We propose a new formal criterion for evaluating secure compilation schemes for unsafe languages, expressing end-to-end security guarantees for software components that may become compromised after encountering undefined behavior—for example, by accessing an array out of bounds.

Our criterion is the first to model dynamic compromise in a system of mutually distrustful components with clearly specified privileges. It articulates how each component should be protected from all the others—in particular, from components that have encountered undefined behavior and become compromised. Each component receives secure compilation guarantees—in particular, its internal invariants are protected from compromised components—up to the point when this component itself becomes compromised, after which we assume an attacker can take complete control and use this component’s privileges to attack other components. More precisely, a secure compilation chain must ensure that a dynamically compromised component cannot break the safety properties of the system at the target level any more than an arbitrary attacker-controlled component (with the same interface and privileges, but without undefined behaviors) already could at the source level.

To illustrate the model, we construct a secure compilation chain for a small unsafe language with buffers, procedures, and components, targeting a simple abstract machine with built-in compartmentalization. We give a machine-checked proof in Coq that this compiler satisfies our secure compilation criterion. Finally, we show that the protection guarantees offered by the compartmentalized abstract machine can be achieved at the machine-code level using either software fault isolation or a tag-based reference monitor.

We want a characterization that supports source-level security reasoning, allowing programmers to reason about the security properties of their code without knowing anything about the complex internals of the compilation chain (compiler, linker, loader, runtime system, system software, etc). What makes this particularly challenging for C and C++ programs is that they may encounter undefined behaviors—situations that have no source-level meaning whatsoever. Compilers are allowed to assume that undefined behaviors never occur in programs, and they aggressively exploit this assumption to produce the fastest possible code for well-defined programs, in particular by avoiding the insertion of run-time checks. For example, memory safety violations [16, 76] (e.g., accessing an array out of bounds, or using a pointer after its memory region has been freed) and type safety violations [28, 37] (e.g., invalid unchecked casts)—cause real C compilers to produce code that behaves arbitrarily, often leading to exploitable vulnerabilities [39, 76].

Of course, not every undefined behavior is necessarily exploitable. However, for the sake of strong security guarantees, we make a worst-case assumption that any undefined behavior encountered within a component can lead to its compromise. Indeed, in the remainder of the paper we equate the notions of “encountering undefined behavior” and “becoming compromised.”
While the dangers of memory safety and casting violations are widely understood, the C and C++ standards [41] call out large numbers of undefined behaviors [38, 51] that are less familiar, even to experienced C/C++ developers [56, 83]. To minimize programmer confusion and lower the risk of introducing security vulnerabilities, real compilers generally give sane and predictable semantics to some of these behaviors. For example, signed integer overflow is officially an undefined behavior in standard C, but many compilers (at least with certain flags set) guarantee that the result will be calculated using wraparound arithmetic. Thus, for purposes of defining secure compilation, the set of undefined behaviors is effectively defined by the compiler at hand rather than by the standard.

The purpose of a compartmentalizing compilation chain is to ensure that the arbitrary, potentially malicious, effects of undefined behavior are limited to the component in which it occurs. For a start, it should restrict the spatial scope of a compromise to the component that encounters undefined behavior. Such compromised components can only influence other components via controlled interactions respecting their interfaces and the other abstractions of the source language (e.g., the stack discipline on calls and returns). Moreover, to model dynamic compromise and give each component full guarantees as long as it has not yet encountered undefined behavior, the temporal scope of compromise must also be restricted. In particular, compiler optimizations should never cause the effects of undefined behavior to show up before earlier “observable events” such as system calls. Unlike the spatial restriction, which requires some form of run-time enforcement in software or hardware, the temporal restriction can be enforced just by foregoing certain aggressive optimizations. For example, the temporal restriction (but not the spatial one) is already enforced by the CompCert C compiler [58, 70], providing a significantly cleaner model of undefined behavior than other C compilers [70].

We want a characterization that is formal—that brings mathematical precision to the security guarantees and attacker model of compartmentalizing compilation. This can serve both as a clear specification for verified secure compilation chains and as useful guidance for unverified ones. Moreover, we want the characterization to provide source-level reasoning principles that can be used to assess the security of compartmentalized applications. To make this feasible in practice, the amount of source code to be verified or audited has to be relatively small. So, while we can require developers to carefully analyze the privileges of each component and the correctness of some very small pieces of security-critical code, we cannot expect them to establish the full correctness—or even absence of undefined behavior—for most of their components.

Our secure compilation criterion improves on the state of the art in three important respects. First, our criterion applies to compartmentalized programs, while most existing formal criteria for secure compilation are phrased in terms of protecting a single trusted program from an untrusted context [2, 4, 5, 7–9, 31, 65]. Second, unlike some recent criteria that do consider modular protection [25, 67], our criterion applies to unsafe source languages with undefined behaviors. And third, it considers a dynamic compromise model—a critical advance over the recent proposal of Juglaret et al. [45], which does consider components written in unsafe languages, but which is limited to a static compromise model. This is a serious limitation: components whose code contains any vulnerability that might potentially manifest itself as undefined behavior are given no guarantees whatsoever, irrespective of whether an attacker actually exploits these vulnerabilities. Moreover, vulnerable components lose all guarantees from the start of the execution—possibly long before any actual compromise. Experience shows that large enough C or C++ codebases essentially always contain vulnerabilities [76]. Thus, although static compromise models may be appropriate for safe languages, they are not useful for unsafe low-level languages.

As we will see in §5, the limitation to static compromise scenarios seems inescapable for previous techniques, which are all based on the formal criterion of full abstraction [2]. To support dynamic compromise scenarios, we take an unconventional approach, dropping full abstraction and instead phrasing our criterion in terms of preserving safety properties [54] in adversarial contexts [7], where, formally, safety properties are predicates over execution traces that are informative enough to detect the compromise of components and to allow the execution to be “rewound” along the same trace. Moving away from full abstraction also makes our criterion easier to achieve efficiently in practice and to prove at scale.

Contributions. Our first contribution is Robustly Safe Compartmentalizing Compilation (RSCC), a new secure compilation criterion articulating strong end-to-end security guarantees for components written in unsafe languages with undefined behavior. This criterion is the first to support dynamic compromise in a system of mutually distrustful components with clearly specified privileges. We start by illustrating the intuition, informal attacker model, and source-level reasoning behind RSCC using a simple example application (§2).

Our second contribution is a formal presentation of RSCC. We start from Robustly Safe Compilation (RSC, §3.1), a simple security criterion recently introduced by Abate et al. [7], and extend this first to dynamic compromise (RSC\textsubscript{DC}, §3.2), then mutually distrustful components (RSC\textsubscript{MD} \textsubscript{DC} §3.3), and finally to the full definition of RSCC (§3.4). We also give an effective and generic proof technique for RSCC (§3.5): We start with a target-level execution and explain any finite sequence of calls and returns in terms of the source language by constructing a whole source program that produces this prefix. Our novel proof architecture manages to achieve this using only (mostly standard) simulation proofs, which is much simpler and more scalable than previous proofs in this space [45]. One particularly important advantage is that it allows us to reuse a whole-program compiler correctness result à la CompCert [58] as a black box, avoiding the need to prove any other simulations between the source and target languages.

