Beyond Good and Evil

Formalizing the Security Guarantees of Low-Level Compartmentalization

Yannis Juglaret\textsuperscript{1,2}, Cătălin Hrițcu\textsuperscript{1}, Arthur Azevedo de Amorim\textsuperscript{3}, Benjamin C. Pierce\textsuperscript{3}

\textsuperscript{1}Inria Paris \hspace{2em} \textsuperscript{2}Université Paris Diderot (Paris 7) \hspace{2em} \textsuperscript{3}University of Pennsylvania

Abstract—Compartmentalization is widely regarded as good security-engineering practice: if we break up 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. But the formal guarantees provided by such low-level compartmentalization have seen surprisingly little investigation.

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

1 Introduction

Today’s computer systems are distressingly insecure. Visiting a website, opening an email, or serving a client request is often enough to cause a computer to be compromised by 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. While many techniques have been proposed for guaranteeing memory safety [13], [18], [21], [38], [45]–[49], the challenges of efficiency [48], [49], precision [63], scalability [66], backwards compatibility [16], and effective deployment [13], [18], [20], [21], [38], [45]–[47] have hampered their widespread adoption.

Meanwhile, new mitigation techniques are proposed to deal with the most onerous consequences of the lack of memory safety—for instance, techniques for attempting to prevent control-flow hijacking even in memory-unsafe settings [2], [3], [22]. Unfortunately, these defenses often underestimate the power of the attackers they will face [17], [22], [24], [25], [27], [59]—if, indeed, they have any clear model at all of what they are supposed to protect against. Formally studying and 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 complex world in which low-level attacks occur.

In this paper we focus on low-level compartmentalization [14], [28], [62], a class of strong, practical defense mechanisms whose security has seen surprisingly little formal investigation. The 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 provides not only software engineering gains, but also 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. Moreover, because compartmentalization is a coarse-grained enforcement mechanism, efficiency and backwards compatibility issues are generally smaller than for full-blown memory safety. Indeed, two different compartmentalization technologies are already widely deployed: process-level privilege separation [14], [28], [34] (e.g., used by OpenSSH [55] and for sandboxing plugins and tabs in modern web browsers [56]) and software-fault isolation [61] (e.g., provided by Google Native Client [64]); and many more such technologies are on the drawing boards [13], [29], [54], [62].

So what security guarantees does compartmentalization 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]–[12], [26], [30], [51]. A fully abstract compiler 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 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—perhaps written in an unsafe language or even directly in assembly—the resulting executable 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 exert—is easier in a high-level language. And second, because by phrasing the property in terms of low-level and high-level programs rather than directly in terms of attacker behaviors, specific notions of privilege, etc., we can re-use the same definition for many specific languages.)

Since full abstraction works by partitioning the world into a program and its context, we might expect it to make sense for compartmentalized programs as well: Compartments that are assumed to be subject to control-hijacking attacks would...
be grouped into the “low-level context,” while those that are 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. Unfortunately, this intuition does not withstand closer examination. Fully abstract compilation, as previously formulated in the literature, suffers from three important limitations that make it unsuitable for characterizing the security guarantees of low-level compartmentalization.

First, fully abstract compilation assumes that our source language is itself 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. Programs in these languages are allowed to do anything—in particular, to do whatever a remote attacker wants—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 quite bizarre and easily exploitable behaviors when this assumption is broken.) The point of compartmentalization is to ensure that the effects of undefined behavior are restricted to compromise of the component in which it occurs, and that the other components can only be influenced by compromised components via specified interactions respecting the existing interfaces. To formally characterize the security of low-level compartmentalization we thus need a 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 it requires that all nondeterminism of the source language—including nondeterminism due to undefined behaviors—be preserved by the compilation process. While targeting machine code, no realistic implementation of an unsafe language like C is going to preserve all of this nondeterminism—and removing it would negate the performance and optimization benefits that are the reason for allowing undefined behaviors in the first place—so no realistic C implementation has any hope of being fully abstract.

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. 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. In our static compartmentalization setting the breakup of the application into components is fixed in advance and so are the privileges of all the components. Compromised components cannot change these basic rules of the compartmentalization game and have to play by them. (We consider here the case where the set of compartments is fixed. In a setting where new components can be dynamically created, the details of the game will probably get more complicated; still, we expect a secure compartmentalization property to make a closed world assumption about the compromise of existing components.)

Third, because the definition of full abstraction involves applying the compiler only to the trusted “program” and not to the untrusted context in which it runs, fully abstract compilers can achieve their protection goals by introducing one single barrier around the trusted part to protect it from the untrusted part [8], [39], [50], [51], [53]. 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 low-level compartmentalization, where we generally cannot predict which components may be vulnerable to compromise by control hijacking attacks and where we therefore need to consider multiple compromise scenarios. Fine-grained compartmentalization allows us to build more secure applications that go beyond the blunt trusted/untrusted distinction made by some fully abstract compilers. To describe its guarantees accurately, we need a new property that captures the protection obtained by breaking up applications into multiple mutually distrustful components running with least privilege, and that allows us to reason about all compromise scenarios.

Our main contribution is to define such a property, which we call secure compartmentalization (§2). While similar in many respects to full abstraction, secure compartmentalization overcomes the three limitations above. First, secure compartmentalization 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 [60] for undefined behavior in any context satisfying fixed interfaces. Second, secure compartmentalization makes a closed-world assumption about compromised components, enforcing the basic rules of the compartmentalization game like the fixed breakup into components and the fixed privilege level for each component. Third, secure compartmentalization ensures protection for multiple, mutually distrustful components; it does not assume we know in advance which components are going to be compromised by undefined behavior, but instead it explicitly quantifies over all possible compromise scenarios.

