure applications that go beyond the blunt trusted/untrusted distinction made by some fully abstract compilers. To describe their guarantees accurately, we thus need a new property that captures the protection obtained by breaking up applications into multiple mutually distrustful components, each running with least privilege, and permits reasoning about multiple scenarios in which different subsets of these components are compromised.

[1] In a setting where new components can be dynamically created and privileges can be exchanged dynamically between components, the details of this story will be more complicated; still, we expect any secure compartmentalizing compilation property to limit the ability of low-level attacker contexts to "guess" the privileges of existing components.

Our main contribution is the definition of such a property, which we call *secure compartmentalizing compilation (SCC)* (§2). While similar in many respects to full abstraction, our property escapes the three limitations discussed above. First, it applies to unsafe source languages with undefined behaviors by introducing a new notion of *fully defined* sets of components. While undefined behavior is a property of whole programs, full definedness is compositional. Intuitively, a set of components is fully defined if they cannot be *blamed* [28] for undefined behavior in any context satisfying fixed interfaces. Second, SCC makes a *closed-world* assumption about compromised components, enforcing the basic rules of the compartmentalization game like the fixed division into components and the fixed privileges of each component, including for instance with which other components it is allowed to interact. Third, SCC ensures protection for multiple, mutually distrustful components; it does not assume we know in advance which components are going to be compromised (i.e., in the C setting, which components may contain exploitable undefined behaviors), but instead explicitly quantifies over all possible compromise scenarios.

Our second contribution is relating SCC to standard formulations of full abstraction both intuitively and formally (§3). We start from full abstraction and show how the three limitations that make it unsuitable in our setting can be lifted one by one. This results in two properties we call *structured full abstraction* and *separate compilation*, which can be combined and instantiated to obtain SCC. While our property directly captures the intuition of our attacker model, reducing it to structured full abstraction is a useful technical step, since the latter is easier to establish for specific examples using a variant of existing proof techniques. Moreover, arriving at the same property by two different paths increases our confidence that we found the right property.

Our third contribution is establishing the SCC property for a simple unsafe imperative language with components interacting via procedure calls and returns, compiling to an abstract machine with protected compartments (§4). Despite the simplicity of the setting, this result gives useful insights. First, the source language and compilation strategy enable interesting attacks on components with potential buffer overflows, similar to those found in C. Second, we illustrate how SCC can be achieved by the cooperation of a compiler (cleaning and restoring registers) and a low-level protection mechanism (totally isolating compartments and providing a secure interaction mechanism using calls and returns). Third, our SCC proof adapts a standard technique called *trace semantics* [35], [56], via the reduction to structured full abstraction. The closed-world assumption about the context made by structured full abstraction requires some nontrivial changes to the trace semantics proof technique.

The remainder of the paper describes each of our three contributions in detail (§2–§4) and closes by discussing related work (§5) and future directions (§6). The supplemental material associated with this paper includes: (a) a Coq proof for Theorem 3.4; (b) technical details and proofs for the SCC instance from §4; and (c) a trace mapping algorithm in OCaml using property-based testing to support Assumption 4.9. It can be found at `http://yannis.computer/papers/bge.html`.

2 Secure Compartmentalizing Compilation

We start with an intuitive explanation of compartmentalizing compilation, its attacker model, and its security benefits, and then introduce *secure compartmentalizing compilation (SCC)*.

We consider compartmentalization mechanisms provided by the compiler and runtime system for an unsafe programming language with some notion of components.[2] In §4 we will present a simple example in detail, but for the present discussion it suffices to think informally of C or C++ enriched with some compartmentalization mechanism. This mechanism allows security-conscious developers to break large applications into mutually distrustful components running with least privilege and interacting only via well-defined interfaces. We assume that the interface of each component also gives a precise description of its privilege. Our notion of interface here is quite generic: interfaces might include any information that can be dynamically enforced on components, including module signatures, lists of allowed system calls, or more detailed access control specifications describing legal parameters to inter-component calls (e.g., ACLs for files). We assume that the division of the application into components and the interfaces of those components are statically determined and fixed throughout execution. In §4, we instantiate this picture with a rather simple and rigid notion of components and interfaces, where components don't directly share any state and where the only thing one component can do to another one is to call the procedures allowed by the interfaces of both components.

We do not fix a specific compartmentalizing compilation mechanism; we just assume that whatever mechanism is chosen can guarantee that, even if one component is compromised (e.g., by a control-hijacking attack), it will still be forced to adhere to its specified interface in its interactions with other components. What a compromised component *can* do in this model is use its access to other components, as allowed by its interface, to trick them into misusing their own privileges (confused deputy attacks) and/or attempt to mount further control-hijacking attacks on other components by communicating with them via defined interfaces.

We do not assume we know in advance which components will be compromised: the compartmentalizing compilation mechanism has to protect each component from all the others. This allows developers to reason informally about various compromise scenarios and their impact on the security of the whole application [31], relying on conditional reasoning of the form: "If *these* components get taken over and *these* do not, then *this* might happen (while *that* cannot), whereas if these other components get taken over, then this other

[2] We use the term "runtime system" loosely to include operating system mechanisms [13], [31], [38], [59], [60] and/or hardware protections [12], [32], [58], [66] that may be used by the compiler.

thing might happen..." If the practical consequences of some plausible compromise scenario are too serious, developers can further reduce or separate privilege by narrowing interfaces or splitting components, or they can make components more defensive by dynamically validating the inputs they receive from other components.

For instance, developers of a compartmentalized web browser [60] might reason about situations in which some subset of plugins and tabs gets compromised and how this might impact the browser kernel and the remaining plugins and tabs. A possible outcome of this exercise might be noticing that, if the browser kernel itself is compromised, then all bets are off for all the components and the application as a whole, so the developers should put extra energy in defending the kernel against attacks from compromised plugins or tabs. On the other hand, if interfaces *between* tabs and plugins are appropriately limited, then compromise of one should not disrupt the rest.

Our goal is to articulate a security property that supports reasoning about multiple compromise scenarios and clarifies the associated attacker model. At the same time, our property is intended to serve as a benchmark for developers of compartmentalizing compilation mechanisms who want to argue formally that their mechanisms are secure. In the rest of this section we explain the technical ideas behind the SCC property and then give its formal definition.

An *application* is a set *Cs* of *components*, with corresponding *interfaces CIs*. These components are separately compiled (individually compiling each component in the set *Cs* is written $\mathit{Cs}{\downarrow}$) and linked together (written $\bowtie(\mathit{Cs}{\downarrow})$) to form an executable binary for the application.

SCC quantifies over all *compromise scenarios*—i.e., over all ways of partitioning the components into a set of compromised ones and a set of uncompromised ones. In order to ensure that the set of compromised components doesn't expand during evaluation, we require that the uncompromised components be *fully defined* with respect to the interfaces of the compromised components. That is, the uncompromised components must not exhibit undefined behaviors even if we replace the compromised components with arbitrary code (obeying the same interfaces).

The full definedness condition is a necessary part of the *static compromise model* considered in this paper. Intuitively, if an uncompromised component can be tricked into an undefined behavior by interface-respecting communication with other components, then we need to conservatively assume that the already compromised components will succeed in compromising this component dynamically, so it belongs in the set of compromised components from the start. This static model is much simpler to reason about than a model of dynamic compromise, in which one could perhaps provide guarantees to not-fully-defined components up to the point at which they exhibit undefined behavior, but which could, however, invalidate standard compiler optimizations that involve code motion. Moreover, it seems highly nontrivial to define our property for such a more complex model.

Figure 1 illustrates one way to partition five components $C_1, \ldots, C_5$ with interfaces $i_1, \ldots, i_5$, representing the scenario where C_2, C_4, and C_5 are compromised and C_1 and C_3 are not. In order for this compromise scenario to be considered by our property, C_1 and C_3 need to be fully defined with respect to interfaces i_2, i_4, and i_5, which means C_1 and C_3 cannot cause undefined behaviors when linked with any components B_2, B_4, B_5 satisfying interfaces i_2, i_4, i_5.

Formally, full definedness is a language-specific parameter to our definition of SCC, just as the program equivalence relations are language-specific parameters to both SCC and vanilla full abstraction. For instance, in the simple imperative language in §4, we will say that components *Cs* are fully defined with respect to a set of adversary interfaces *BIs* if, for all components *Bs* satisfying *BIs*, the complete program $\bowtie(\mathit{Cs}\downarrow \cup\ \mathit{Bs}\downarrow)$ cannot reduce to a stuck non-final state (corresponding to undefined behavior) where the currently executing component is one of the ones in *Cs* (i.e., no component in *Cs* can be "blamed" [28] for undefined behavior). Full definedness might well be defined differently for another language; for instance, in a concurrent language undefined behaviors cannot be as easily modeled by stuckness since normally other threads can proceed even if one of the threads is stuck. One last thing to note is that full definedness of a set of components is generally a much weaker property than the full definedness of each individual component in the set. Since the interfaces of the adversary components *BIs* can (and in §4 do) restrict not only the operations they export but also the operations they import from *Cs*, the components in the set can export dangerous operations just to other components in the set; the components actually in the set might then all use these operations properly, whereas arbitrary components with the same interfaces could abuse them to trigger undefined behaviors.