Our third contribution is a proof-of-concept secure compilation chain (§4) for a simple unsafe sequential language featuring buffers, procedures, components, and a CompCert-like block-based memory model [59] (§4.1). Our entire compilation chain is implemented in the Coq proof assistant. The first step compiles our source language to a simple low-level abstract machine with built-in compartmentalization (§4.2). We use the proof technique from §3.5 to construct machine-checked proofs in Coq showing that this compiler satisfies RSCC (§4.3). Finally, we describe two back ends for our compiler, showing that the protection guarantees of the compartmentalized abstract machine can be achieved at the lowest level using either software fault isolation (SFI, §4.4) or a tag-based reference monitor (§4.5). The tag-based back end, in particular, is novel, using linear
return capabilities to enforce a cross-component call/return discipline. Neither back end has yet been formally verified, but we have used property-based testing to gain confidence that the SFI back end satisfies RSC\textsuperscript{DC}\textsuperscript{MD}.

These contributions lay a solid foundation for future secure compilation chains that could bring sound and practical compartmentalization to C, C++, and other unsafe low-level languages. We address three fundamental questions: (1) What is the desired secure compilation criterion and to what attacker model and source-level security reasoning principles does it correspond? Answer: We propose the RSCC criterion from §2–§3. (2) How can we effectively enforce secure compilation? Answer: Various mechanisms are possible; the simple compilation chain from §4 illustrates how either software fault isolation or tagged-based reference monitoring can enforce RSCC. (3) How can we achieve high assurance that the resulting compilation chain is indeed secure? Answer: We show that formal verification (§4.3) and property-based testing (§4.4) can be successfully used together for this in a proof assistant like Coq.

We close with related (§5) and future (§6) work. The appendix presents omitted details. Our Coq development is available at https://github.com/secure-compilation/when-good-components-go-bad/

A previous version of this work has been published in conference proceedings [6]. The current version has been extended to include the details of the proofs, and, compared to the conference version, more proofs are now fully mechanized in Coq, and our proof architecture has also been significantly simplified.

## 2 RSCC BY EXAMPLE

We begin by an overview of compartmentalizing compilation chains, our attacker model, and how viewing this model as a dynamic compromise game leads to intuitive principles for security analysis.

We need not be very precise, here, about the details of the source language; we just assume that it is equipped with some compartmentalization facility [35, 81] that allows programmers to break up security-critical applications into mutually distrustful components that have clearly specified privileges and can only interact via well-defined interfaces. In fact we assume that the interface of each component gives a precise description of its privilege. The notions of component and interface that we use for defining the secure compilation criteria in §3 are quite generic: interfaces can include any requirements that can be enforced on components, including type signatures, lists of allowed system calls, or more detailed access-control specifications describing legal parameters to cross-component calls (e.g., ACLs for operations on files). We assume that the division of an application into components and the interfaces of those components are statically determined and fixed. For the illustrative language of §4, we will use a simple setup in which components don’t directly share state, interfaces just list the procedures that each component provides and those that it expects to be present in its context, and the only thing one component can do to another one is to call procedures allowed by their interfaces.

The goal of a compartmentalizing compilation chain is to ensure that components interact according to their interfaces even in the presence of undefined behavior. Our secure compilation criterion does not fix a specific mechanism for achieving this: responsibility can be divided among the different parts of the compilation chain, such as the compiler, linker, loader, runtime system, system software, and hardware. In §4 we study a compilation chain with two alternative back ends—one using software fault isolation and one using tag-based reference monitoring for compartmentalization. What a compromised component can still do in this model is to use its access to other components, as allowed by its interface, to either trick them into misusing their own privileges (i.e., confused deputy attacks) or even compromise them as well (e.g., by sending them malformed inputs that trigger control-hijacking attacks exploiting undefined behaviors in their code).

We model input and output as interaction with a designated environment component E that is given an interface but no implementation. When invoked, environment functions are assumed to immediately return a non-deterministically chosen value [58]. In terms of security, the environment is thus the initial source of arbitrary, possibly malformed, inputs that can exploit buffer overflows and other vulnerabilities to compromise other components.

As we argued in the introduction, it is often unrealistic to assume that we know in advance which components will be compromised and which ones will not. This motivates our model of dynamic compromise, in which each component receives secure compilation guarantees until it becomes compromised by encountering an undefined behavior, causing it to start attacking the remaining uncompromised components. In contrast to earlier static-compromise models [45], a component only loses guarantees in our model after an attacker discovers and manages to exploit a vulnerability, by sending it inputs that lead to an undefined behavior. The mere existence of vulnerabilities—undefined behaviors that can be reached after some sequence of inputs—is not enough for the component to be considered compromised.

This model allows developers to reason informally about various compromise scenarios and their impact on the security of the whole application [35]. If the consequences of some plausible compromise seem too serious, developers can further reduce or separate privileges by narrowing interfaces or splitting components, or they can make components more defensive by validating their inputs.

As a first running example, consider the idealized application in Figure 1. It defines three components (C\textsubscript{0}, C\textsubscript{1}, and C\textsubscript{2}) that interact with the environment E via input (E.read) and output (E.write) operations. Component C\textsubscript{1} defines a main() procedure, which first invokes C\textsubscript{2}.init() and then reads a request x from the environment (e.g., coming from some remote client), parses it by calling an internal procedure to obtain y, and then invokes C\textsubscript{2}.process(x,y). This, in turn, calls C\textsubscript{2}.prepare() and C\textsubscript{2}.handle(y), obtaining some data that it validates using C\textsubscript{0}.valid and, if this succeeds, writes data together with the original request x to the environment.

Suppose we would like to establish two properties:

(S\textsubscript{1}) any call E.write(<data,x>) happens as a response to a previous E.read() call by C\textsubscript{1} obtaining the request x; and

(S\textsubscript{2}) the application only writes valid data (i.e., data for which C\textsubscript{0}.valid returns true).

These can be shown to hold of executions that do not encounter undefined behavior simply by analyzing the control flow. But what if undefined behavior does occur? Suppose that we can rule out this possibility—by auditing, testing, or formal verification—for some parts of the code, but we are unsure about three subroutines:
When Good Components Go Bad

Abate et al.

component $C_0$

export valid;
valid(data) { ... }
}

component $C_1$

import E.read, C_2.init, C_2.process;
maint() {
  C_2.init();
x := E.read();
y := C_1.parse(x); // (V_1) can yield Undefined for some x
C_2.process(x,y);
}
parse(x) { ... }

component $C_2$

import E.write, C_0.valid;
export init, process;
init() { ... }
process(x,y) {
  // (V_2) can yield Undefined if not initialized
  data := C_2.handle(y); // (V_1) can yield Undefined for some y
  if C_0.valid(data) then E.write(<data,x>)
}
prepare() { ... }
handle(y) { ... }

Figure 1: Pseudocode of compartmentalized application

(V_1) $C_1.parse(x)$ performs complex array computations, and we do not know if it is immune to buffer overflows for all x;
(V_2) $C_2.prepare()$ is intended to be called only if $C_2.init()$ has been called beforehand to set up a shared data structure; otherwise, it might dereference an undefined pointer;
(V_3) $C_2.handle(y)$ might cause integer overflow on some inputs.