Our second contribution is relating secure compartmentalization to more standard formulations of full abstraction both intuitively and formally (§3). We start from full abstraction and illustrate 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 put together and instantiated to obtain secure compartmentalization. While secure compartmentalization 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 proving secure compartmentalization for a simple unsafe imperative language with procedures compiling to an abstract machine with protected interacting components (§4). The proof adapts a standard technique called trace semantics [32], [52], via the reduction to structured full abstraction. In the future, we expect that our secure compartmentalization property can be used to formally assess the security of existing compartmentalization mechanisms [13], [28], [29], [34], [54]–[56], [64]. However, this is nontrivial because of the extreme complexity of all the involved artifacts (e.g., efficient compilers for C or C++ targeting a realistic hardware platform). Our simple secure-compartmentalization proof constitutes a first step in this direction and a sanity check for our property.

We describe each of our three contributions in detail (§2–§4), and close 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) complete technical details for the secure compartmentalization instance from §4; and (c) a trace mapping algorithm in OCaml supporting Assumption 4.9. It can all be found at http://yannis.computer/papers/bge.html

2 Secure Compartmentalization

In this section we give an intuitive explanation of low-level compartmentalization, its attacker model, and its security benefits, and we introduce our proposed secure compartmentalization property.

We consider low-level compartmentalization mechanisms provided by the compiler and runtime system for an unsafe programming language with some notion of components. (We use the term “runtime system” loosely to include operating system mechanisms [14], [28], [34], [55], [56] and/or hardware protections [13], [29], [54], [62] that may be used by the compiler.) In §4 we 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 (e.g., a mechanism for dividing an application into components during its initialization phase and a runtime enforcement mechanism for ensuring that these component boundaries are respected during the rest of its execution). Security-conscious developers can break up large applications written in such a language 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-compartment calls (e.g., ACLs for files). In §4, we instantiate this picture with a rather simple and rigid notion of compartments and interfaces, where compartments don’t directly share any state and where the only thing one compartment can do to another one is to call the procedures allowed by the interfaces of both components. We also assume that the division of the application into components and the interfaces of those components are statically determined and fixed throughout execution.

We do not fix a specific compartmentalization mechanism; we just assume that whatever mechanism is chosen is able to guarantee that, even if a component is compromised 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 to compromise more components.

We do not assume we know in advance which components will be compromised: the compartmentalization 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 [28], 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 thing might happen...” If the practical consequences of some plausible compromise scenario are too serious, developers can reduce or separate privilege by narrowing interfaces or splitting components into smaller pieces. They can also make components more defensive by dynamically validating the inputs they receive from other potentially compromised components.

For instance, developers of a compartmentalized web browser [56] 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. One outcome of this exercise might be the observation 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 might not disrupt another.

Our goal is to articulate a secure compartmentalization property that supports this sort of reasoning about multiple compromise scenarios and that clarifies the attacker model behind low-level compartmentalization. At the same time, secure compartmentalization may be useful as a target property for developers of compartmentalization mechanisms who want
to argue formally that their mechanisms are secure. In the rest of this section we first explain the main idea behind secure compartmentalization, and then its formal definition.

An application is a set $C$s of components, with corresponding interfaces $C$s. These components are separately compiled (written $C$s) and linked together (written $\Gamma(C)$) to form an executable binary for the application. The interface information is carried along through compilation, linking, and loading, for use at runtime by the dynamic part of the compartmentalization mechanism.

Secure compartmentalization 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 perform undefined behaviors even if we replace the compromised components with any code that obeys the original interfaces. (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.) Figure 1 illustrates one way to partition five components $C_1, \ldots, C_5$ with interfaces $i_1, \ldots, i_5$, representing the scenario when $C_2$, $C_4$, and $C_5$ are compromised and $C_1$ and $C_3$ are not. In order for this compromise scenario to be accepted, $C_1$ 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 defined as follows:

**Definition 2.1 (Full definedness).**

We call a set of components $C$s fully defined with respect to interfaces $B$s if for all components $B$s satisfying $B$s, $C$s cannot cause undefined behavior alongside $B$s.

The relation defining when $C$s cannot cause undefined behavior alongside $B$s is a language-specific parameter to our definition of secure compartmentalization. For instance, in the simple imperative language in §4, we say that $C$s cannot cause undefined behavior alongside $B$s if the program $\approx(C \cup B)$ does not reduce to a stuck non-final state (corresponding to undefined behavior) in which the currently executing component is one of the ones in $C$s (i.e., no component in $C$s can be “blamed” [60] for undefined behavior).

Secure compartmentalization states that, in all such compromise scenarios, the compiled compromised components must not cause more harm to the compiled uncompromised components via low-level attacks than some high-level components already could in the source language. Basically this ensures 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 secure compartmentalization, instead of having to reason about the low-level consequences of unde-
fined 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, secure compartmentalization 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 this 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- or low-level) complete programs are distinguishable, i.e., they produce different observable outcomes when executed. For this compromise scenario, secure compartmentalization specifies that, if compiled compromised components $C_2\downarrow$, $C_4\downarrow$, $C_5\downarrow$ can distinguish the two variants at the low level, then there must exist some (fully defined) components $A_2, A_4, A_5$ that distinguish the two variants at the high level. With all this in mind, the secure compartmentalization property is formally expressed as follows:

**Definition 2.2 (Secure Compartmentalization).**
- For any complete compartmentalized program and for all ways of partitioning this program into a set of uncompromised components $Cs$ and their interfaces $Cl$s, and a set of compromised components $Bs$ and their interfaces $Bl$s, so that $Cs$ is fully defined with respect to $Cl$s...
- for all ways of replacing the uncompromised components with components $Ds$ satisfying the same interfaces $Cl$s and being fully defined with respect to $Bl$s...
- if $\triangleright(Cs\cup Bs\downarrow) \not\sim_L \triangleright(Ds\cup Bl\downarrow)$,
- then there exist components $As$ satisfying interfaces $Bl$s and fully defined with respect to $Cl$s such that $\triangleright(Cs \cup As) \not\sim_H \triangleright(Ds \cup As)$.