SCC states that, *in all such compromise scenarios*, the compiled compromised components must not be able to cause more harm to the compiled uncompromised components via low-level attacks than can be caused by some high-level components written in the source language. Basically this means that any low-level attack can be mapped back to a high-level attack by compromised components satisfying the given interfaces. The property additionally ensures that the high-level components produced by this "mapping back" are fully defined with respect to the interfaces of the uncompromised components. So with SCC, instead of having to reason about the low-level consequences of undefined behavior in the compromised components, we can reason in the source language and simply replace the compromised components by equivalent ones that are guaranteed to cause no undefined behavior.

Formally, SCC is stated by quantifying over multiple distinguishability games, one for each compromise scenario, where the individual games are reminiscent of full abstraction. The goal of the attacker in each game is to distinguish between two variants of the uncompromised components. Figure 2 illustrates these two variants as C_1, C_3 and D_1, D_3, where we use $\not\sim_H$ and $\not\sim_L$ to indicate that the behaviors of two (high-

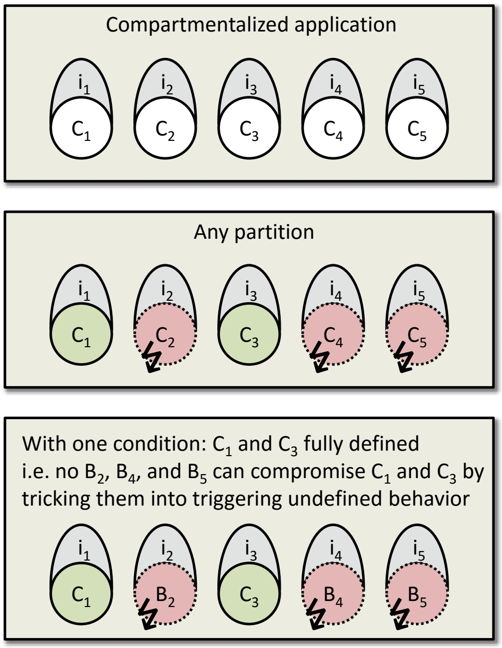

Figure 1. Compromise scenarios

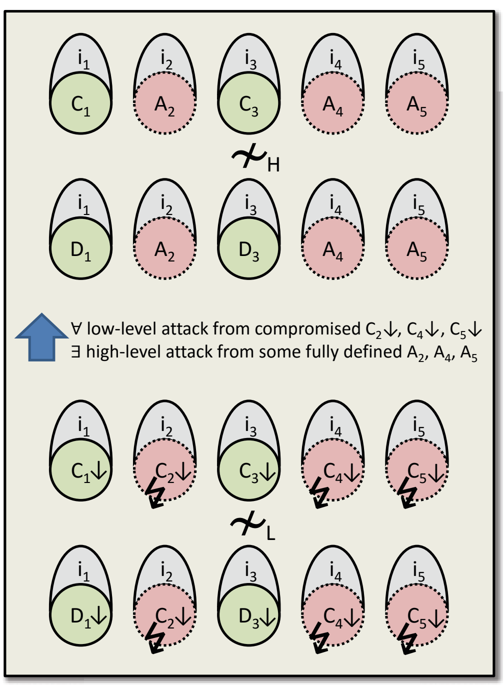

Figure 2. SCC distinguishability game, for one of the compromise scenarios

or low-level) complete programs are distinguishable, i.e., they produce different observable outcomes when executed. For this compromise scenario, SCC specifies that, if compiled compromised components $C_2\!\downarrow, C_4\!\downarrow, C_5\!\downarrow$ can distinguish the $C_1\!\downarrow, C_3\!\downarrow$ and $D_1\!\downarrow, D_3\!\downarrow$ variants at the low level, then there must exist some (fully defined) components A_2, A_4, A_5 that distinguish C_1, C_3 and D_1, D_3 at the high level.

With all this in mind, the SCC property is formally expressed as follows:

Definition 2.1 (SCC).

- For any complete compartmentalized program and for all ways of *partitioning* this program into a set of *uncompromised* components *Cs* and their interfaces *CIs*, and a set of *compromised* components *Bs* and their interfaces *BIs*, so that *Cs* is fully defined with respect to *BIs*, and
- for all ways of replacing the uncompromised components with components *Ds* that satisfy the same interfaces *CIs* and are fully defined with respect to *BIs*,
- if $\bowtie(Cs\!\downarrow \cup\, Bs\!\downarrow) \not\sim_L \bowtie(Ds\!\downarrow \cup\, Bs\!\downarrow)$,
- then there exist components *As* satisfying interfaces *BIs* and fully defined with respect to *CIs* such that $\bowtie(Cs \cup As) \not\sim_H \bowtie(Ds \cup As)$.

As suggested before, our property applies to any *fully defined* sets of components *Cs* and *Ds* (which cannot be dynamically compromised by some components with interfaces *BIs*). We conjecture that this full definedness precondition is strictly required in the static corruption model we are assuming. It is worth noting that we are not proposing any method for proving that programs are fully defined; this comes with the territory when dealing with C-like languages. What we are after is bringing formal foundations to *conditional* reasoning of the form “if these *Cs* are fully defined and the remaining components *Bs* get compromised, then...”

Note that the *Bs* in our SCC definition need not be fully defined—i.e., the property allows the compromised components to contain undefined behaviors (this may well be why they are compromised!) and promises that, even if they do, we can find some other components *As* that are able to distinguish between *Cs* and *Ds* in the source language without causing any undefined behaviors. Indeed, for those compromise scenarios in which *Bs* are already fully defined, our SCC property trivially follows from correct compilation (Assumption 4.2) since in that case we can always pick $As = Bs$.

This generic property is parameterized over a source and a target language with a notion of component for each, source- and target-level notions of linking sets of components ($\bowtie$), source- and target-level notions of distinguishability ($\not\sim$), a compiler mapping source components to target components ($\downarrow$), a source-level notion of interface and an interface satisfaction relation (lifted to sets of components and interfaces), and a notion of a set of components *Cs* being fully defined with respect to a set of adversary interfaces *BIs*.

3 From Full Abstraction to SCC

§2 presented SCC by directly characterizing the attacker model against which it defends. In this section we step back and show how SCC can instead be obtained by starting from the well-established notion of full abstraction and removing each of the three limitations that make it unsuitable in our setting. This results in two properties, *structured full abstraction* and *separate* *compilation*, which we then combine and instantiate to obtain SCC. This reduction is not only theoretically interesting, but also practically useful, since structured full abstraction can more easily be proved by adapting existing proof techniques, as we will see in §4.

Full abstraction A *fully abstract* compiler protects compiled programs from their interaction with unsafe low-level code and thus allows sound reasoning about security (and other aspects of program behavior) in terms of the source language. Fully abstract compilation [1] intuitively states that no low-level attacker can do more harm to a compiled program than some program in the source language already could. This strong property requires enforcing all high-level language abstractions against arbitrary low-level attackers.

Formally, full abstraction is phrased as a distinguishability game requiring that low-level attackers have no more distinguishing power than high-level ones.

Definition 3.1. We call a compilation function (written $\downarrow$) *fully abstract* if, for all P and Q,

$$(\forall A.\ A[P] \sim_H A[Q]) \Rightarrow (\forall a.\ a[P\downarrow] \sim_L a[Q\downarrow]).$$

Here, P and Q are partial programs, A is a high-level context whose job is to try to distinguish P from Q, and a is a low-level "attacker context" that tries to distinguish $P\downarrow$ from $Q\downarrow$. The relations $\sim_L$ and $\sim_H$ are parameters to the definition, representing behavioral equivalence at the two levels. To be useful, they should allow the context to produce an observable action every time it has control, letting it convert its knowledge into observable behaviors. For instance, a common choice for behavioral equivalence is based on termination: two deterministic programs are behaviorally equivalent if they both terminate or both diverge.

When stated this way (as an implication rather than an equivalence), full abstraction is largely orthogonal to compiler correctness [41], [44]. While compiler correctness is about preserving behaviors when compiling from the source to the target, proving full abstraction requires some way to map each distinguishing context target to a sourge-level one, which goes in the opposite direction. This is easiest to see by looking at the contrapositive:

$$\forall a.\ a[P\downarrow] \not\sim_L a[Q\downarrow] \Rightarrow \exists A.\ A[P] \not\sim_H A[Q]$$

Problem 1: Undefined behavior The first limitation of full abstraction is that it cannot realistically be applied to compiling from an unsafe language with undefined behaviors. Undefined behaviors are (arbitrarily!) nondeterministic, and no realistic compiler can preserve this nondeterminism in the target as required by full abstraction. (Removing it from the source language would negate the performance and optimization benefits that are the reason for allowing undefined behaviors in the first place.)

To adapt full abstraction to a source language with undefined behaviors, we need to restrict attention only to *defined* complete programs in the source language. And even with this restriction, defining full abstraction still requires a little care. For instance, the following variant is wrong (formally, *defined* is another parameter to this property):

$$(\forall A.\ A[P] \text{ and } A[Q] \textit{ defined } \Rightarrow A[P] \sim_H A[Q]) \quad\Rightarrow \quad (\forall a.\ a[P\downarrow] \sim_L a[Q\downarrow])$$

Any P and Q that both trigger undefined behavior as soon as they get control would be considered equivalent in the high-level language because there is no context that can make these programs defined while observing some difference between them. All such programs would thus need to be equivalent at the low level, which is clearly not the case (since their nondeterminism can be resolved in different ways by the compiler). The problem here is that if P and Q trigger undefined behavior then the context often cannot make up for that and make the program defined in order be able to cause an observation that distinguishes P and Q.