If the attacker finds an input that causes the undefined behavior in V_1 to occur, then C_1 can get compromised and call C_2.process(x,y) with values of x that it hasn’t received from the environment, thus invalidating S_1. Nevertheless, if no other undefined behavior is encountered during the execution, this attack cannot have any effect on the code run by C_2, so S_2 remains true.

Now consider the possible undefined behavior from V_2. If C_1 is not compromised, this undefined behavior cannot occur, since $C_2.init()$ will be called before $C_2.prepare()$. Moreover, this undefined behavior cannot occur even if C_2 is compromised by the undefined behavior in V_1, because that can only occur after $C_2.init()$ has been called. Hence V_1 and V_2 together are no worse than V_1 alone, and property S_2 remains true. Inferring this crucially depends on our model of dynamic compromise, in which C_1 can be treated as honest and gets guarantees until it encounters undefined behavior. If instead we were only allowed to reason about C_1’s ability to do damage based on its interface, as would happen in a model of static compromise [45], we wouldn’t be able to conclude that C_2 cannot be compromised: an arbitrary component with the same interface as C_1 could indeed compromise C_2 by calling C_2.process before C_2.init. Finally, if execution encounters undefined behavior in V_3, then C_2 can get compromised irrespective of whether C_1 is compromised beforehand, invalidating both S_1 and S_2.

Though we have not yet made it formal, this security analysis already identifies C_2 as a single point of failure for both desired properties of our system. This suggests several ways the program could be improved: The code in C_2.handle could be hardened to reduce its chances of encountering undefined behavior, e.g. by doing better input validation. Or C_1 could validate the values it sends to $C_2.process$, so that an attacker would have to compromise both C_1 and C_2 to break the validity of writes. To ensure the correspondence of reads and writes despite the compromise of C_1, we could make C_2 read the request values directly from E, instead of via C_1.

To achieve the best security though, we can refactor so that the read and write privileges are isolated in C_0, which performs no complex data processing and thus is a lot less likely to be compromised by undefined behavior (Figure 2). In this variant, C_0 reads a request, calls C_1.parse on this request, passes the result to $C_2.process$, validates the data C_2 returns and then writes it out. This way both our desired properties hold even if both C_1 and C_2 are compromised, since now the core application logic and privileges have been completely separated from the dangerous data processing operations that could cause vulnerabilities.

Let’s begin making all this a bit more formal. The first step is to make the security goals of our example application more precise. We do this in terms of execution traces that are built from events such as cross-component calls and returns. The two intuitive properties from our example can be phrased in terms of traces as follows: If E.write(<data,x>) appears in an execution trace, then

(S_1) E.read was called previously and returned x, and (S_2) C_0.valid(data) was called previously and returned true.

The refactored application in Figure 2 achieves both properties despite the compromise of both C_1 via V_1 and C_2 via V_3, but, for the first variant in Figure 1 the properties need to be weakened as follows: If E.write(<data,x>) appears in an execution trace then

(W_1) E.read previously returned x or E.read previously returned an x’ that can cause undefined behavior in C_1.parse(x’) or $C_2.process(x,y)$ was called previously with a y that can cause undefined behavior in $C_2.handle(y)$, and (W_2) C_0.valid(data) was called previously and returned true or $C_2.process(x,y)$ was called previously with a y that can cause undefined behavior in $C_2.handle(y)$.

While these properties are significantly weaker (and harder to understand), they are still not trivial; in particular, they still tell us
something useful under the assumption that the attacker has not actually discovered how to compromise C₁ or C₂.

Properties S₁, S₂, W₁, W₂ are all safety properties \[54\]—inspired, in this case, by the sorts of “correspondence assertions” used to specify authenticity in security protocols \[33, 86\]. A trace property is a safety property if, within any (possibly infinite) trace that violates the property, there exists a finite “bad prefix” that violates it. For instance here is a bad prefix for S₂ that includes a call to E.write(</data,x>) with no preceding call to C₀.valid(data):

\[
[C₀ \text{ main}(1); C₂ \text{ init}(1); \text{Ret;} E \text{. read}; \text{Ret}(x); C₁ \text{ parse}(x); \\
\text{Ret}(y); C₂ \text{ process}(y); \text{Ret data}; E \text{. write}(</data,x>)]
\]

The program from Figure 2 cannot produce traces with this bad prefix, but it could do so if we removed the validity check in C₀.main(); this variant would invalidate safety property S₂.

Compiler correctness is often phrased in terms of preserving trace properties in general \[58\] (and thus safety properties as a special case). However, this is often predicated on the assumption that the source program has no undefined behavior; if it does, all security guarantees are lost, globally. By contrast, we want our secure compilation criterion to still apply even when some components are dynamically compromised by encountering undefined behavior. In particular, we want to ensure that dynamically compromised components are not able to break the safety properties of the system at the target level more than equally privileged components without undefined behavior already could in the source.

We call our criterion Robustly Safe Compartmentalizing Compilation (RSCC). It is phrased in terms of a “security game,” illustrated in Figure 3 for our running example. With an RSCC compilation chain, given any execution of the compiled and linked components C₀, C₁ and C₂ producing trace t in the target language, we can explain any (intuitively bad) finite prefix m of t (written m ≤ t) in terms of the source language. As soon as any component of the program has an undefined behavior though, the semantics of the source language can no longer directly help us. Similar to CompCert \[58\], we model undefined behavior in our source language as a special event Undef(C₀) that terminates the trace. For instance, in step 0 of Figure 3, component C₁ is the first to encounter undefined behavior after producing a prefix m₁ of m.

Since undefined behavior can manifest as arbitrary target-level behavior, the further actions of component C₁ can no longer be explained in terms of its source code. So how can we explain the rest of m in the source language? Our solution in RSCC is to require that one can replace C₁, the component that encountered undefined behavior, with some other source component A₁ that has the same interface and can produce its part of the whole m in the source language without itself encountering undefined behavior. In order to replace component C₁ with A₁ we have to go back in time and re-execute the program from the beginning after a longer trace, in this case m₂.Undef(C₂) (where we write `\(\rightarrow\)` for appending the event Undef(C₂) to m₂). We iterate this process until all components that encountered undefined behavior have been replaced with new source components that do not encounter undefined behavior and produce the whole m. In the example dynamic compromise scenario from Figure 3, this means replacing C₁ with A₁ and C₂ with A₂, after which the program can produce the whole prefix m in the source.