3 From Full Abstraction to Secure Compartmentalization

§2 presented secure compartmentalization by directly characterizing the attacker model against which it defends. In this section we step back and show how it can instead be obtained by starting from the well-established notion of full abstraction and successively 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 secure compartmentalization. Reducing secure compartmentalization to structured full abstraction is not only interesting from a theoretical point of view, but also practically useful, since structured full abstraction can be more easily shown 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 a program in the source language already could. This is a much stronger property than just compiler correctness [37], [40], as it requires enforcing high-level language abstractions against arbitrary low-level attackers, not just against code produced by the same compiler.

Formally, full abstraction is phrased as a distinguishability game where low-level and high-level attackers are specified as having exactly the same distinguishing power.

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

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

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, thus 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.

The right-to-left implication normally follows from compiler correctness by simply using the compiler to translate any high-level distinguishing context. The left-to-right direction is more interesting and difficult to obtain, since it requires some way to map each low-level distinguishing context to a high-level one. To see this, it may help to look 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 meaningfully be applied to compiling an unsafe language with undefined behaviors to a deterministic machine. Undefined behaviors are highly nonterministic, compilers to deterministic machines make choices that resolve this nondeterminism, and this breaks the full abstraction property, which requires that the source and target programs have exactly the same amount of nondeterminism.

To adapt full abstraction to a source language with undefined behaviors, we need to restrict attention only to well-defined complete programs in the source language, which we assume to be deterministic. (For a source language like C, obtaining determinism requires also expanding out any “implementation specific” behavior.) Defining full abstraction still requires a little care, though. For instance, the following variant is wrong:

$$\forall A. A[P] \text{ and } A[Q] \text{ defined } \Rightarrow A[P] \not\sim_H A[Q] \iff (\forall a. a[P\downarrow] \sim_L a[Q\downarrow])$$

Any programs $P$ and $Q$ that 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. 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 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 need a compositional definition of _fully defined_ behavior for programs and contexts. A direct definition fails though, since it would be circular: intuitively, a program is fully defined if in does not cause undefined behavior in any fully defined context, and dually a context is fully defined if it does not cause undefined behavior when we plug any fully defined program into it. As done implicitly in §2, we break this circularity using blame [60]: intuitively we call a partial program _fully defined_ when it cannot be blamed for undefined behavior in any context. Similarly, we call a context fully defined when it cannot be blamed for undefined behavior for any program that we plug into it. This allows us to define 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 $Q$

$$\forall A. \text{A fully defined } \Rightarrow A[P] \sim_H A[Q] \iff (\forall a. a[P]_L \sim_L a[Q]_L)$$

By requiring that $P$, $Q$, and $A$ are fully defined 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. If we look at the interesting (left-to-right) direction of the equivalence in contrapositive form

$$\forall P, Q \text{ fully defined. } (\exists a. a[P]_L \not\sim_L a[Q]_L) \Rightarrow (\exists A. A \text{ fully defined } \Rightarrow A[P] \not\sim_H A[Q])$$

the $P, Q$ fully defined pre-condition makes this weaker than full abstraction, while _A fully defined_ post-condition makes it stronger. The post-condition is valuable, as it allows 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 a partial program to be very defensive about validating the inputs it receives. Without additional information about the program’s context though, we can only 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 (and separate compilation in §2) will in fact gather more information about contexts and thus use a weaker notion of full definedness. Moreover, separate compilation will allow us to cover 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 work in our compartmentalization setting though, in which the breakup of the application into components is fixed in advance and so are the interfaces (and thus privileges) of all the components. In our setting, the definition of full abstraction needs to be changed to keep track of and respect such structural constraints, otherwise a low-level context with 2 components could be mapped back to a high-level context with 3 components that have different exported interfaces (and thus privileges).

**Solution 2: Structured full abstraction**

For solving this problem we introduce a structured variant of full abstraction, in which partial programs ($\bullet$) and contexts ($\circ$) are assigned dual parts of pre-defined complete program _shapes_. A shape can be anything, from a breakup into components with their interface (Theorem 3.4), to maybe the maximal size of a component’s code after compilation (exposing size leakage in a setting where it’s too costly to hide component sizes).

**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 $P \in^s s$ and $Q \in^s s$ so that $P$ and $Q$ are _fully defined_ with respect to contexts of shape $\circ s$,

$$\forall A \in^s s. \ A \text{ fully defined wrt programs of shape } \bullet s \Rightarrow A[P] \sim_H A[Q]$$

$$\iff (\forall a \in^s s. a[P]_L \sim_L a[Q]_L)$$

This property universally quantifies over any complete program shape $s$ and requires that $P \in^s s$ (read “program $P$ has shape $s$”), $Q \in^s s$, and $A \in^s s$ (“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 compartmentalization**

Secure compartmentalization can be recovered in a natural way as an instance of structured full abstraction (Definition 3.3). For both the source and the target language 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.** Structured full abstraction instantiated to components as described above implies secure compartmentalization.

**Proof.** Straightforward (though tedious). A machine-checked Coq proof can be found in the auxiliary materials. \(\square\)
Problem 3: Statically known trusted/untrusted split. So we can recover secure compartmentalization as an instance of structured full abstraction. However, while secure compartmentalization 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 a 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.