Solution 1: Full abstraction for unsafe languages The responsibility of keeping $A[P]$ defined should be thus shared between A and P. For this we assume a compositional notion of *fully defined* behavior for programs and contexts as two parameters to Definition 3.2 below. We require that these parameters satisfy the following properties: (1) a program is fully defined if it does not cause undefined behavior in any fully defined context, and (2) a context is fully defined if it does not cause undefined behavior when we plug any fully defined program into it. Note that properties (1) and (2) are circular and therefore cannot be used as the definition of full definedness. For specific languages (e.g., the one in §4) we can break this circularity and define full definedness using *blame* [28]: intuitively we call a partial program *fully defined* when it cannot be blamed for undefined behavior in any context whatsoever. Similarly, we call a context fully defined when it cannot be blamed for undefined behavior for any program that we plug into it. Such a blame-based definition satisfies the properties (1) and (2) above. Full definedness allows us to introduce a new variant of full abstraction that applies to unsafe source languages with undefined behavior:

Definition 3.2 (Full abstraction for unsafe languages). We call a compiler $\downarrow$ for an unsafe language *fully abstract* if for all *fully defined* partial programs P and Q

$$(\forall A.\ A \textit{ fully defined} \Rightarrow A[P] \sim_H A[Q]) \quad\Rightarrow \quad (\forall a.\ a[P\downarrow] \sim_L a[Q\downarrow]).$$

Requiring that P, Q, and A are fully defined means that we can safely apply $\sim_H$ to $A[P]$ and $A[Q]$, because neither the programs nor the context can be blamed for undefined behavior. This property is incomparable with the original definition of full abstraction. Looking at the contrapositive,

$$\forall P, Q \textit{ fully defined}. \qquad (\exists a.\ a[P\downarrow] \not\sim_L a[Q\downarrow]) \Rightarrow (\exists A.\ A \textit{ fully defined} \wedge A[P] \not\sim_H A[Q]),$$

the P, Q *fully defined* pre-condition makes this weaker than full abstraction, while the A *fully defined* post-condition makes it stronger. The post-condition greatly simplifies reasoning about programs by allowing us to replace reasoning about low-level contexts with reasoning about high-level contexts *that cannot cause undefined behavior*.

One might wonder whether the P, Q *fully defined* pre-condition is too restrictive, since full definedness is a rather strong property, requiring each component to be very defensive about validating inputs it receives from others. In the static compromise model inherent to full abstraction and without additional restrictions on the program's context, we must be conservative and assume that, if any context can cause undefined behavior in a program, it can compromise it in a way that the compiler can provide no guarantees for this program. The structured full abstraction definition below will in fact restrict the context and thus use a weaker notion of full definedness. Moreover, separate compilation will allow us to quantify over all splits of a program into a fully defined partial program and a compromised context, which also makes the presence of the full definedness pre-condition more palatable.

Problem 2: Open-world assumption about contexts While full abstraction normally requires the contexts to be compatible with the partial program, for instance by respecting the partial program's typed interface, these restrictions are minimal and do not restrict the shape, size, exported interface, or privilege of the contexts in any way. This *open world* assumption about contexts does not fit with our compartmentalization setting, in which the breakup of the application into components is fixed in advance, as are the interfaces (and thus privileges) of all the components. In our setting, the definition of full abstraction needs to be refined to track and respect such structural constraints; otherwise a low-level context with 2 components might be mapped back to a high-level context with, say, 3 components that have completely different interfaces, and thus privileges. In particular, the high-level components' interfaces could give them more privileges than the low-level components had, increasing their distinguishing power.

Solution 2: Structured full abstraction We therefore introduce a structured variant of full abstraction, in which partial programs (indicated by $\bullet$ below) and contexts ($\circ$) are assigned dual parts of predefined complete program *shapes*. A shape might be anything, from a division into components with their interfaces (as in Theorem 3.4 below), to, e.g., the maximum size of a component's code after compilation (which might expose component sizes in a setting where it's too costly to hide them by padding to a fixed maximum size [58]).

Definition 3.3 (Structured full abstraction).
We say that a compiler $\downarrow$ for an unsafe language satisfies *structured full abstraction* if, for all *program shapes* s and partial programs $P \in^\bullet s$ and $Q \in^\bullet s$ so that P and Q are *fully defined* with respect to contexts of shape $\circ s$,

$$\left(\begin{array}{l}\forall A \in^\circ s. \quad A \textit{ fully defined} \text{ wrt. programs of shape } \bullet s \\ \hfill \Rightarrow A[P] \sim_H A[Q]\end{array}\right)$$
$$\Rightarrow (\forall a \in^\circ s.\ a[P\downarrow] \sim_L a[Q\downarrow]).$$

This property universally quantifies over any complete program shape s and requires that $P \in^\bullet s$ (read "program P has shape s"), $Q \in^\bullet s$, and $A \in^\circ s$ ("context A matches programs of shape s"). Moreover, the property only requires programs that are fully defined with respect to contexts of the right shape, and dually it only considers contexts that are fully defined with respect to programs of the right shape.

Recovering secure compartmentalizing compilation SCC can be recovered in a natural way as an instance of structured full abstraction (Definition 3.3). For both source and target languages, we take partial programs and contexts be sets of components and context application be set union. Compilation of sets of components works pointwise. To obtain an instance of structured full abstraction we additionally take shapes to be sets of component interfaces, where each interface is marked as either compromised or uncompromised.

Theorem 3.4. For any deterministic target language and any source language that is deterministic for defined programs, structured full abstraction instantiated to components as described above implies SCC.

Proof. Straightforward, though tedious. A machine-checked Coq proof can be found in the auxiliary materials. □

Problem 3: Statically known trusted/untrusted split While SCC can deal with multiple compromise scenarios, not all instances of structured full abstraction can. In general, if a compiler satisfies (structured) full abstraction, how can we know whether it can deal with multiple compromise scenarios, and what does that even mean? While we can instantiate full abstraction to a *particular* compromise scenario by letting the partial program P contain the uncompromised components and the low-level context a contain the compromised ones, a fully abstract compiler (together with its linker, loader, runtime etc.) might exploit this static split and introduce only one single barrier protecting the uncompromised components from the compromised ones. When presented with a different compromise scenario for the same program, the compiler could adapt and produce a different output.

The source of confusion here is that a fully abstract compiler does not need to compile contexts—only programs. In fact, even the *types* of contexts and of partial programs might well be completely different (e.g., the types of lambda calculus contexts and terms are different; a compiler for one cannot compile the other). Even when the types do match so that we can apply the same compiler to the context, the low-level context-application operation $A\downarrow[P\downarrow]$ can freely exploit the fact that its first argument is a compiled untrusted context and its second argument is a compiled trusted program that should be protected from the context. So if we start with a complete high-level program C and look at two different compromise scenarios $C = A_1[P_1]$ and $C = A_2[P_2]$, compiling each of the parts and combining the results using context application does not necessarily yield the same result (i.e., it could well be that $A_1\!\Downarrow[P_1\!\Downarrow] \neq A_2\!\Downarrow[P_2\!\Downarrow]$) or indeed even behaviorally equivalent results (i.e., it could well be that $A_1\!\Downarrow[P_1\!\Downarrow] \not\sim_L A_2\!\Downarrow[P_2\!\Downarrow]$). This means that the user of a fully abstract compiler may need to commit *in advance* to a single compromise scenario.

This weakness significantly limits the applicability of full abstraction. After all, uncertainty about sources of vulnerability is precisely the motivation for compartmentalizing compilation: if we knew which components were safe and which were not, there would be no reason to distinguish more than two levels of privilege, and we could merge each group into a single mega-component. Even in rare cases where we are certain that some code cannot be compromised—for instance because we have proved it safe—protecting only the verified code from all the rest using a fully abstract compiler [7] is still suboptimal in terms of protection, since it provides no guarantees for all the code that is not verified.

Moreover, this weakness is not hypothetical: several fully abstract compilers proposed in the literature are only capable of protecting a single trusted module from its untrusted context [8], [43], [54], [55], [57] (recently proposed extensions [58] do aim at lifting this restriction in some cases). While this setup is appropriate when all one wants to achieve is protecting trusted (e.g., verified) code from its untrusted context [7], it is not suitable for a compartmentalization setting where we do not know in advance which components will be dynamically compromised and which ones not, so that we want to simultaneously protect against all possible compromise scenarios.

Solution 3: Separate compilation We can address this by requiring that the compiler toolchain have one additional property:

Definition 3.5. We say that the compiler toolchain (i.e., the compiler $-\downarrow$, the linker $-[-]$, and the runtime system embodied in the low-level behavioral equivalence) satisfies *separate compilation* if

1) the type of contexts and programs is the same (so that the compiler can also compile contexts), and
2) $(A[P])\downarrow \sim_L A\downarrow[P\downarrow]$ for all A and P.

Requiring that context application and compilation commute (condition 2) implies that, if some complete program C can be written as both $C = A_1[P_1]$ and $C = A_2[P_2]$, then separately compiling each of these splits yields behaviorally equivalent results: $(A_1[P_1])\downarrow \sim_L (A_2[P_2])\downarrow$. With separate compilation, full abstraction for an unsafe language (Definition 3.2) can be instantiated as follows:

$$\begin{aligned}\forall B.\ \forall P,Q \textit{ fully defined}. \quad & ((B[P])\downarrow \not\sim_L (B[Q])\downarrow) \\ & \Rightarrow (\exists A.\ A \textit{ fully defined} \wedge A[P] \not\sim_H A[Q])\end{aligned}$$

One compelling reading of this is that, for all compromise scenarios (ways to break a complete program into a compromised context B and an uncompromised program P), and for all programs Q that we can substitute for P, if the context B can distinguish P from Q when compiled to low-level code, then there exists a fully defined context A that can distinguish them at the high-level.