![Figure 3: The RSCC dynamic compromise game for our example.](image)

Let’s now use this RSCC security game to deduce that in our example from Figure 2, even compromising both C₁ and C₂ does not break property S₂ at the target level. Assume, for the sake of a contradiction, that a trace of our compiled program breaks property S₂. Then there exists a finite prefix \(m \cdot E \text{. write}(</data,x>)\) such that C₀.valid(data) does not appear in m. Using RSCC we obtain that there exists some dynamic compromise scenario explaining m in the source. The simplest case is when no components are compromised. The most interesting case is when this scenario involves the compromise of both C₁ and C₂ as in Figure 3. In this case, replacing C₁ and C₂ with arbitrary A₁ and A₂ with the same interfaces allows us to reproduce the whole bad prefix m in the source (step 2 from Figure 3). We can now reason in the source, either informally or using a program logic for robust safety \[75\], that this cannot happen, since the source code of C₀ does call C₀.valid(data) and only if it gets true back does it call E.write(</data,x>).
3 FORMALLY DEFINING RSCC

For pedagogical purposes, we define RSCC in stages, incrementally adapting the existing notion of Robustly Safe Compilation (RSC) introduced by Abate et al. [7] (and reviewed in §3.1). We first bring RSC to unsafe languages with undefined behavior (§3.2), and then further extend its protection to any set of mutually distrustful components (§3.3). These ideas lead to the more elaborate RSCC property (§3.4), which directly captures the informal dynamic compromise game from §2. These definitions are generic, and will be illustrated with a concrete instance in §4. Finally, we describe an effective and general proof technique for RSCC (§3.5).

3.1 RSC: Robustly Safe Compilation

RSC [7] is a recent criterion for secure compilation that captures the preservation of all robust safety properties—i.e., safety properties that hold in the presence of arbitrary adversarial contexts [33, 53, 75]. A trace property (i.e., a set of potentially infinite traces built over events like I/O with the environment [58]) is a safety property [54] if any trace violates it has a finite “bad prefix” that already violates it. We focus on robust safety since it captures many important program properties (e.g., robust partial correctness), while allowing for a simple secure-compilation proof technique (§3.5).

RSC is a property of a whole compilation chain: the source language and its trace-based big-step operational semantics (we write $P\rightarrow t$ to mean that the complete program $P$ can produce trace $t$), plus its compiler ($P \downarrow$), source and target linking (where $C_S[P]$ denotes a partial program $P$ with context $C_S$ to obtain a whole source program, and $C_T[P T]$ does the same in the target), and target-level semantics ($P T \rightarrow t$) including for instance the target machine, loader, and deployed protection mechanisms.

Definition 3.1. A compilation chain provides RSC iff

\[
\forall P \ C_T t. C_T[P T] \rightarrow t \Rightarrow \forall m \leq t. \exists C_S t'. C_S[P] \rightarrow t' \land m \leq t'.
\]

That is, RSC holds for a compilation chain if, for any partial source program $P$ and any target context $C_T$, where $C_T$ linked with the compilation of $P$ can produce a trace $t$ in the target ($C_T[P T] \rightarrow t$), and for any finite prefix $m$ of trace $t$ (written $m \leq t$), we can construct a source-level context $C_S$ that can produce prefix $m$ in the source language when linked with $P$ (i.e., $C_S[P] \rightarrow t'$ for some $t'$ so that $m \leq t'$). Intuitively, if we think of the contexts as adversarial and $m$ as a bad behavior, RSC says that any finite attack $m$ that a target context $C_T$ can mount against $P T$ can already be mounted against $P$ by some source context $C_S$. So proving RSC requires that we be able to back-translate each finite prefix $m$ of $C_T[P T]$ into a source context $C_S$ that performs $m$ together with the original program $P$. Conversely, any safety property that holds of $P$ when linked with an arbitrary source context will still hold for $P T$ when linked with an arbitrary target context [7].

As in CompCert, we assume that the traces are exactly the same in the source and target languages. We anticipate no trouble relating this to an arbitrary relation between source and target traces.

3.2 RSCDC: Dynamic Compromise

The RSC criterion above is about protecting a partial program written in a safe source language against adversarial target-level contexts. We now adapt the idea behind RSC to an unsafe source language with undefined behavior, in which the protected partial program itself can become compromised. As explained in §2, we model undefined behavior as a special $\text{Undef}$ event terminating the trace: whatever happens afterwards at the target level can no longer be explained in terms of the code of the source program. We further assume that each undefined behavior in the source language can be attributed to the part of the program that causes it by labeling the $\text{Undef}$ event with "blame the program" (P) or "blame the context" (C) (while in §3.3 we will blame the precise component encountering undefined behavior).

Definition 3.2. A compilation chain provides Robustly Safe Compilation with Dynamic Compromise (RSCDC) if

\[
\forall P \ C_T t. C_T[P T] \rightarrow t \Rightarrow \forall m \leq t. \exists C_S t'. C_S[P] \rightarrow t' \land (m \leq t' \land t' \leq \text{pm}).
\]

Roughly, this definition relaxes RSC by forgoing protection for the partial program $P$ after it encounters undefined behavior. More precisely, instead of always requiring that the trace $t'$ produced by $C_S[P]$ contain the entire prefix $m$ (i.e., $m \leq t'$), we also allow $t'$ to be itself a prefix of $m$ followed by an undefined behavior in $P$, which we write as $t' \leq \text{pm}$ (i.e., $t' \leq \text{pm} \land \exists m' \leq m. t' = (m' \cdot \text{Undef}(P))$). In particular, the context $C_S$ is guaranteed to be free of undefined behavior before the whole prefix $m$ is produced or $P$ encounters undefined behavior. However, nothing prevents $C_S$ from passing values to $P$ that try to trick $P$ into causing undefined behavior.

To illustrate, consider the partial program $P$ defined below.

```
program P | import E; write; export foo;
foo(x) {
    y := P.read(x);
    E.write(y);
    // can encounter Undef for some x
    process(x) [ ... ]
}
```

Suppose we compile $P$ with a compilation chain that satisfies RSCDC, link the result with a target context $C_T$ obtaining $C_T[P T]$, execute this and observe the following finite trace prefix:

```
m = E.read(); Ret("feedback"); P.foo("feedback"); E.write("bad")
```
According to \textit{RSC} there exists a source-level context $C_S$ (for instance the one above) that explains the prefix $m$ in terms of the source language in one of two ways: either $C_S[P]$ can do the entire $m$ in the source, or $C_S[P]$ encounters an undefined behavior in $P$ after a prefix of $m$, for instance the following one:

\[ t' = \text{[E.read()]; Ret('feedback'); P.foo('feedback'); Undef(P)} \]

As in CompCert \cite{58,70}, we treat undefined behaviors as observable events at the end of the execution trace, allowing compiler optimizations that move an undefined behavior to an earlier point in the execution, but not past any other observable event. While some other C compilers would need to be adapted to respect this discipline \cite{70}, limiting the temporal scope of undefined behavior is a necessary prerequisite for achieving security against dynamic compromise. Moreover, if trace events are coarse enough (e.g., system calls and cross-component calls) we expect this restriction to have a negligible performance impact in practice.