Formally, a fully abstract compiler does not need to compile contexts—only programs. In fact, even the types of contexts and of partial programs could well be completely different (for instance, the type of lambda calculus contexts and terms are different, and a compiler for lambda calculus terms cannot compile contexts). Even when the types do match and 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. This means that, if we start with a complete high-level program $C$ and look at two different compromise scenarios $C = A_1[\varphi]$ and $C = A_2[\varphi]$, 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 $A_1 \downarrow \varphi \neq A_2 \downarrow \varphi$) or indeed even behaviorally equivalent results (i.e., it could well be that $A_1 \downarrow \varphi \not\approx L A_2 \downarrow \varphi$). 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. Indeed, uncertainty about sources of vulnerability is precisely the motivation for compartmentalization: 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 verified its safety—protecting only the verified code from all the rest using a fully abstract compiler [7] is likely still suboptimal in terms of protection, since this 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], [39], [50], [51], [53] (although recently proposed extensions [54] aim at lifting this restriction in some cases). While this setup is appropriate for protecting trusted (e.g., verified) code from its untrusted context [7], it is not suitable for our compartmentalization setting, where we do not know in advance which components will be dynamically compromised and which ones not and want to simultaneously protect against all possible compromise scenarios.

Solution 3: Separate compilation. We can address this by requiring that the compiler (and low-level context application operation, e.g., linker and loader) satisfy one additional property that we call separate compilation:

Definition 3.5. We say that a compiler $\downarrow$ satisfies separate compilation if:

1) The type of contexts and programs is the same, so the $\downarrow$ compiler can also compile contexts; and

Requiring that context application and compilation commute (condition 2) implies that, if some complete program $C$ can be written as both $C = A_1[\varphi]$ and $C = A_2[\varphi]$, then separately compiling each of these splits yields behaviorally equivalent results: $(A_1[\varphi] \downarrow) \approx L (A_2[\varphi] \downarrow)$. With separate compilation, the interesting implication direction of full abstraction (Definition 3.2) can be instantiated as follows:

$$\forall B. \forall P, Q \text{ fully defined. } ((B[P] \downarrow) \not\approx L (B[Q] \downarrow)) \Rightarrow (\exists A. A \text{ fully defined } \Rightarrow A[P] \not\approx L A[Q])$$

One compelling reading of this is that for all ways to break a complete program into a compromised context $B$ and an uncompromised program $P$ (i.e., for all compromise scenarios), and for all programs $Q$ that we can replace 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.

Note that in a language without undefined behavior this property would be trivially true from just (whole program) correct compilation (e.g., see Assumption 4.2) by picking $A = B$. However, it is nontrivial for a language with undefined behavior because context $B$ can cause undefined behavior, so correct compilation does not apply for $B[P]$ and $B[Q]$. In our setting, this property allows us to replace reasoning about the low-level implications of the undefined behavior in the original context, by reasoning about a fully defined high-level context. It is trivial to check that our instance of structured full abstraction from Theorem 3.4 does satisfy separate compilation.

4 A Simple Instance

We illustrate secure compartmentalization in a very simple setting that captures its main features. To this end, we build a compiler from an unsafe language with components to an abstract machine with compartments. We show how to adapt a standard full abstraction proof technique called trace semantics [51] to prove secure compartmentalization. This technique is well suited for deterministic target languages such as machine code.
Source: simple imperative language with components, procedures, and buffers

We work with an unsafe language whose programs consist of components communicating via procedure calls. Buffer overflows are undefined behavior and may open the door to low-level attacks after compilation. However, because of low-level compartmentalization, the effects of these attacks will be contained to the offending component.

The syntax for expressions, given below, is that of a standard imperative language with mutually recursive procedures and buffers. Each component \( C \) has local procedures \( P \) and private local buffers \( b \). Loops are encoded using recursive calls, sequencing is encoded as a binary operation, and variables are encoded using buffers. In particular, the variable that holds the unique procedure call argument is always passed in the first cell of the first buffer of the callee component. For simplicity, only integers are first class values and can be passed over component boundaries, using procedure calls and returns. Buffers are second class.

\[
e ::= i | e \otimes e | if \ e \ then \ e_1 \ else \ e_2 | b[e] | b[e_1] := e_2 | C \cdot P(e) | exit
\]

where \( \otimes \in \{ +, -, \times, \div, \leq, \ldots \} \).

The small-step operational semantics of this language is mostly standard and reduces configurations \((C, s, \sigma, K, e)\) where \( C \) is the current component, \( s \) is a state recording buffers values for each component, \( \sigma \) is a global call stack, \( K \) is an evaluation context, and \( e \) the currently evaluated expression. This semantics is deterministic for defined programs. Given that undefined behaviors cannot be described by the high-level semantics in any reasonable way, we model them as stickiness [41]. Hence, reduction gets stuck when trying to access or update a buffer out of bounds, which are the only forms of undefined behavior in our language. We can state partial type safety (partial progress and usual preservation), which says that well-formed programs can only go wrong by reducing to an out-of-bounds operation on a buffer.

**Conjecture 4.1 (Partial progress).** For any well-formed configuration \( cfg \), one of the following holds:

1. \( cfg \) is a final configuration (value or exit);
2. \( cfg \) takes a step;
3. \( cfg \) is stuck but of one of the following forms:
   a) \((C, s, \sigma, b[\square] : K, i)\) where \( s[C, b, i] \) is undefined;
   b) \((C, s, \sigma, b[i] := \square : K, i')\) where \( s[C, b, i] \) is undefined.

Components have interfaces which specify which procedures they import and export. To satisfy an interface, a component must define the procedures exported in the interface and must only perform cross-component calls to procedures imported in the interface. In our source language, this is checked statically. Interfaces are the same in this language and in our target abstract machine.