In a language without undefined behavior, this property would trivially follow just from (whole program) correct compilation (see Assumption 4.2 below) by picking $A = B$. However, it is nontrivial for a language in which context B might cause undefined behavior, since then correct compilation does not apply for $B[P]$ and $B[Q]$. In our setting, this property allows us to avoid reasoning about the low-level implications of undefined behavior in a low-level context and instead consider just fully defined high-level contexts.

It is trivial to check that our instance of structured full abstraction from Theorem 3.4 does satisfy separate compilation. It should also be easy to show that many previous fully abstract compilers [8], [43], [54], [55], [57] do not satisfy separate compilation, since they were not designed to support a setting of mutual distrust.

4 A Simple Instance of SCC

In this section, we illustrate the main ideas behind SCC with a proof-of-concept compiler from an unsafe language with components to an abstract machine with compartments. We discuss key design decisions for providing secure compartmentalization, such as cleaning register values to prevent unintended communication between compartments. We also explain how a compiler optimization for component-local calls makes unwary compiled components vulnerable to realistic control-flow hijacking attacks. Finally, we show how to adapt a standard full abstraction proof technique called *trace semantics* [55] to prove SCC.

In the following, an assumption denotes a property that we believe is true and rely on, but which we haven't proved. Lemmas, theorems and corollaries denote properties that we have proved, possibly relying on some of the stated assumptions.

Source Language We work with an unsafe source language with components, procedures, and buffers. A program in this language is a set of components communicating via procedure calls. Buffer overflows have undefined behavior and may open the door to low-level attacks after compilation. However, thanks to the cooperation between the low-level compartmentalization mechanism and the compiler, the effects of these attacks will be limited to the offending component.

Components have statically checked interfaces that specify which procedures they import and export. To satisfy an interface, a component can only call external procedures that it imports explicitly, and it must define all procedures exported by its interface. Thus, interfaces define privileges by preventing components from calling non-imported procedures, and enable components to define private procedures (that are not exported in their interfaces). We will use the same notion of interfaces in our target abstract machine.

The syntax of expressions, given below, is that of a standard imperative language with mutable buffers and mutually recursive procedures. Each component C has local procedures "$C.P$" and private local buffers b. Loops are encoded using recursive calls, sequencing is encoded as a binary operation, and variables are encoded using buffers. Procedures take a single argument, which by convention is always passed in the first cell of the first buffer of the callee component. The only first class values are integers i; these can be passed across component boundaries using procedure calls and returns. Buffers and procedures are second class.

$$e ::= i \mid e_1 \otimes e_2 \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid b\texttt{[}e\texttt{]} \mid b\texttt{[}e_1\texttt{] := } e_2 \mid C\texttt{.}P(e) \mid \texttt{exit}$$

where $\otimes \in \{;, +, -, \times, =, \leq, \ldots\}$.

We define a standard continuation-based small-step semantics that reduces configurations *cfg*. It is deterministic for programs that don't cause undefined behavior.

$$cfg ::= (C, s, \sigma, K, e) \qquad K ::= \texttt{[]} \mid E\texttt{::}K$$

$$E ::= \square \otimes e_2 \mid i_1 \otimes \square \mid \texttt{if } \square \texttt{ then } e_1 \texttt{ else } e_2 \mid b\texttt{[}\square\texttt{] := } e_2 \mid b\texttt{[}i_1\texttt{] := } \square \mid C\texttt{.}P(\square)$$

A configuration (C, s, σ, K, e) represents a call in progress within component C, in which e is the expression being reduced and K is the continuation for this expression, up to the latest procedure call. Continuations are evaluation contexts, here represented as lists of flat evaluation contexts E. We denote by s a global state recording the values of the local buffers for each component. Continuations for pending calls are stored on a call stack σ, together with their call arguments' values and the names of the compartments they execute in. We omit the obvious definitions for call stacks σ and states s.

Evaluation starts as a call to a fixed procedure of a fixed main component, and completes once this call completes, or whenever the current expression e is `exit`. We illustrate the small-step semantics with the three rules that deal with procedure call evaluation. In these rules, Δ is a mapping from procedure identifiers to procedure bodies.

$$\frac{s' = s[C', 0, 0 \mapsto i] \qquad \sigma' = (C, s[C, 0, 0], K)\texttt{::}\sigma}{\Delta \vdash (C, s, \sigma, C'.P'(\square)\texttt{::}K, i) \to (C', s', \sigma', \texttt{[]}, \Delta[C', P'])}$$

$$\frac{s' = s[C', 0, 0 \mapsto i']}{\Delta \vdash (C, s, (C', i', K)\texttt{::}\sigma, \texttt{[]}, i) \to (C', s', \sigma, K, i)}$$

$$\frac{}{\Delta \vdash (C, s, \sigma, K, C'.P'(e)) \to (C, s, \sigma, C'.P'(\square)\texttt{::}K, e)}$$

As shown on the right-hand side of the first rule, a call starts with an empty continuation and the procedure body $\Delta[C', P']$ as the current expression. The first cell in the first buffer of the callee compartment is updated with the call argument, while information about the caller's state when performing the call gets pushed on the call stack σ. A call completes once an empty continuation is reached and the current expression is a value, as is the case in the left-hand side of the second rule. In this case, the caller's state is restored from the call stack, and execution resumes with the call result i as the current expression. The intermediate steps between the start and the end of a call reduce the procedure body to a value, as the last rule illustrates: Whenever e is not a value, reduction deconstructs e into a subexpression e' and a flat evaluation context E such that $e = E[e']$, where $E[e']$ means filling the hole $\square$ in E with e'. This expression e' becomes the new currently reduced expression, while E gets appended on top of the current continuation K. Finally, when e is a value i and the call has not completed ($K \neq \texttt{[]}$), the next step is chosen based on the flat evaluation context found on top of K, which gets removed from K. In the left-hand side of the first rule, for example, this flat evaluation context is $C'.P'(\square)$, for which the next chosen step, as shown on the right-hand side, is to start a procedure call to $C'.P'$, using i as the call argument.

Since undefined behaviors are allowed to take the machine to an arbitrary *low-level* state, it wouldn't make much sense to try to make the source-language semantics describe what can happen if an undefined point is reached. We therefore model them at the source level simply as stuckness (as done for instance in CompCert [45]). In particular, reduction gets stuck when trying to access or update a buffer out of bounds, and the type safety theorem says that well-formed programs can only go wrong (get stuck) by reducing to an out-of-bounds operation on a buffer. A program is well-formed if all the used buffers are defined, all imported components are defined, all imported external procedures are public, and if the names of all components are unique. Well-formedness extends straightforwardly to configurations.

Theorem 4.1 (Partial type safety). For any well-formed configuration $cfg = (C, s, \sigma, K, e)$, one of the following holds:

(1) *cfg* is a final configuration (either e is `exit` or else it is a value and K and σ are both empty);
(2) *cfg* reduces in one step to a well-formed configuration;
(3) *cfg* is stuck and has one of the following forms:
   (a) $(C, s, \sigma, b\texttt{[}\square\texttt{]} :: K, i)$ where $s[C, b, i]$ is undefined;
   (b) $(C, s, \sigma, b\texttt{[}i\texttt{]:=}\square\texttt{::}K, i')$ where $s[C, b, i]$ is undefined.

In the following, we use the term *undefined behavior configurations* for the configurations described in (3), and we say that a well-formed program is *defined* if reducing it never reaches an undefined behavior configuration.

Target Our compiler targets a RISC-based abstract machine extended with a compartmentalization mechanism, inspired by a similar design featured in previous work [12]. Each compartment in this target has its own private memory, which cannot be directly accessed by others via loads, stores, or jumps. Instead, compartments must communicate using special call and return instructions, which, as explained below, include checks to ensure that the communication respects compartment interfaces. (Note that this scheme requires a protected call stack, which, in a real system, could be implemented e.g., using a shadow call stack [3], [25] or return capabilities [37].) Because resource exhaustion and integer overflow issues are orthogonal to our present concerns, we assume that words are unbounded and memory is infinite.