One of the top-level CompCert theorems does, in fact, already capture dynamic compromise in a similar way to \textit{RSC}. Using our notations this CompCert theorem looks as follows:

\[
\forall \ t. \ (P) \xRightarrow{\sim} t' \Rightarrow \exists t'. \ P \xRightarrow{\sim} t' \land (t' = t \lor t' < t)
\]

This says that if a compiled whole program $P$ can produce a trace $t$ with respect to the target semantics, then in the source $P$ can produce either the same trace or a prefix of $t$ followed by undefined behavior. In particular this theorem does provide guarantees to undefined programs up to the point at which they encounter undefined behavior. The key difference compared to our secure compilation chains is that CompCert does not restrict undefined behavior spatially in CompCert undefined behavior breaks all security guarantees of the whole program, while in our work we restrict undefined behavior to the component that causes it. This should become clearer in the next section, where we explicitly introduce components, but even in \textit{RSC} we can already imagine $P_1$ as a set of uncompromised components for trace prefix $m$, and $C_T$ as a set of already compromised ones.

A smaller difference with respect to the CompCert theorem is that (like \textit{RSC}) \textit{RSC} only looks at finite prefixes in order to simplify the difficult proof step of context back-translation, which is not a concern that appears in CompCert and the usual verified compilers. Appendix A precisely characterizes the subclass of safety properties that is preserved by \textit{RSC} even in adversarial contexts.

### 3.3 RSC\textsc{dc} MD: Mutually Distrustful Components

\textit{RSC} gives a model of dynamic compromise for secure compilation, but is still phrased in terms of protecting a trusted partial program from an untrusted context. We now adapt this model to protect any set of mutually distrustful components with clearly specified privileges from an untrusted context. Following Juglaret \textit{et al.}’s work in the full abstraction setting \cite{45}, we start by taking both partial programs and contexts to be sets of components; linking a program with a context is then just set union. We compile sets of components by separately compiling each component. Each component is assigned a well-defined interface that precisely captures its privilege; components can only interact as specified by their interfaces. Most importantly, context back-translation respects these interfaces: each component of the target context is mapped back to a source component with exactly the same interface. As Juglaret \textit{et al.} argue, least-privilege design crucially relies on the fact that, when a component is compromised, it does not gain any more privileges.

#### Definition 3.3. A compilation chain provides Robustly Safe Compilation with Dynamic Compromise and Mutual Distrust (\textit{RSC} \textsc{dc} \textsc{md}) if there exists a back-translation function $\uparrow$ taking a finite trace prefix $m$ and a component interface $I_i$ to a source component with the same interface, such that, for any compatible interfaces $I_P$ and $I_C$,

\[
\forall P, I_P. \ \forall C_T : I_C. \ \forall t. \ (C_T \cup [P]) \xRightarrow{\sim} t \Rightarrow \forall m \leq t.
\]

This definition closely follows \textit{RSC}, but it restricts programs and contexts to compatible interfaces $I_P$ and $I_C$. We write $P : I$ to mean “partial program $P$ satisfies interface $I$.” The source-level context is obtained by applying the back-translation function $\uparrow$ pointwise to all the interfaces in $I_C$. As before, if the prefix $m$ is cropped prematurely because of an undefined behavior, then this undefined behavior must be in one of the program components, not in the back-translation context components ($t' < i_P, m$).

### 3.4 Formalizing RSCC

Using these ideas, we now define \textit{RSCC} by following the dynamic compromise game illustrated in Figure 3. We use the notation $P \xRightarrow{\sim} m$ when there exists a trace $t$ that extends $m$ (i.e., $m \leq t$) such that $P \xRightarrow{\sim} t$. We start with all components being uncompromised and incrementally replace each component that encounters undefined behavior in the source with an arbitrary component with the same interface that may now attack the remaining components.

#### Definition 3.4. A compilation chain provides Robustly Safe Compartmentalizing Compilation (\textit{RSCC}) iff \textit{compatible} interfaces $I_1, ..., I_n$,

\[
\forall C_1 : I_1, ..., C_n : I_n. \ \forall m. \ \{C_1 \cup ..., C_n\} \xRightarrow{\sim} m \Rightarrow \exists A_{i_1}, ..., A_{i_k}, i_{i_k}.
\]

(1) $\forall j \in 1 ... k. \ \exists m_j. \ (m_j < A_{i_j} m) \land (m_j = 1 < A_{i_j-1} m_j) \land (\forall C_{i_1}, ..., C_{i_n} \{C_{i_1}', ..., C_{i_{j-1}}', C_{i_j} \cup A_{i_j}, ..., A_{i_k} \} \xRightarrow{\sim} m_j \land (2) \ (\forall C_{i_1}, ..., C_{i_k} \{C_{i_1}, ..., C_{i_k} \cup A_{i_1}, ..., A_{i_k} \} \xRightarrow{\sim} m).$

This says that $C_{i_1}, ..., C_{i_k}$ constitutes a compromise sequence corresponding to finite prefix $m$ produced by a compiled set of components $\{C_1 \cup ..., C_n\}$. In this compromise sequence each component $C_{i_j}$ is taken over by the already compromised components at that point in time $\{A_{i_1}, ..., A_{i_{j-1}}\}$ (part 1). Moreover, after replacing all the compromised components $\{C_{i_1}, ..., C_{i_k}\}$ with their corresponding source components $\{A_{i_1}, ..., A_{i_k}\}$ the entire $m$ can be reproduced in the source language (part 2).

This formal definition allows us to play an iterative game in which components that encounter undefined behavior successively become compromised and attack the other components. This is the first security definition in this space to support both dynamic compromise and mutual distrust, whose interaction is subtle and has eluded previous attempts at characterizing the security guarantees of compartmentalizing compilation as extensions of fully abstract compilation \cite{45} (further discussed in §5).

### 3.5 A Generic Proof Technique for RSCC

We now describe an effective and general proof technique for \textit{RSCC}. First, we observe that the slightly simpler \textit{RSCC} \textsc{md} implies \textit{RSCC}.
When Good Components Go Bad

Abate et al.

Then we provide a generic proof in Coq that a compilation chain obeys $RSC_{MD}$ if it satisfies certain well-specified assumptions on the source and target languages and the compilation chain.

Our proof technique yields simpler and more scalable proofs than previous work in this space [45]. In particular, it allows us to directly reuse a compiler correctness result à la CompCert, which supports separate compilation but only guarantees correctness for whole programs [47], which avoids proving any other simulations between the source and target languages. Achieving this introduces some slight complications in the proof structure, but it nicely separates the correctness and security proofs and allows us to more easily tap into the CompCert infrastructure. Moreover, this novel proof structure completely removes the need for any kind of "partial semantics” [7, 45] (a.k.a. “trace semantics” [43, 66]), allowing the proof to be done entirely with respect to the semantics of whole programs in the source and target languages. Finally, since only the last step of our proof technique is specific to unsafe languages, our technique could be further simplified to provide scalable proofs of vanilla $RSC$ for safe source languages [7, 68].

$RSC_{MD}$ implies $RSC_{DC}$  The first step in our proof technique reduces $RSC$ to $RSC_{DC}$, using a theorem showing that $RSC$ can be obtained by iteratively applying $RSC_{MD}$. This result crucially relies on back-translation in $RSC_{MD}$ being performed pointwise and respecting interfaces, as explained in §3.3.

Theorem 3.5. $RSC_{DC_{MD}}$ implies $RSC_{DC}$.