A component also has private procedures, which are those that are not exported according to its interface. The component that defines a private procedure is the only component that can call it, while public procedures can be called by any other component that imports the procedure.

Target: abstract machine with interacting components

Our target machine models a RISC instruction set extended with a compartmentalization mechanism. The compiler will map each source component to its own compartment. The machine has a fixed set of registers and a program counter \( pc \) register. Because resource exhaustion and integer-size related problems are orthogonal to our concerns in this paper, we assume unbounded words and infinite memory. This memory is split into separate infinite address spaces, one for each compartment. Memory addressing is relative to the current compartment. The machine has the following instructions [13]:

\[
instr ::= \text{Nop} | \text{Const} \ i \rightarrow r_d | \text{Mov} r_s \rightarrow r_d \\
| \text{Load} \ r_p \rightarrow r_d | \text{Store} \ r_p \leftarrow r_s \\
| \text{Jump} \ r | \text{Jal} \ r | \text{Call} \ C \cdot P | \text{Return} \\
| \text{Binop} r_1 \otimes r_2 \rightarrow r_d | \text{Bnz} \ r \ i | \text{Halt}
\]

Instructions Jal (jump-and-link) and Jump redirect control flow to an address in the same address space stored in a register. However, Jal also communicates a return address in a register, so that the target code can later resume execution at the location that followed the Jal instruction. We use instructions Call \( C \cdot P \) and Return to change compartments. These instructions are subject to dynamic compartmentalization constraints; in a realistic system such constraints could be enforced e.g. using a shadow call stack [3], [23] or return capabilities [33]. Since Call and Return would probably come with monitoring overhead, Jump and Jal instructions are still useful to perform compartment-local calls. The reduction rules for Call and Return are provided below.

\[
mem[C, pc] = i \quad \text{decode} \ i = \text{Call} \ C' \cdot P', \ pc' = E[C'[i][P'] \\
C' = C \lor C', P' \in \psi[C].\text{import} \quad \sigma' = (C, pc+1) :: \sigma \\
\psi; E \vdash (C, \sigma, \text{mem}, \text{reg}, pc) \rightarrow (C', \sigma', \text{mem}, \text{reg}, pc')
\]

The Call instruction dynamically 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, Call puts the calling compartment’s name and program counter on a global, protected call stack \( \sigma \). Finally, it redirects control flow to the entry point of the called procedure. The Return instruction is issued to give back control to the caller compartment. It retrieves the caller’s name and return address from the protected call stack and resumes execution.

**Simple secure compiler**

We employ a very simple compiler that is nonetheless interesting in two ways. First, the compiler plays a role in secure compartmentalization by being defensive regarding register values flowing from and to other compartments. Second, it implements an optimization that distinguishes cross-component calls from component-local calls and consequently removes overhead on local calls and returns.
Compiled components can obviously not rely on other compartments to preserve any invariants about values stored in registers, so our compiler must store and restore their register environment upon cross-compartment calls and returns. In our simple setting storing the environment involves saving the current value of the call argument on the stack and writing the current stack pointer to a fixed location in the current compartment’s memory. Moreover, in order to be secure our compiler must ensure that compromised compartments cannot distinguish compiled components based on low-level information that high-level fully defined attackers don’t get. In the high-level, only the call argument or return value is communicated upon calls and returns, therefore the compiler must clear the registers it has used before issuing a Call or Return instruction. This prevents compromised compartments from reading intermediate computation results of the previous compartments.

Our proof actually relies on compiled components cleaning all registers but the one that holds the call argument or return value. Not cleaning unused registers would provide a covert channel for two compromised compartments between which interfaces would forbid any direct communication. These compartments could now exchange values through uncleaned 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 very simple setting. In a setting where registers could be used to transmit capabilities, however, this would give more power to the attacker, so our compiler clears all registers but one, which also simplifies our proof.

The compiler partitions each compiled component’s memory into buffers, procedures’ code, and a local stack that can grow infinitely. This local stack is used both to store intermediate results and as a call stack. Because all procedures of a component will live in the same address space and local calls need not be monitored by the compartmentalization mechanism, these calls can be implemented more efficiently using regular Jal and Jump instructions. To this end, we use different procedure entry points for component-local and cross-component calls, and we skip, for local calls, the steps that store and restore the register environment and perform register cleaning.

Because we do not check bounds when compiling buffer read and write operations, buffer overflows can corrupt a compartment’s memory in arbitrary ways. Many buffer overflow attacks can be reproduced in our simple setting, including, due to the local calls optimization, return-oriented programming attacks [15], [58]. In return-oriented programming, an attacker overwrites return addresses on the local stack to produce an unexpected sequence of instructions of his will by reusing parts of the code of component-local procedures. In our setting, buffer overflow attacks thus enable compiled components that are not defensive enough to shoot themselves in the foot. However, as we will prove (Theorem 4.7), buffer overflows can only do limited harm to other compiled components.

We assume compiler correctness as stated below for our compiler (note that in the presence of partial type safety, Conjecture 4.1, proving either (1) or (2) below is enough to get the other).

**Assumption 4.2 (Compiler correctness).**

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

Combining this assumption with separate compilation (Definition 3.5), which trivially holds for our compiler, we immediately get the following corollary:

**Corollary 4.3 (Separate compiler correctness).**

\[ \forall s, A \vdash \circ s, \text{P } \vdash \bullet s. \text{A fully defined wrt } s \Rightarrow \]
\[ (1) \text{A[P] terminates } \iff \text{A}_\downarrow \text{ [P] terminates } \land \]
\[ (2) \text{A[P] diverges } \iff \text{A}_\downarrow \text{ [P] diverges } \]

**Proof technique for structured full abstraction** Trace semantics were initially proposed by Jeffrey and Rathke [31], [32] to define fully abstract models for high-level languages via may testing. Patrignani et al. later showed how to use such a trace semantics [52] to prove full abstraction for a compiler targeting machine code [51].