The instruction set for our machine is mostly standard.

$$\begin{aligned} instr ::= \;& \mathsf{Nop} \mid \mathsf{Const}\ i \to r_d \mid \mathsf{Mov}\ r_s \to r_d \\ & \mid \mathsf{Load}\ {}^*r_p \to r_d \mid \mathsf{Store}\ {}^*r_p \leftarrow r_s \\ & \mid \mathsf{Jump}\ r \mid \mathsf{Jal}\ r \mid \mathsf{Call}\ C\ P \mid \mathsf{Return} \\ & \mid \mathsf{Binop}\ r_1 \otimes r_2 \to r_d \mid \mathsf{Bnz}\ r\ i \mid \mathsf{Halt} \end{aligned}$$

$\mathsf{Const}\ i \to r_d$ puts an immediate value i into register r_d. $\mathsf{Mov}\ r_s \to r_d$ copies the value in r_s into r_d. $\mathsf{Load}\ {}^*r_p \to r_d$

and Store $^*r_p \leftarrow r_s$ operate on the memory location whose address is stored in r_p (the $*$ in the syntax of Load and Store indicates that a pointer dereference is taking place), either copying the value found at this location to r_d or overwriting the location with the content of r_s. Jump r redirects control flow to the address stored in r. Jal r (jump-and-link) does the same but also communicates a return address in a dedicated register r_{ra}, so that the target code can later resume execution at the location that followed the Jal instruction. Call $C\ P$ transfers control to compartment C at the entry point for procedure "$C.P$". Return $C\ P$ transfers control back to the compartment that called the current compartment. Binop $r_1 \otimes r_2 \rightarrow r_d$ performs the mathematical operation $\otimes$ on the values in r_1 and r_2 and writes the result to r_d. Finally, Bnz $r\ i$ (branch-non-zero) is a conditional branch to an offset i, which is relative to the current program counter. If r holds anything but value zero, the branching happens, otherwise execution simply flows to the next instruciton.

While Jal is traditionally used for procedure calls and Jump for returns, in this machine they can only target the current compartment's memory. They are nonetheless useful for optimizing compartment-local calls, which need no instrumentation; in a realistic setting, the instrumented primitives Call and Return would likely come with monitoring overhead.

In the low-level semantics, we represent machine states *state* as $(C, \sigma, \mathit{mem}, \mathit{reg}, \mathit{pc})$ where C is the currently executing compartment, *mem* is a partitioned memory, *reg* is a register file, *pc* is the program counter, and σ is a global protected call stack. We assume a partial function *decode* from words to instructions. We write $\psi; E \vdash \mathit{state} \rightarrow \mathit{state}'$ to mean that *state* reduces to *state'* in an environment where component interfaces are given by ψ and component entry points by E. Here are the reduction rules for Call and Return:

$$\frac{\begin{array}{c}\mathit{mem}[C,\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Call}\ C'\ P' \quad \mathit{pc}' = E[C'][P'] \\ C' = C\ \vee\ C'.P' \in \psi[C].\mathit{import} \quad \sigma' = (C, \mathit{pc}{+}1) :: \sigma\end{array}}{\psi; E \vdash (C, \sigma, \mathit{mem}, \mathit{reg}, \mathit{pc}) \rightarrow (C', \sigma', \mathit{mem}, \mathit{reg}, \mathit{pc}')}$$

$$\frac{\mathit{mem}[C,\mathit{pc}] = i \quad \mathit{decode}\ i = \mathsf{Return} \quad \sigma = (C', \mathit{pc}') :: \sigma'}{\psi; E \vdash (C, \sigma, \mathit{mem}, \mathit{reg}, \mathit{pc}) \rightarrow (C', \sigma', \mathit{mem}, \mathit{reg}, \mathit{pc}')}$$

The Call rule checks that the call is valid with respect to the current compartment's interface—i.e., the target procedure is imported by the current compartment—which ensures that even if a compiled component is compromised it cannot exceed its static privilege level. Then it puts the calling compartment's name and program counter on the global protected call stack σ. Finally, it redirects control to the entry point of the called procedure. The Return instruction retrieves the caller's compartment and return address from the protected call stack and resumes execution there.

Compiler We next define a simple compiler that produces one low-level memory compartment for each high-level component. Each compartment is internally split into buffers, the code of procedures, and a local stack that can grow infinitely. The local stack is used to store both intermediate results and return addresses.

In standard calling conventions, the callee is generally expected to restore the register values of the caller, if it has modified them, before returning from a call. Here, however, compiled components cannot assume that other components will necessarily follow an agreed calling convention, so they must save any register that may be needed later. This means, for instance, that we save the value of the current call argument on the local stack and write the local stack pointer to a fixed location in the current compartment's memory before any cross-compartment call instruction is performed, so that the compartment can restore them when it gets control back.

The compiler must also prevent a compromised compartment from reading intermediate states from code in other compartments that may be in the middle of a call to this one. Intuitively, a secure compiler must prevent compromised compartments from distinguishing compiled components based on low-level information that (fully defined) high-level attackers don't get. In the source language, only a single argument or return value is communicated at call and return points. Hence, besides preserving their values for later, the compiler ensures that all[3] registers are cleaned before transferring control to other compartments.

The compiler implements a simple optimization for local calls. Since all procedures of a component live in the same address space and local calls don't need instrumentation, these calls can be implemented more efficiently using Jal and Jump instructions. We therefore use different procedure entry points for component-local and cross-component calls, and we skip, for local calls, the steps that store and restore register values and clean registers.

Because we do not check bounds when compiling buffer read and write operations, buffer overflows can corrupt a compartment's memory in arbitrary ways. Consequently, many buffer overflow attacks can be reproduced even in our simple setting, including, due to the local-call optimization, return-oriented programming attacks [14], [62]. In return-oriented programming, an attacker overwrites return addresses on the local stack to produce an unexpected sequence of instructions of his choice by reusing parts of the code of component-local procedures. In our setting, buffer overflow attacks thus enable compiled components to shoot themselves in the foot by storing beyond the end of a buffer and into the local call stack.

We assume compiler correctness as stated below for our compiler. Note that, in the presence of partial type safety,

[3] Technically speaking, we believe that, in our very simple setting, the compiler could choose not to clean *unused* registers and still be secure. However, our proof relies on compiled components cleaning *all* registers except the one that holds the call argument or return value. Indeed, not cleaning unused registers makes things harder because it can provide a covert channel for two compromised compartments between which interfaces would forbid any direct communication. These compartments could now exchange values through uncleared registers by interacting with the same unsuspecting uncompromised component. We conjecture that this possible cooperation between compromised components doesn't yield more attacker power in our case. However, in a setting where registers could be used to transmit capabilities, this *would* give more power to the attacker, so our compiler clears all registers but one, which also simplifies our proof.

(Theorem 4.1), proving either (1) or (2) below is enough to get the other.

Assumption 4.2 (Whole-program compiler correctness).

$$\forall P.\ P \text{ defined} \Rightarrow$$
$$(1)\ P \text{ terminates} \iff P{\downarrow} \text{ terminates} \land$$
$$(2)\ P \text{ diverges} \iff P{\downarrow} \text{ diverges}$$

Instantiating structured full abstraction We define program shapes, partial programs, and contexts in a similar way to Theorem 3.4, as detailed below. More precisely, we use isomorphic definitions so that we can later apply this theorem.

A program shape s is the pairing of a mapping from component names to component interfaces and a set that indicates uncompromised components. In the rest of the paper, we implicitly restrict our attention to *well-formed* shapes. A shape is well-formed when (1) all component interfaces in the shape only import procedures from components that are part of the shape, and (2) these procedures are exported according to the shape.

High-level partial programs P and contexts A are defined as mappings from component names to component definitions. A high-level partial program P has shape $\bullet s$ when it defines exactly the components that are marked as *uncompromised* in s, with definitions that satisfy the corresponding interfaces, and when it moreover satisfies the simple well-formedness condition that all the local buffers it uses are defined. A high-level context A has shape $\circ s$ under the same conditions, adapted for *compromised* components instead of uncompromised ones.

A low-level partial program p or context a is formed by pairing a partitioned memory with a mapping from procedure identifiers to entry points. This choice is isomorphic to having sets of named compartment memories with entry point annotations. A low-level partial program p has shape $\bullet s$ when the partitioned memory has partitions under exactly the component names that are marked as *uncompromised* in s, and the entry point mapping provides addresses for exactly the procedures that are exported by these components according to s. A low-level context a has shape $\circ s$ under the same conditions, adapted for *compromised* components instead of uncompromised ones.

We say that a high-level partial program $P \in^\bullet s$ is fully defined with respect to contexts of shape $\circ s$ when it cannot be blamed for undefined behavior when interacting with such contexts: for every $A \in^\circ s$, either reducing $A[P]$ never reaches an undefined behavior configuration, or else the current component in this undefined behavior configuration belongs to A. Similarly, a high-level context $A \in^\bullet s$ is fully defined with respect to programs of shape $\circ s$ when it cannot be blamed for undefined behavior when interacting with such programs.

Because we perform a point-wise compilation of high-level programs, separate compilation (Definition 3.5) trivially holds for our compiler. Combining it with whole-program compiler correctness (Assumption 4.2) immediately leads to the following corollary:

Corollary 4.3 (Separate compilation correctness).

$$\forall s, A \in^\circ s, P \in^\bullet s.$$
$$P \text{ fully defined wrt. contexts of shape } \circ s \Rightarrow$$
$$A \text{ fully defined wrt. programs of shape } \bullet s \Rightarrow$$
$$(1)\ A[P] \text{ terminates} \iff A{\downarrow}\ [P{\downarrow}] \text{ terminates} \land$$
$$(2)\ A[P] \text{ diverges} \iff A{\downarrow}\ [P{\downarrow}] \text{ diverges}$$

Proof technique for structured full abstraction Trace semantics were initially proposed by Jeffrey and Rathke [34], [35] to define fully abstract models for high-level languages. Patrignani *et al.* later showed how to use trace semantics [56] to prove full abstraction for a compiler targeting machine code [55]. This technique is well suited for deterministic target languages such as machine code.