We proved this by defining a non-constructive function that produces the compromise sequence $A_{i_1}, ..., A_{i_t}$ by case analysis on the disjunction in the conclusion of $RSC_{DC_{MD}}$ (using excluded middle in classical logic). If $m \leq t'$ we are done and we return the sequence we accumulated so far, while if $t' < m$ we obtain a new compromised component $c_i : I_i$ that we back-translate using $(m, I_i) \uparrow$ and add to the sequence before iterating this process.

Generic $RSC_{DC_{MD}}$ proof outline  Our high-level $RSC_{DC_{MD}}$ proof is generic and works for any compilation chain that satisfies certain well-specified assumptions, which we introduce informally for now, leaving details to the end of this sub-section. The $RSC_{DC_{MD}}$ proof for the compiler chain in §4 proves all these assumptions.

The proof outline is shown in Figure 4. We start (in the bottom left) with a complete target-level program $C_T \cup P$ producing a trace with a finite prefix $m$ that we assume contains no undefined behavior (since we expect that the final target of our compilation will be a machine for which all behavior is defined). The prefix $m$ is first back-translated to synthesize a complete source program $C_S \cup P'$ producing $m$ (the existence and correctness of this back-translation are Assumption 1). For example, for the compiler in §4, each component $C_i$ produced by back-translation uses a private counter to track how many events it has produced during execution. Whenever $C_i$ receives control, following an external call or return, it checks this counter to decide what event to emit next, based on the order of its events on $m$ (see §4.3 for details).

The generated source program $C_S \cup P'$ is then separately compiled to a target program $C_S \downarrow \cup P' \downarrow$ that, by compiler correctness, produces again the same prefix $m$ (Assumption 2). Now from $(C_T \cup P) \Rightarrow \ast m$ and $(C_S \downarrow \cup P') \Rightarrow \ast m$ we would like to obtain $(C_S \downarrow \cup P) \Rightarrow \ast m$ by “recomposing” (Assumption 3) the components from the two executions and forming a new execution for $C_S \downarrow \cup P$.

We use a three-way simulation to relate the executions of the two complete programs, to a third execution that combines the program part of the first execution, and the context part of the second execution.

Once we know that $(C_S \downarrow \cup P) \Rightarrow \ast m$, we use compiler correctness again—in the backwards direction (Assumption 4)—to obtain an execution of the source program $C_S \cup P$ producing trace $t$. Because our source language is unsafe, however, $t$ need not be an extension of $m$; it can end earlier with an undefined behavior (§3.2). So the final step in our proof shows that if the source execution ends earlier with an undefined behavior ($t' \prec m$), then this undefined behavior can only be caused by $P$ (i.e., $t' \prec p m$), not by $C_S$, which was correctly generated by our back-translation (Assumption 5).

Assumptions of the $RSC_{DC_{MD}}$ proof  The generic $RSC_{DC_{MD}}$ proof outlined above relies on assumptions about the compartmentalization compilation chain. In the reminder of this subsection we give details about these assumptions, while still trying to stay at a high level by omitting some of the low-level details in our Coq formalization.

The first assumption we used in the proof above is that every trace prefix that a target program can produce can also be produced by a source program with the same interface. A bit more formally, we assume the existence of a back-translation function $\uparrow$ that given a finite prefix $m$ that can be produced by a whole target program $P_T$, returns a whole source program with the same interface $I_P$ as $P_T$ and which can produce the same prefix $m$ (i.e., $(m, I_P) \uparrow \Rightarrow \ast m$).

Assumption 1 (Back-translation).

$\exists t : \forall P : I_P. \forall m \text{ defined}. P \Rightarrow \ast m \Rightarrow (m, I_P) \uparrow : I_P \land (m, I_P) \uparrow \Rightarrow \ast m$

Back-translation only finite prefixes enables our proof technique, but at the same time limits it to only safety properties. While the other assumptions from this section can probably also be proved for infinite traces, there is no general way to define a finite program that produces an arbitrary infinite trace. We leave devising scalable back-translation proof techniques that go beyond safety properties to future work.

It is not always possible to take an arbitrary finite sequence of events and obtain a source program that realizes it. For example, in a language with a call stack and events \{\text{call}, \text{return}\}, there is no program that produces the single event trace \text{return}, since every return must be preceded by a call. Thus we only assume we can back-translate prefixes that are produced by the target semantics.

As further discussed in §5, similar back-translation techniques that start from finite execution prefixes have been used to prove fully abstract compilation [43, 66] and very recently $RSC$ [68] and even stronger variants [7]. Our back-translation, on the other hand, produces not just a source context, but a whole program. In the top-left corner of Figure 4, we assume that this resulting program, $(m, I_C \cup I_P)$, can be partitioned into a context $C_S$ that satisfies the interface $I_C$, and a program $P'$ that satisfies $I_P$.

Our second assumption is a form of forward compiler correctness for unsafe languages and a direct consequence of a forward simulation proof in the style of CompCert [58]. We assume separate compilation, in the style of a recent extension proposed by Kang et al. [47] and implemented in CompCert since version 2.7. Our assumption says that if a whole program composed of $P$ and $C$...
When Good Components Go Bad

Assumption 2 (Forward Compiler Correctness with Separate Compilation and Undefined Behavior).

∀C P. ∀m defined. (C ∪ P)→→m = (C_S ∪ P)^∗ \text{m}

The next assumption we make is recomposition, stating that if two programs, each composed of a partial program side \( P_L \) or \( P_R \), and a context side \( C_T \) or \( C_S \), sharing the same interfaces, produce the same finite trace prefix \( m \), then the complete program \( C_S ∪ P_L \) obtained by linking one side of each produce the same \( m \).

Assumption 3 (Recomposition).

∀P_Ip. ∀P′_Ip. ∀C_T: Ic. ∀C_S: Ic. ∀m.

(C_T ∪ P_L)→→m ∧ (C_S ∪ P′_L)→→m \Rightarrow (C_S ∪ P_L)→→m

In a previous version of this work [6], we used to split this theorem into two separate theorems, using a notion of partial semantics [7, 45] (a.k.a. "trace semantics" [43, 66]). First, decomposition stated that if a program obtained by linking two partial programs \( P_T \) and \( C_T \) produces a finite trace prefix \( m \) that does not end in an undefined behavior in the complete semantics, then each of the two partial programs (below we take \( P_T \), but the \( C_T \) case is symmetric) can produce \( m \) in the partial semantics. Then, the converse of decomposition, composition, stated that if two partial programs with matching interfaces produce the same prefix \( m \) with respect to the partial semantics, then they can be linked to produce the same \( m \) in the complete semantics. When taken together, composition and decomposition captured the fact that the partial semantics of the target language were adequate with respect to their complete counterparts. However, we noticed that proving those two assumptions independently was particularly complex. Indeed, while decomposition is a straightforward lemma provided the partial semantics are correctly setup, this is not the case for composition. Composing the two runs in the face of missing information was fraught with delicate and tedious reasoning, and did not fit the standard simulation mold we hoped it would. This is why we instead propose to obtain the new run in the complete semantics from the two runs in the complete semantics directly. We report how this new assumption is proved for our instance in §4.3.