The proof technique has two steps: First, one devises a trace semantics for low-level partial programs and contexts and relates it to the target machine’s operational semantics—e.g. by proving it fully abstract [52]. Second, we use the trace semantics to characterize the interaction between an arbitrary low-level context and two compiled programs this context distinguishes, resulting in two traces with a common prefix followed by different actions. We map these traces to a high-level attacker, proving that this attacker distinguishes between the two source programs in the high level.

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 but sufficient 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 own trace semantics fully abstract [31], [32]. 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 [31], [32], [52].

Adapting the technique to undefined behavior takes little effort: It amounts to proving standard full abstraction for the safe subset of the language. So one simply proves that the context produced by the mapping is fully-defined, thus safe. Moving to a closed world, however, can take more work. The trace mapping argument of Patrignani et al. [51] for instance relies on changes in the context’s interface: for simplicity, they choose a maximal interface that lets the context import every declaration it might need, and they also add a helper component to the context that had no existence in the low level. This is no longer possible for structured full abstraction where the original context shape is fixed and we thus need to provide a mapping argument that preserves the context shape.
We achieve this by making the trace semantics more fine-grained: We track internal cross-component actions within the program and the context, and use interfaces to constrain the actions (both internal and external) that can be taken. This way, we ensure that every compartment’s actions adhere to the fixed interface of the original component. This change models the fact that, with our compartmentalization, the context cannot call every procedure exported by a program component. At any point, it can only call each program procedure for which there is a chain of internal actions going from the component used as a context entry point to some other context component whose interface explicitly imports the procedure.

**Trace semantics for the low-level language** We define a trace semantics where traces are words over an alphabet $E\alpha$ and alternate between program external actions “$!\gamma$” and context external actions “$?\gamma$”. Seeing external actions as moves in a game, we say that the context and the partial program are players and each other’s opponent.

$$E\alpha ::= \gamma? | \gamma! \quad \gamma ::= \text{Call}_{\text{reg}} C \ P | \text{Return}_{\text{reg}} \ | \ ✓$$

Possible external actions are program termination ($\gamma = ✓$), to which the opponent cannot respond ($✓$ ends the game), and cross-boundary communication actions ($\gamma \neq ✓$), which give control to the opponent. At any point where it has control, a player can take internal steps that are not reflected in the trace. In particular, a player can end the game by triggering an infinite sequence of internal steps and making execution diverge, which corresponds to not making any move.

Intuitively, a program $p$ has trace $t$ if it answers with the program actions described in $t$ when facing a context $a$ that produces the context actions described in $t$. We make no formal distinction between partial programs and contexts, so asking that $a$ produces the context actions described in $t$ is the same as asking that $a$ (seen as a partial program) produces the program actions in $t^{-1}$ when facing $p$, where $t^{-1}$ is obtained by swapping “!” for “?” and “?” for “!” in $t$. The intuition is that, from $a$’s point of view, $a$ is the program and $p$ is the context. We define $Tr(p)$ to be the set of traces of a partial program $p$, and make use of this definition to talk about the traces of a context: context $a$ produces the context actions in trace $t$ when facing a program $p$ that produces the program actions in $t$ if and only if $t^{-1} \in Tr(a)$.

The trace semantics is deterministic with respect to program actions and nondeterministic with respect to context actions. All possible actions an actual context could take have a corresponding nondeterministic choice, which is formalized as a property called trace extensibility.

**Assumption 4.4 (Trace extensibility).**

$$\forall t, s, p \vdash_\bullet s, a \vdash_\circ s. \ t \in Tr(p) \land (t, ?\gamma) \in Tr(a) \Rightarrow (t, \gamma?) \in Tr(p)$$

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

**Lemma 4.5 (Trace decomposition).**

$$\forall s, p \vdash_\bullet s, a \vdash_\circ s. \ a[p] \text{ terminates } \Rightarrow \exists t. \ t \text{ ends with } ✓ \land t \in Tr(p) \land t^{-1} \in Tr(a)$$

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 dual traces $t$ and $t^{-1}$. Because execution terminates, these traces end with a termination marker.

**Lemma 4.6 (Trace composition).**

$$\forall t, s, p \vdash_\bullet s, a \vdash_\circ s. \ t \in Tr(p) \land t^{-1} \in Tr(a) \Rightarrow (\forall E\alpha. \ (t. E\alpha) \notin Tr(p) \lor (t. E\alpha)^{-1} \notin Tr(a)) \Rightarrow (a[p] \text{ terminates } \iff t \text{ ends with } ✓)$$

Trace composition is the opposite of trace decomposition, and reconstructs a sequence of reductions based on synthesized interaction information: It considers a program and a context with dual shapes, such that one has trace $t$ and the other has the dual trace $t^{-1}$. The condition on the second line states that interaction is over: trace $t$ cannot be extended by any action $E\alpha$ while having the dual $(t. E\alpha)^{-1}$ in $Tr(a)$. Under these assumptions, trace composition tells us that one of the following holds: either (1) the trace ends with a termination marker $✓$ and in this case putting $p$ in context $a$ will produce a terminating program, or (2) putting $p$ in context $a$ will produce a diverging program. Intuitively, if interaction is over but there is no termination marker, it must be because the program or the context went into an infinite sequence of internal actions and will neither give control back nor terminate.