This proof technique is well suited for deterministic target languages such as machine code and proceeds in two steps. First, we devise a trace semantics for low-level partial programs and contexts and relate it to the target machine's operational semantics (e.g., by proving it fully abstract [56]). This trace semantics will provide a set of traces for every partial program, describing all the execution paths that this program can reach by interacting with an arbitrary context. Second, we use the trace semantics to characterize the interaction between an arbitrary low-level context and, separately, two compiled programs that this context distinguishes, resulting in two traces with a common prefix followed by different actions. We can then use these traces to construct a high-level attacker, proving that this attacker distinguishes between the two source programs.

As our proof demonstrates, proving the trace semantics fully abstract is not a mandatory first step in the technique. Instead, we relate our trace semantics to the operational one using two weaker trace composition and decomposition conditions (Lemma 4.5 and Lemma 4.6), adapted from the key lemmas that Jeffrey and Rathke used to prove their trace semantics fully abstract [34], [35]. This reduces proof effort, since proving a trace semantics fully abstract typically requires proving a third lemma with a trace-mapping argument of its own [34], [35], [56].

Adapting the technique to undefined behavior is straightforward, essentially amounting to proving standard full abstraction for the safe subset of the language. Then one simply proves that the context produced by the mapping is fully defined, thus safe. Adapting to a closed world, however, takes more work.

The trace semantics that have been used previously to prove full abstraction characterize the interaction between a partial program and arbitrary contexts. The context's shape is typically constructed as reduction goes, based on the steps that the context takes in the generated trace. For instance, if the trace said that the context performs a call, then the target procedure would be appended to the context's interface so that this call becomes possible. For structured full abstraction, we want a finer-grained trace semantics that enables reasoning about the interaction with contexts of a specific shape. We achieve this by making the shape a parameter to the reduction relation underlying our trace semantics. To make sure that traces are compatible with this shape, we also keep track of the current compartment during reduction. This allows us to generate only context steps that adhere to the current compartment's interface, and hence to the context's shape. In particular, the context will only be able to call program procedures for which (1) there is a context compartment whose interface explicitly imports the target procedure, thus granting the privilege to call that procedure, and (2) this other context compartment is reachable from the current compartment via a chain of cross-compartment calls or returns within the context.

Moving to a closed world also makes the trace mapping argument harder. The one from Patrignani *et al.* [55], for instance, relies on changes in the context's shape, e.g., adding a helper component to the context that is not present in the low level. This is no longer possible for structured full abstraction, where the context shape is fixed..

Trace semantics for the low-level language We define a trace semantics in which traces are finite words over an alphabet $E\alpha$ of external actions, alternating between program external actions "$\gamma!$" and context external actions "$\gamma?$". We treat external actions as moves in a two-player game, viewing the context and the partial program as the players. The trace semantics is parameterized by a shape s, which the two players have. External actions either transfer control to the other player or end the game.

$$E\alpha \ ::= \ \gamma! \mid \gamma? \qquad \gamma \ ::= \ \mathsf{Call}_{\mathit{reg}}\ C\ P \mid \mathsf{Return}_{\mathit{reg}} \mid \checkmark$$

Traces ($E\alpha^*$) track the external actions (γ) performed by the context and the program. The first kind of external action is cross-boundary communication, which corresponds to the use of instrumented call instructions $\mathsf{Call}\ C\ P$ and Return when they transfer control to a compartment that belongs to the opponent. For these external actions, traces keep track of the instruction used together with *reg*, the values held by all registers when the instruction is issued. The second kind of external action is program termination, which we denote with a tick $\checkmark$ and which the opponent cannot answer ($\checkmark$ ends the game). It corresponds to the use of an instruction that makes execution stuck, such as Halt.

At any point where it has control, a player can take internal actions (any instruction that neither terminates execution nor transfers control to the opponent); these are not reflected in the trace. In particular, cross-compartment communication is considered an *internal* action when it transfers control to a compartment that belongs to the current player. Besides halting, a player can also end the game by triggering an infinite sequence of internal actions, making execution diverge. In the trace, this will correspond to not making any move: the trace observed thus far will be a maximal trace for the interaction between the program and context involved, i.e., any extension of this trace will not be shared by both the program and the context.

Intuitively, a program $p \in^\bullet s$ has trace t if it answers with the program actions described in t when facing a context $a \in^\circ s$ that produces the context actions described in t. Similarly, a program $a \in^\circ s$ has trace t if it answers with the context actions described in t when facing a program $p \in^\bullet s$ that produces the program actions described in t. We define $\mathit{Tr}_{\circ s}(p)$ to be the set of traces of a partial program p with respect to contexts of shape $\circ s$, and $\mathit{Tr}_{\bullet s}(a)$ to be the set of traces of a context a with respect to programs of shape $\bullet s$.

The player that starts the game is the one that owns the main component according to s. For each player, the trace semantics is deterministic with respect to its own actions and nondeterministic with respect to the opponent's actions. All possible actions an actual opponent could take have a corresponding nondeterministic choice, which is formalized by a property we call trace extensibility.

Lemma 4.4 (Trace extensibility).

$$\begin{aligned}&\forall t,\ s,\ p \in^\bullet s,\ a \in^\circ s.\\ &\quad (t \in \mathit{Tr}_{\circ s}(p) \wedge t.\gamma? \in \mathit{Tr}_{\bullet s}(a) \Rightarrow t.\gamma? \in \mathit{Tr}_{\circ s}(p)) \wedge\\ &\quad (t \in \mathit{Tr}_{\bullet s}(a) \wedge t.\gamma! \in \mathit{Tr}_{\circ s}(p) \Rightarrow t.\gamma! \in \mathit{Tr}_{\bullet s}(a))\end{aligned}$$

Nondeterminism disappears once we choose a particular opponent for a player, as the two key lemmas below illustrate.

Lemma 4.5 (Trace decomposition).

$$\begin{aligned}&\forall s,\ p \in^\bullet s,\ a \in^\circ s.\ \ a[p] \text{ terminates} \Rightarrow\\ &\quad \exists t.\ \ t \text{ ends with } \checkmark \wedge t \in \mathit{Tr}_{\circ s}(p) \cap \mathit{Tr}_{\bullet s}(a)\end{aligned}$$

Trace decomposition is stated for terminating programs. It extracts the interaction between a program p and a context a with dual shapes by looking at how $a[p]$ reduces, synthesizing that interaction into a trace t. Because execution terminates, this trace ends with a termination marker.

Lemma 4.6 (Trace composition).

$$\begin{aligned}&\forall t,\ s,\ p \in^\bullet s,\ a \in^\circ s.\ \ t \in \mathit{Tr}_{\circ s}(p) \cap \mathit{Tr}_{\bullet s}(a) \Rightarrow\\ &\quad (\forall E\alpha.\ (t.E\alpha) \notin \mathit{Tr}_{\circ s}(p) \cap \mathit{Tr}_{\bullet s}(a)) \Rightarrow\\ &\qquad (a[p] \text{ terminates} \iff t \text{ ends with } \checkmark)\end{aligned}$$

Trace composition is the opposite of trace decomposition, reconstructing a sequence of reductions based on synthesized interaction information. It considers a program and a context with dual shapes, that share a common trace t. The condition on the second line states that the game has ended: trace t cannot be extended by any action $E\alpha$ such that the two players share trace "$t.E\alpha$". Under these assumptions, trace composition tells us that one of the following holds: either (1) the trace ends with a termination marker $\checkmark$ and putting p in context a will produce a terminating program, or (2) putting p in context a will produce a diverging program and the trace does not end in $\checkmark$. Intuitively, if the game has ended but there is no termination marker, it must be because one of the players went into an infinite sequence of internal actions and will neither give control back nor terminate.

While the statement of these lemmas is quite close to that used in an open world setting [34], [35], the trace semantics itself has to be adapted in order to prove them in the presence of our closed world assumption. To this end, we incorporate *internal* actions within the trace semantics, thus adding more options to the nondeterministic choice of the next context action, which allows us to track at any point the currently executing compartment. When in control, a player can only perform communicating actions allowed by the interface of the current compartment. This restricts external actions as required, while also making it possible to internally switch the current compartment through allowed internal actions. Using our semantics, we thus end up with finer-grained traces that include internal communication, which can be directly mapped to high-level attackers (Assumption 4.9). The traces we use otherwise are obtained by erasing internal actions from the finer-grained traces.

Proof of SCC We prove our instance of structured full abstraction, which implies SCC by Theorem 3.4 since we have isomorphic definitions to the ones in §3.

Theorem 4.7 (Structured full abstraction). Our compiler satisfies structured full abstraction.

Recall that the basic idea behind the proof technique is to extract two traces that characterize the interaction between a low-level context and two compiled fully defined high-level programs, and then to map these two traces to a fully defined high-level context. The high-level context should reproduce the context actions described in the traces when facing the same programs as the low-level context.

Unfortunately, a compiled fully defined context cannot reproduce any arbitrary low-level trace, because the values transmitted in registers are part of external communication actions in low-level traces: As enforced by the compiler, these contexts always clear all registers but the one used for communication before giving control to the program. They can thus only produce traces in which registers are cleared in all context actions, which we call *canonical* traces. We denote by $\zeta(\gamma)$ the operation that rewrites action γ so that all registers but that one are clear. A canonical trace $\zeta_\circ(t)$ can be obtained from an arbitrary trace t by replacing all context actions "$\gamma?$" by "$\zeta(\gamma)?$". We call this operation trace canonicalization.