In order to get back to the source language our proof uses a backwards compiler correctness assumption, again with separate compilation. As also explained in §3.2, we need to take into account that a trace prefix \( m \) in the target can be explained in the source either by an execution producing \( m \) or by one ending in an undefined behavior (i.e., producing \( t < m \)).

Assumption 4 (Backward Compiler Correctness with Separate Compilation and Undefined Behavior).

∀C P. \((C ∪ P_L)→→m \Rightarrow \exists t. (C ∪ P)→→t ∧ (m ≤ t ∨ t < m)\)

Finally, we assume that the context obtained by back-translation can’t be blamed for undefined behavior:

Assumption 5 (Blame). ∀C_S : Ic. ∀P′ : Ip. ∀m defined. ∀t.

If \((C_S ∪ P′)→→m \) and \((C_S ∪ P′)→→t \) and \( t < m \) then \( m ≤ t ∨ t < p m \).

We used Coq to prove the following theorem that puts together the assumptions from this subsection to show \( RSCC_{MD}^{DC} \):

Theorem 3.6. The assumptions above imply \( RSCC_{MD}^{DC} \).

4 SECURE COMPILATION CHAIN

We designed a simple proof-of-concept compilation chain to illustrate the \( RSCC \) property. The compilation chain is implemented in Coq. The source language is a simple, unsafe imperative language with buffers, procedures, and components (§4.1). It is first compiled to an intermediate compartmentalized machine featuring a compartmentalized, block-structured memory, a protected call stack, and a RISC-like instruction set augmented with an \texttt{ALLOC} instruction for dynamic storage allocation plus cross-component calls (§4.2). We can then choose one of two back-ends, which use different techniques to enforce the abstractions of the compartmentalized machine against realistic machine-code-level attackers, protecting the integrity of component memories and enforcing interfaces and cross-component call/return discipline.

When the compartmentalized machine encounters undefined behavior, both back ends instead produce an extended trace that respects high-level abstractions; however, they achieve this in very different ways. The \texttt{SFI back end} (§4.4) targets a bare-metal machine that has no protection mechanisms and implements an inline reference monitor purely in software, by instrumenting code to add address masking operations that force each component’s writes and (most) jumps to lie within its own memory. The \texttt{Micro-policies back end} (§4.5), on the other hand, relies on specialized hardware [27] to support a novel tag-based reference monitor for compartmentalization. These approaches have complementary advantages: SFI requires no specialized hardware, while micro-policies can be engineered to incur little overhead [27] and are a good target for formal verification [15] due to their simplicity. Together, these two back ends provide evidence that our \( RSCC \) security criterion is compatible with any sufficiently strong compartmentalization mechanism. It seems likely that other mechanisms such as capability machines [84] could also be used to implement the compartmentalized machine and achieve \( RSCC \).
When Good Components Go Bad

Abate et al.

Both back ends target variants of a simple RISC machine. In contrast to the abstract, block-based memory model used at higher levels of the compilation chain, the machine-level memory is a single infinite array addressed by mathematical integers. (Using unbounded integers is a simplification that we hope to remove in the future, e.g., by applying the ideas of Mullen et al. [61].) All compartments must share this flat address space, so—without proper protection—compromised components can access buffers out-of-bounds and read or overwrite the code and data of other components. Moreover, machine-level components can ignore the stack discipline and jump to arbitrary locations in memory.

We establish high confidence in the security of our compilation chain with a combination of proof and testing. For the compiler from the source language to the compartmentalized machine, we prove RSYC in Coq (§4.3) using the proof technique of §3.5. For the SFI back end, we use property-based testing with QuickChick [64] to systematically test $RSC^{DC}_{MDY}$.

### 4.1 Source Language

The source language from this section was designed with simplicity in mind. Its goal was to allow us to explore the foundational ideas of this paper and illustrate them in the simplest possible concrete setting, keeping our formal proofs tractable. The language is expression based (see Figure 5). A program is composed of an interface, a set of procedures, and a set of static buffers. Interfaces contain the names of the procedures that the component exports to and imports from other components. Each procedure body is a single expression whose result value is returned to the caller. Internal and external calls share the same global, protected call stack. Additional buffers can be allocated dynamically. As in C, memory is manually managed; out-of-bounds accesses lead to undefined behavior.

Values include integers, pointers, and an undefined value ⊤, which is obtained when reading from an uninitialized piece of memory or as the result of an erroneous pointer operation. As in CompCert and LLVM [57], our semantics propagates these ⊤ values and yields an undefined behavior if a ⊤ value is ever inspected. (The C standard, by contrast, specifies that a program is undefined as soon as an uninitialized read or bad pointer operation takes place.)

Memory Model The memory model for both source and compartmentalized machine is a slightly simplified version of the one used in CompCert [59]. Each component has an infinite memory composed of finite blocks, each an array of values. Accordingly, a pointer is a triple $(C, b, o)$, where $C$ is the identifier of the component that owns the block, $b$ is a unique block identifier, and $o$ is an offset inside the block. Arithmetic operations on pointers are limited to testing equality, testing ordering (of pointers into the same block), and changing offsets. Pointers cannot be cast to or from integers. Dereferencing an integer yields undefined behavior. For now, components are not allowed to exchange pointers; as a result, well-defined components cannot access each others’ memories at all. We hope to lift this restriction in the near future. This abstract memory model is shared by the compartmentalized machine and is mapped to a more realistic flat address space by the back ends.

Events Following CompCert, we use a labeled operational semantics whose events include all interactions of the program with the external world (e.g., system calls), plus events tracking control transfers from one component to another. Every call to an exported procedure produces a visible event $C$ Call $P(P')$, recording that component $C$ called procedure $P$ of component $P'$, passing argument $n$. Cross-component returns are handled similarly. All other computations, including calls and returns within the same component, result in silent steps in the operational semantics.

### 4.2 The Compartmentalized Machine

The compartmentalized intermediate machine aims to be as low-level as possible while still allowing us to target our two rather different back ends. It features a simple RISC-like instruction set (Figure 6) with two main abstractions: a block-based memory model and support for cross-component calls. The memory model leaves the back ends complete freedom in their layout of blocks. The machine has a small fixed number of registers, which are the only shared state between components. In the syntax, $l$ represent labels, which are resolved to pointers in the next compilation phase.

The machine uses two kinds of call stacks: a single protected global stack for cross-component calls plus a separate unprotected one for the internal calls of each component. Besides the usual Jal and Jump instructions, which are used to compile internal calls and returns, two special instructions, Call and Return, are used for cross-component calls. These are the only instructions that can manipulate the global call stack.

The operational semantics rules for Call and Return are presented in Figure 7. A state is composed of the current executing component $C$, the protected stack $\sigma$, the memory mem, the registers reg and the program counter pc. If the instruction fetched from the program counter is a Call to procedure $P$ of component $C'$, the semantics produces an event $\alpha$ recording the call, the callee, the procedure and its argument, which is stored in register $R_{C'DM}$. The protected stack $\sigma$ is updated with a new frame containing the next point in the code of the current component. Registers are mostly invalidated atCalls; $reg_{\sigma}$ has all registers set to ⊤ and only two registers are passed on: $R_{C'DM}$ contains the procedure’s argument and $R_{RA}$ contains the return address. So no data accidentally left by the caller in the other registers can be relied upon; instead the
We have proved that a compilation chain targeting the compartmentalized machine satisfies SCC, applying the technique from §3.5. As explained in §2, the responsibility for enforcing secure compilation can be divided among the different parts of the compilation chain. In this case, it is the target machine of §4.2 that enforces compartmentalization, while the compiler itself is simple, standard, and not particularly interesting (so omitted here).