While the statement of these lemmas is very close to that used in an open world setting [31], [32], we had to adapt the trace semantics itself in order to prove them despite our closed world assumption. We did this by incorporating internal actions within the trace semantics, thus adding more options to the nondeterministic choice of the next context action. When in control, a player can now only perform communicating actions allowed by the interface of the current component. This restricts external actions as needed, while also enabling to internally switch the current component through allowed internal actions. So we use finer-grained traces that include internal communication and can later 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.

**Secure compartmentalization proof** We prove structured full abstraction, which implies secure compartmentalization by our general result from §3 (Theorem 3.4).

**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 communicated in registers are part of external 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. A canonical trace \( \zeta(t) \) can be obtained from an arbitrary trace \( t \) by rewriting context external actions so that the context always clears registers. We call this operation canonicalization.

As we will see, being able to reproduce arbitrary canonical traces is enough for the high-level context to distinguish the programs. The reason is that because they can’t trust other compartments, compiled fully defined components never read communicated values except for the one stored in the register used for communication. As a consequence, such 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 \( \zeta(t) \), as formalized by Assumption 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.

**Assumption 4.8 (Canonicalization).**

\[
\forall t, s, P \vdash_{\bullet} s. \ P \text{ fully defined wrt } s \Rightarrow t \in \text{Tr}(P|_{\bullet}) \iff \zeta(t) \in \text{Tr}(P|_{\bullet})
\]

The definability assumption below gives the characterization of our mapping from a canonical trace \( t \) and an action \( \gamma_0 \) to a compiled fully defined context \( A|_{\gamma} \) that reproduces the context actions in \( t \) and, depending on the next action \( \gamma \) the program takes, ends interaction with either termination (if \( \zeta(\gamma) = \zeta(\gamma_0) \)) or divergence (if \( \zeta(\gamma) \neq \zeta(\gamma_0) \)). The context \( A|_{\gamma} \) will thus distinguish a program \( P \) producing trace \( \cdot \ treating \cdot \) from any program producing \( \cdot \ treating \cdot \) with \( \zeta(\gamma) \neq \zeta(\gamma_0) \).

**Assumption 4.9 (Definability).**

\[
\forall t, \gamma_0, s. \ t = \zeta(t) \wedge (\exists p \vdash_{\bullet} s. \ (t, \gamma_0) \in \text{Tr}(p)) \Rightarrow \\
\exists A \vdash_{\gamma} s. \ A \text{ fully defined wrt } \bullet s \wedge \\
(1) t^{-1} \in \text{Tr}(A|_{\gamma}) \wedge \\
(2) \gamma_0 \neq \gamma \Rightarrow ((t, \gamma_0)! \cdot \gamma')^{-1} \in \text{Tr}(A|_{\gamma}) \wedge \\
(3) \forall \gamma, \zeta(\gamma) \neq \zeta(\gamma_0) \Rightarrow \forall \gamma', ((t, \gamma)! \cdot \gamma')^{-1} \notin \text{Tr}(A|_{\gamma})
\]

The definability assumption gives us a fully defined context that follows the dual trace \( t^{-1} \) (1) and that if given control afterwards via action \( \gamma' \) such that \( \gamma \neq \gamma' \) acts as follows: if \( \gamma = \gamma_0 \) the context terminates (2) and if the context can distinguish \( \gamma \) from \( \gamma_0 \), it will make execution diverge by not issuing any action \( \gamma' \) (3). As a compiled fully defined context, \( A|_{\gamma} \) can only access communicated values from register \( r_{com} \), the register that holds the call argument or return value. So \( A|_{\gamma} \) can only distinguish between \( \gamma \) and \( \gamma_0 \) when they differ in \( r_{com} \), which is captured formally by the \( \zeta(\gamma) \neq \zeta(\gamma_0) \) condition.

We built an algorithm (in OCaml) that constructs \( A \) out of \( t \). More precisely, the algorithm inputs a trace with internal cross-component actions (the finer-grained trace that erases to \( t \)) and builds an \( A \) that successfully reproduces context internal and external actions as prescribed by that trace. If, after following \( t \), the program gives back control to \( A \) by issuing some action \( \cdot \ treating \cdot \), \( A \) chooses a different appropriate response depending on the point in \( A \) to which action \( \cdot \ treating \cdot \) leads control flow. Because the trace we input already includes internal actions, we do not have to reconstruct them, hence our algorithm was not more difficult to come with than one for an open-world setting [51]. In the following, we assume that the algorithm is correct, i.e. that Assumption 4.9 holds. We can now prove our main theorem.

**Detailed proof of structured full abstraction.** Consider a low-level attacker \( a \vdash_{\circ} s \) distinguishing two fully defined partial programs with a dual shape \( P, Q \vdash_{\bullet} s. \ Suppose wlog that \( a|_{P|_{\bullet}} \) terminates and \( a|_{Q|_{\bullet}} \) diverges. We will build a high-level attacker \( A \vdash_{\circ} s \) 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 \( P|_{\bullet} \) to get a trace \( t_i \) of \( P|_{\bullet} \) that ends with \( \cdot \ treating \cdot \), such that \( t_{i-1} \in \text{Tr}(a) \). Call \( t_p \) the longest prefix of \( t_i \) such that \( t_p \in \text{Tr}(Q|_{\circ}) \). Because trace sets are prefix-closed by construction, we know that \( t_p \in \text{Tr}(P|_{\bullet}) \) and \( t_{i-1} \in \text{Tr}(a) \).