As we will see, being able to reproduce arbitrary canonical traces gives enough distinguishing power to the high-level context. The reason is that, because they can't trust other compartments, compiled fully defined components never read values transmitted in registers with the exception of the one used for communication. As a consequence, these components cannot distinguish context external actions based on the content of these unread registers, which are exactly the ones a compiled fully defined context cleans. Fully defined programs thus perform the exact same actions when facing a trace t or its canonicalization $\zeta_\circ(t)$, as formalized by Lemma 4.8. This means that having the high-level attacker reproduce canonical traces instead of the original traces of the low-level context will be enough to lead compiled programs into reproducing the actions they took when facing the low-level context.

Lemma 4.8 (Canonicalization).

$$\begin{array}{l}\forall t,\ s,\ P \in^\bullet s.\\ \quad P \text{ fully defined wrt. contexts of shape } \circ s \Rightarrow\\ \qquad t \in \mathit{Tr}_{\circ s}(P{\downarrow}) \iff \zeta_\circ(t) \in \mathit{Tr}_{\circ s}(P{\downarrow})\end{array}$$

The definability assumption below gives a characterization of our mapping from a canonical trace t and an action γ_1 to a compiled fully defined context $A{\downarrow}$ that reproduces the context actions in t and, depending on the next action γ the program takes, ends the game with either termination (if $\zeta(\gamma) = \zeta(\gamma_1)$) or divergence (if $\zeta(\gamma) \neq \zeta(\gamma_1)$). The context $A{\downarrow}$ will thus distinguish a program p producing trace "$t.\gamma_1!$" from any program producing "$t.\gamma!$" with $\zeta(\gamma) \neq \zeta(\gamma_1)$.

Assumption 4.9 (Definability).

$$\begin{array}{l}\forall t,\ \gamma_1,\ s.\ \ t = \zeta_\circ(t) \wedge (\exists p \in^\bullet s.\ (t.\gamma_1!) \in \mathit{Tr}_{\circ s}(p)) \Rightarrow\\ \quad \exists A \in^\circ s.\ A \text{ fully defined wrt. programs of shape } \bullet s\ \wedge\\ \qquad (1)\ t \in \mathit{Tr}_{\bullet s}(A{\downarrow}) \wedge\\ \qquad (2)\ (\gamma_1 \neq \checkmark \Rightarrow (t.\gamma_1!.\checkmark?) \in \mathit{Tr}_{\bullet s}(A{\downarrow})) \wedge\\ \qquad (3)\ \forall \gamma.\ \text{if } \zeta(\gamma) \neq \zeta(\gamma_1) \text{ then } \forall \gamma'.\ (t.\gamma!.\gamma'?) \notin \mathit{Tr}_{\bullet s}(A{\downarrow})\end{array}$$

The definability assumption gives us a fully defined context that follows trace t (1) and that, if given control afterwards via action "$\gamma!$" such that $\gamma \neq \checkmark$, acts as follows: if $\gamma = \gamma_1$ the context terminates (2) and if the context can distinguish γ from γ_1, it will make execution diverge by not issuing any action γ' (3). Since it is a compiled fully defined context, $A{\downarrow}$ can only access values transmitted using register r_{com}, the register that holds the call argument or return value. So $A{\downarrow}$ can only distinguish between γ and γ_1 when they differ in r_{com}, which is captured formally by the $\zeta(\gamma) \neq \zeta(\gamma_1)$ condition.

Proving this assumption (even on paper) would be quite tedious, so we settled for testing its correctness using QuickCheck [16]. We built an algorithm (in OCaml) that constructs A out of t. More precisely, the algorithm inputs a trace with internal actions (the finer-grained trace that erases to t) and builds a context A that reproduces context internal and external actions as prescribed by that trace. Execution will afterwards resume at a different point in A depending on the next action taken by the program. At each such point, A will either terminate execution or make it diverge depending on whether the program action is distinguishable from action γ_1. Because the trace taken as input already includes internal actions, we do not have to reconstruct them, hence our algorithm is not more difficult to come up with than one that works an open-world setting [55]. In the following, we assume that the algorithm is correct, i.e., that Assumption 4.9 holds. We can now turn to the main theorem.

Detailed proof of structured full abstraction. Consider a low-level attacker $a \in^\circ s$ distinguishing two fully defined partial programs $P, Q \in^\bullet s$ after compilation. Suppose without loss of generality that $a[P{\downarrow}]$ terminates and $a[Q{\downarrow}]$ diverges. We build a high-level attacker $A \in^\circ s$ that is fully defined with respect to programs of shape $\bullet s$ and can distinguish between P and Q.

We can first apply trace decomposition (Lemma 4.5) to a and $P{\downarrow}$ to get a trace $t_i \in \mathit{Tr}_{\circ s}(P)$ that ends with ✓, such that $t_i \in \mathit{Tr}_{\bullet s}(a)$. Call t_p the longest prefix of t_i such that $t_p \in \mathit{Tr}_{\circ s}(Q{\downarrow})$. Because trace sets are prefix-closed by construction, we know that $t_p \in \mathit{Tr}_{\circ s}(P{\downarrow}) \cap \mathit{Tr}_{\bullet s}(a)$.

Moreover, t_p is necessarily a *strict* prefix of t_i: otherwise, we could apply trace composition (Lemma 4.6) and get that $a[Q{\downarrow}]$ terminates, a contradiction. So there exists an external action $E\alpha$ such that trace "$t_p.E\alpha$" is a prefix of t_i. Now $E\alpha$ cannot be a context action, or else trace extensibility (Lemma 4.4) would imply that "$t_p.E\alpha$" is a trace of $\mathit{Tr}_{\circ s}(Q{\downarrow})$, which is incompatible with t_p being the *longest* prefix of t_i in $\mathit{Tr}_{\circ s}(Q{\downarrow})$. Therefore, $E\alpha$ is a program action, i.e., there exists γ_1 such that "$E\alpha = \gamma_1!$". Intuitively, $P{\downarrow}$ and $Q{\downarrow}$ take the same external actions until the end of t_p, where $P{\downarrow}$ takes external action "$\gamma_1!$" and $Q{\downarrow}$ does not (it takes either a different action $\gamma \neq \gamma_1$ or no external action at all).

Now, let t_c be the canonicalization of trace t_p, i.e., $t_c = \zeta_\circ(t_p)$. By canonicalization (Lemma 4.8), "$t_c.\gamma_1!$" $= \zeta_\circ(t_p.\gamma_1!)$ is a trace of $P{\downarrow}$. We can thus use apply definability (Assumption 4.9) to trace t_c and action γ_1, using $P{\downarrow} \in^\bullet s$ as a witness having trace "$t_c.\gamma_1!$". This yields a fully defined context $A \in^\circ s$ such that:

(1) $t_c \in \mathit{Tr}_{\bullet s}(A{\downarrow})$,
(2) $\gamma_1 \neq ✓ \Rightarrow (t_c.\gamma_1!.✓?) \in \mathit{Tr}_{\bullet s}(A{\downarrow})$,
(3) $\forall \gamma,\ \gamma'.\ \ (t_c.\gamma!.\gamma'?) \in \mathit{Tr}_{\bullet s}(A{\downarrow}) \Rightarrow \zeta(\gamma) = \zeta(\gamma_1)$.

We now show that these conditions imply that $A{\downarrow}\ [P{\downarrow}]$ terminates while $A{\downarrow}\ [Q{\downarrow}]$ diverges.

First, we look at $P{\downarrow}$. Consider the case where $\gamma_1 = ✓$. In this case, by applying trace extensibility to $A{\downarrow}$ in (1), we get that "$t_c.✓!$" is a trace of $A{\downarrow}$, so trace composition allows us to conclude that $A{\downarrow}\ [P{\downarrow}]$ terminates. Now if $\gamma_1 \neq ✓$ then this action gives back control to the context, which, given (2), will perform action "$✓?$". Applying trace extensibility to $P{\downarrow}$, $P{\downarrow}$ has trace "$t_c.\gamma_1!.✓?$", so we can apply trace composition and deduce that $A{\downarrow}\ [P{\downarrow}]$ terminates in this case as well.

Now, regarding $Q{\downarrow}$, we first obtain the following by applying canonicalization to t_p, "$t_p.✓!$", and "$t_p.\gamma_1!$":

(a) $t_c = \zeta_\circ(t_p) \in \mathit{Tr}_{\circ s}(Q{\downarrow})$,
(b) $(t_c.✓!) = \zeta_\circ(t_p.✓!) \in \mathit{Tr}_{\circ s}(Q{\downarrow}) \Rightarrow (t_p.✓!) \in \mathit{Tr}_{\circ s}(Q{\downarrow})$,
(c) $(t_c.\gamma_1!) = \zeta_\circ(t_p.\gamma_1!) \in \mathit{Tr}_{\circ s}(Q{\downarrow}) \Rightarrow (t_p.\gamma_1!) \in \mathit{Tr}_{\circ s}(Q{\downarrow})$.