Moreover, \( t_p \) is necessarily a strict prefix of \( t_i \); otherwise, we could apply trace composition (Lemma 4.6) and get that \( a|_{Q|_{\bullet}} \) terminates, which is false. So there exists an external action \( E_{\alpha} \) such that trace \( \cdot \ treating \cdot , E_{\alpha} \cdot \) is a prefix of \( t_i \). Now \( E_{\alpha} \) cannot be a context action, or else trace extensibility (Assumption 4.4) would imply that \( \cdot \ treating \cdot , E_{\alpha} \cdot \) is a trace of \( \text{Tr}(Q|_{\circ}) \), which is incompatible with \( t_p \) being the longest prefix of \( t_i \) in \( \text{Tr}(Q|_{\circ}) \). Therefore, \( E_{\alpha} \) is a program action, i.e. there exists \( \gamma_0 \) such that \( \cdot \ treating \cdot , E_{\alpha} \cdot \) = \( \cdot \ treating \cdot , \gamma_0 \cdot \). Intuitively, \( P|_{\bullet} \) and \( Q|_{\circ} \) take the same external actions until the end of \( t_p \), where \( P|_{\bullet} \) takes external action \( \cdot \ treating \cdot , \gamma_0 \cdot \) and \( Q|_{\circ} \) does not (it either takes a different action \( \gamma \neq \gamma_0 \), or no external action at all).

Now, let \( t_c \) be the canonicalization of trace \( t_p \), i.e. \( t_c = \zeta(t_p) \). By canonicalization (Assumption 4.8), \( \cdot \ treating \cdot , \cdot \ treating \cdot , \gamma_0 \cdot \) is a trace of \( P|_{\bullet} \). We can thus use apply definability (Assumption 4.9) to trace \( t_c \) and action \( \gamma_0 \), using \( P|_{\bullet} \vdash_{\bullet} s \) as a witness having trace \( \cdot \ treating \cdot , \cdot \ treating \cdot , \gamma_0 \cdot \). This yields a fully defined context \( A \vdash_{\circ} s \) such that:

\[
(1) t_{c-1}^{-1} \in \text{Tr}(A|_{\gamma}) \\
(2) \gamma_0 \neq \gamma \Rightarrow ((t_c, \gamma_0)! \cdot \gamma')^{-1} \in \text{Tr}(A|_{\gamma}) \\
(3) \forall \gamma, \zeta(\gamma) \neq \zeta(\gamma_0) \Rightarrow \forall \gamma', ((t_c, \gamma)! \cdot \gamma')^{-1} \notin \text{Tr}(A|_{\gamma})
\]

We now show that conditions (1), (2) and (3) ensure that \( A|_{ \cdot \ treating \cdot } \) terminates while \( A|_{ \cdot \ treating \cdot , \gamma_0 \cdot } \) diverges.

First, we look at \( A|_{ \cdot \ treating \cdot } \). Consider the case where \( \gamma_0 = \gamma \). In this case, by applying trace extensibility to \( A|_{ \cdot \ treating \cdot } \) in (1) (seeing \( A|_{ \cdot \ treating \cdot } \) as a partial program), we get that \( (t_c, \gamma)!^{-1} \) is a trace of \( A|_{ \cdot \ treating \cdot } \), so trace composition allows us to conclude that \( A|_{ \cdot \ treating \cdot , \gamma_0 \cdot } \) diverges.
terminates. Now if $\gamma_0 \neq \checkmark$ then this action gives back control to the context, which, given (2), will perform action “$\checkmark$?”.
Applying trace extensibility to $P_{\downarrow}$, $P_{\downarrow}$ has trace “$t_0, \gamma_0!$. $\checkmark$?”
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 $t_p$, “$t_p, \checkmark$!” and “$t_p, \gamma_0$!”:

(a) $t_c = \zeta(t_p) \in Tr(Q_{\downarrow})$
(b) $(t_c, \checkmark!) = \zeta(t_p), \checkmark! \in Tr(Q_{\downarrow}) \Rightarrow (t_p, \checkmark!) \in Tr(Q_{\downarrow})$
(c) $(t_c, \gamma_0!) = \zeta(t_p), \gamma_0! \in Tr(Q_{\downarrow}) \Rightarrow (t_p, \gamma_0!) \in Tr(Q_{\downarrow})$

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 “$t_p, \checkmark$!” get that $a(Q_{\downarrow})$ terminates, which is false. $Q_{\downarrow}$ cannot perform action $\gamma_0$ either, since (c) would then violate the fact that $t_p$ is the longest prefix of $t_c$ in $Tr(Q_{\downarrow})$. So $Q_{\downarrow}$ only has two options left.

The first option of $Q_{\downarrow}$ 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 $\gamma$ so that $\checkmark \neq \gamma \neq \gamma_0$. Because fully defined compiled programs clean registers, they only yield canonical actions, i.e. $\gamma = \zeta(\gamma) \wedge \gamma_0 = \zeta(\gamma_0)$. Combined with (3), this entails that if $A_{\downarrow}$ produced an action $\gamma'$, we would have $\gamma = \gamma_0$, 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.

6 Conclusion and Future Work

We have introduced a new secure compartmentalization property, related it to the more established notion of full abstraction, and applied this property in a carefully simplified setting: a simple imperative language with procedures compiling to a compartmentalized abstract machine. This lays the formal foundations for studying the secure low-level interaction of mutually distrustful components.

In the future we plan to use 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 variant of CompCert and proving its security formally. We would like to implement the abstract compartmentalization machine from §4 in terms of various enforcement mechanisms, including: process-level sandboxing [14], [28], [34], [55], software-fault isolation (SFI) [64], micro-policies [13], capability machines [62], and multi-PMA systems [54]. As we target lower-level machines, new problems will appear: for instance trying to hide components’ size and resource consumption will likely be too costly, and we will need to weaken our property to allow controlled release of such information to the attacker. 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 are grateful to André DeHon 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


[63] C. Williams. Oracle’s Larry Ellison claims his Sparc M7 chip is hacker-proof — errr... The Register, 2015.