After following trace t_c, which $Q{\downarrow}$ has from (a), $Q{\downarrow}$ cannot perform a terminating action: otherwise using (b) and trace extensibility for a and t_p, we could apply trace composition to trace "$t_p.✓$" and get that $a[Q{\downarrow}]$ terminates, which is a contradiction. $Q{\downarrow}$ cannot perform action γ_1 either, since (c) would then violate the fact that t_p is the longest prefix of t_i in $\mathit{Tr}_{\circ s}(Q{\downarrow})$. So $Q{\downarrow}$ only has two options left. The first is to perform no external action by going into an infinite sequence of internal transitions. In this case, using (1), we can apply trace composition to get that $A{\downarrow}\ [Q{\downarrow}]$ diverges. The second option is to give control back to the context using an external action γ so that $✓ \neq \gamma \neq \gamma_1$. Because fully defined compiled programs clean registers, they only yield canonical actions, i.e. $\gamma = \zeta(\gamma) \wedge \gamma_1 = \zeta(\gamma_1)$. Combined with (3), this entails that if $A{\downarrow}$ produced an action γ', we would have $\gamma = \gamma_1$, which is false. Hence, $A{\downarrow}$ doesn't produce any action: it goes into an infinite sequence of local transitions. We can again apply trace composition to get that $A{\downarrow}\ [Q{\downarrow}]$ diverges.

We finally apply separate compiler correctness (Corollary 4.3) to conclude the proof. □

5 Related Work

Fully abstract compilation Fully abstract compilation was introduced in the seminal work of Martín Abadi [1] and later investigated by the academic community. (Much before this, the concept of full abstraction was coined by Milner [46].) For instance, Ahmed *et al.* [9]–[11] proved the full abstraction of type-preserving compiler passes for functional languages and devised proof techniques for *typed* target languages. Abadi and Plotkin [6] and Jagadeesan *et al.* [33] expressed the protection provided by a mitigation technique called address space layout randomization as a probabilistic variant of full abstraction. Fournet *et al.* [29] devised a fully abstract compiler from a subset of ML to JavaScript.

Patrignani *et al.* [43], [55] were recently the first to study fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting hardware architectures featuring a new coarse-grained isolation mechanism. They also recently proposed proof techniques for full abstraction that work for untyped target languages [20], [56]. Until recently, Patrignani *et al.* studied fully abstract compilers that by design violate our separate compilation property, so they cannot be applied to our compartmentalizing compilation setting.

In recent parallel work, Patrignani *et al.* [58] proposed an extension of their compilation scheme to protecting multiple components from each other. The attacker model they consider is different, especially since their source language does not have undefined behavior. Still, if their source language were extended with unsafe features, we expect that our SCC property might hold for their compiler.

Formal reasoning about compartmentalized code SCC is orthogonal to formal techniques for reasoning about compartmentalized software: SCC allows *transferring* security guarantees for compartmentalized code written in a source language to machine code via compartmentalizing compilation, but SCC itself does not provide effective reasoning principles to obtain those security guarantees in the first place. The literature contains interesting work on formally characterizing the security benefits of compartmentalizing software. Promising approaches include Jia *et al.*'s work on System M [36], and Devriese *et al.*'s work on logical relations for a core calculus based on JavaScript [21], both of which allow bounding the behavior of a program fragment based on the interface or capabilities it has access to. One significant challenge we attack in this paper is languages with undefined behaviors, while in these other works illegal actions such as accessing a buffer out of bounds must be detected and make the program halt.

Verifying correct low-level compartmentalization Recent work focused on formally verifying the correctness of low-level compartmentalization mechanisms based on software fault isolation [40], [47], [70] or tagged hardware [12]. That work, however, only considers the correctness of the low-level compartmentalization mechanism, not the compiler and not high-level security properties and reasoning principles for code written in a programming language with components. Communication between low-level compartments is generally done by jumping to a specified set of entry points, while the model we consider in §4 is more structured and enforces correct calls and returns.

Finally, seL4 is a verified operating system microkernel [39], that uses a capability system to separate user level threads and for which correct access control [61] and noninterference properties [48] were proved formally.

6 Conclusion and Future Work

We have introduced a new secure compartmentalizing compilation property, related it to the established notion of full abstraction, and applied our property in a carefully simplified setting: a small imperative language with procedures compiling to a compartmentalized abstract machine. This lays the formal foundations for studying the secure compilation of mutually distrustful components written in unsafe languages.

In the future we plan to build on this groundwork to study more realistic source and target languages, compilers, and enforcement mechanisms. In the long run, we would like to apply this to the C language by devising a secure compartmentalizing variant of CompCert that targets a tag-based reference monitor [12] running on a real RISC processor [18]. We have in fact started working towards this long term goal [37], but this will take time to achieve. Beyond tagged hardware, we would also like to implement the abstract compartmentalization machine from §4 in terms of various enforcement other mechanisms, including: process-level sandboxing [13], [31], [38], [59], software fault isolation (SFI) [68], capability machines [66], and multi-PMA systems [58]. As we target lower-level machines, new problems will appear: for instance we need to deal with the fact that memory is finite and resource exhaustion errors cannot be hidden from the attacker, which will require slightly weakening the security property. Finally, we would like to study more interesting compartmentalization models including dynamic component creation and nested components, and the way these extensions influence the security property.

Acknowledgments We thank André DeHon, Deepak Garg, and Andrew Tolmach for helpful discussions and thoughtful feedback on earlier drafts. Yannis Juglaret is supported by a PhD grant from the French Department of Defense (DGA) and Inria. Arthur Azevedo de Amorim and Benjamin C. Pierce are supported by NSF award 1513854, *Micro-Policies: A Framework for Tag-Based Security Monitors*.

References

[1] M. Abadi. Protection in programming-language translations. Research Report 154, SRC, 1998.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. *ACM CCS*. 2005.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 13(1), 2009.

[4] M. Abadi and J. Planul. On layout randomization for arrays and functions. *POST*. 2013.

[5] M. Abadi, J. Planul, and G. D. Plotkin. Layout randomization and nondeterminism. *MFPS*, 298:29–50, 2013.

[6] M. Abadi and G. D. Plotkin. On protection by layout randomization. *TISSEC*, 15(2):8, 2012.

[7] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. *POPL*. 2015.

[8] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012.

[9] A. Ahmed. Verified compilers for a multi-language world. *SNAPL*. 2015.

[10] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *ICFP*. 2008.

[11] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011.

[12] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. 2015.

[13] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008.

[14] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proc. ACM CCS*, 2008.

[15] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015.

[16] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000.

[17] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. *USENIX Security*, 2014.

[18] A. DeHon, E. Boling, R. Nikhil, D. Rad, J. Schwarz, N. Sharma, J. Stoy, G. Sullivan, and A. Sutherland. DOVER: A Metadata-Extended RISC-V. In *RISC-V Workshop*, 2016. Accompanying talk at http://youtu.be/r5dIS1kDars.

[19] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. *ASPLOS*, 2008.

[20] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. *POPL*, 2016.

[21] D. Devriese, F. Piessens, and L. Birkedal. Reasoning about object capabilities with logical relations and effect parametricity. *EuroS&P*, 2016.

[22] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. *ASPLOS*, 2015.

[23] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, A. DeHon, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, and G. Sullivan. Hardware support for safety interlocks and introspection. *AHNS*, 2012.

[24] Ú. Erlingsson. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, 2007.

[25] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. *OSDI*. 2006.

[26] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe,

S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. *IEEE S&P*, 2015.

[27] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. C. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. *CCS*. 2015.

[28] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *ICFP*. 2002.

[29] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013.

[30] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. *IEEE S&P*, 2014.

[31] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. *CCS*. 2015.

[32] Intel software guard extensions (Intel SGX) programming reference, 2014.

[33] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011.

[34] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.

[35] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. *ESOP*. 2005.

[36] L. Jia, S. Sen, D. Garg, and A. Datta. A logic of programs with interface-confined code. *CSF*. 2015.

[37] Y. Juglaret, C. Hritcu, A. A. de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Towards a fully abstract compiler using micro-policies: Secure compilation for mutually distrustful components. *CoRR*, abs/1510.00697, 2015.

[38] D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003.

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *SOSP*. 2009.

[40] J. Kroll, G. Stewart, and A. Appel. Portable software fault isolation. *CSF*. 2014.

[41] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. *POPL*. 2014.

[42] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. *CCS*. 2013.

[43] A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for ML modules. *APLAS*, 2015.

[44] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[45] X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.

[46] R. Milner. *Processes: a mathematical model of computing agents*, pages 157–174. North-Holland, 1975.

[47] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. *PLDI*. 2012.

[48] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. *IEEE S&P*. 2013.

[49] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Hardware-Enforced Comprehensive Memory Safety. *IEEE Micro*, 33(3):38–47, 2013.

[50] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. *CGO*. 2014.

[51] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. *SNAPL*. 2015.

[52] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. *PLDI*. 2009.

[53] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. *ISMM*. 2010.

[54] M. Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, 2015.

[55] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015.

[56] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42:22–45, 2015.

[57] M. Patrignani, D. Clarke, and F. Piessens. Secure compilation of object-oriented components to protected module architectures. *APLAS*. 2013.

[58] M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. *CSF*, 2016.

[59] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*. 2003.

[60] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. *EuroSys*. 2009.

[61] T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. *ITP*. 2011.

[62] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *ACM CCS*. 2007.

[63] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. *IEEE S&P*. 2013.

[64] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. *USENIX Security*. 2014.

[65] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.

[66] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *IEEE S&P*, 2015.

[67] C. Williams. Oracle's Larry Ellison claims his Sparc M7 chip is hacker-proof – errr... The Register, 2015.

[68] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *CACM*, 53(1):91–99, 2010.

[69] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. *USENIX Security*. 2013.

[70] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: fully verified software fault isolation. *EMSOFT*. 2011.

[71] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *FSE*. 2004.