For showing $RSCC_{\text{MD}}$ all the assumptions from §3.5 are proved using simulations. This proof was formalized in Coq, with the exception of compiler correctness (Assumptions 2 and 4), which is standard and essentially orthogonal to secure compilation; eventually, we hope to scale the source language up to a compartmentalized variant of C and reuse CompCert’s mechanized correctness proof. Our mechanized formalization is more detailed than previous paper proofs in the area [3, 5, 10–12, 31, 42, 43, 45, 63, 65–67]. Indeed, we are aware of only one fully mechanized proof about secure compilation: Devriese et al.’s [25] recent full abstraction result for the simply typed to the untyped $\lambda$-calculus in around 11KLOC of Coq.

Our Coq development comprises around 20KLOC, with proofs taking about 60%. Much of the code is devoted to generic models for components, traces, memory, and undefined behavior that we expect to be useful in proofs for more complex languages and compilers, such as CompCert. We discuss some of the most interesting aspects of the proof below.

**Back-translation function.** We proved Assumption 1 by defining a function that takes a finite trace prefix $m$ and a program interface $l$ and returns a whole source program that respects $l$ and produces $m$. Each generated component uses the local variable $\text{local}[0]$ to track how many events it has emitted. When a procedure is invoked, it increments $\text{local}[0]$ and produces the event in $m$ whose position is given by the counter’s value. For this back-translation to work correctly, $m$ is restricted to look like a trace emitted by a real compiled program with an $l$ interface—in particular, every return in the trace must match a previous call.

This back-translation is illustrated in Figure 8 on a trace of four events. The generated program starts running $\text{mainC}$, mainP, with all counters set to 0, so after testing the value of $\text{mainC}.\text{local}[0]$, the program runs the first branch of mainP:

\[
\text{local}[0]++;\quad C\ p(0);\quad \text{mainC}.\text{mainP}(0);
\]

After bumping $\text{local}[0]$, mainP emits its first event in the trace: the call $C\ p(0)$. When that procedure starts running, C’s counter is still set to 0, so it executes the first branch of procedure p:

\[
\text{local}[0]++;\quad \text{return}\ 1;
\]

The return is C’s first event in the trace, and the second of the program. When mainP regains control, it calls itself recursively to emit the other events in the trace (we can use tail recursion to iterate in the standard way, since internal calls are silent events). The program continues executing in this fashion until it has emitted all events in the trace, at which point it terminates execution.

**Theorem 4.1 (Back-translation).** The back-translation function $\uparrow$ illustrated above satisfies Assumption 1.

**Recomposition.** We prove Assumption 3 by proving a slightly more general lemma, obtained by considering arbitrary target programs $P_T$ and $P_T'$ instead of the compiled source programs $P_1$ and $P_1'$, and arbitrary target context $C_T'$ instead of the compiled source context $C_1$. To do so, we use a three-way simulation, stated in terms of the underlying small-step semantics.

First, we state a mergeability relation $\sim$ between pairs of complete states $s$ and $s'$, and prove that it is preserved during a parallel execution guided by the trace. In particular, this relation ensures that the shapes of the two stacks of the two executions are related, in that they must capture the same interactions (calls and returns) between program and context, despite the possibly different implementations. Furthermore, we define a merge function, implicitly parameterized by the program and context interfaces, that defined how to merge two mergeable states: if $s$ is a state of $(C_T \cup P_T)$, and $s'$ is a state of $(C_T' \cup P_T')$, then merge $s$ and $s'$ is a state of $(C_T' \cup P_T)$. To merge two states $s$ and $s'$, one has to (1) retain the PC and registers of the execution that is currently in control of the run, i.e., s if the program side is executing (the PC points to code belonging to a component inside $P_T$), and $s'$ otherwise; (2) keep only the memory belonging to the program $P_T$ in $s$, and to the context $C_T'$ in $s'$, and take their union; (3) recombine the stacks, interleaving the frames from $C_T'$ and $P_T$ in order to preserve the (compatible) structure of the original stacks.

Second, we prove the three-way simulation proper, showing that if $s_1 \xrightarrow{m} s_2$ and $s'_1 \xrightarrow{m} s'_2$ where $s_1 \sim s'_1$, then $s_2 \sim s'_2$ and merge $s_1$ and $s'_1$ merge $s_2$ and $s'_2$, where $\xrightarrow{m}$ is the reflexive transitive closure of the step relation, producing the events in $m$. At each step, the simulation is guided by either both programs or both contexts. A step in one of the runs is matched by any number of silent steps in the other. The two runs synchronize on observable events, corresponding to comparable event-producing steps, possibly padded with silent steps before and after. Control alternates between programs and contexts at cross-component events. These new definitions are more natural, completely symmetric, and easier to work with than the previous ones about partial semantics [6].
Moreover, using the complete semantics allows us to reuse theorems already proved for these semantics. As a result, the resulting proofs are fully mechanized, several times smaller than the original proofs, and several times faster to check.

While we are able to prove a generalized lemma for arbitrary target programs and contexts in our case, we suspect that it is not possible in the general setting. Indeed, if the compiler plays a bigger role for enforcing security than in our setting, one likely needs to use the fact that the programs are compiled programs. For example, the compiler might be inserting protections in the compiled code at boundaries between components that ensure cross-components calls are well-behaved. In such a case, it is hard to see how a recomposition proof could go without considering the compiler.

**Theorem 4.2 (Recomposition).** Assumption 3 is satisfied.

**Blame.** We prove Assumption 5 by noting that the behavior of the context $C_S$ can only depend on its own state and on the events emitted by the program. A bit more formally, suppose that the states $c_{s1}$ and $c_{s2}$ have the same context state, which, borrowing the partialization notation from above, we write as $\text{par}(c_{s1}, I_p) = \text{par}(c_{s2}, I_p)$. Then:

- If $c_{s1} \xrightarrow{a} c_{s1}'$, $c_{s2} \xrightarrow{a} c_{s2}'$, and $C_S$ has control in $c_{s1}$ and $c_{s2}$, then $c_1 = a_2$ and $\text{par}(c_{s1}', I_p) = \text{par}(c_{s2}', I_p)$.
- If $c_{s1} \xrightarrow{a} c_{s1}'$ and the program has control in $c_{s1}$ and $c_{s2}$, then $\text{par}(c_{s1}', I_p) = \text{par}(c_{s2}', I_p)$.
- If $c_{s1} \xrightarrow{r} c_{s1}'$, the program has control in $c_{s1}$ and $c_{s2}$, and $r \neq a$, then there exists $c_{s2}'$ such that $c_{s2} \xrightarrow{a} c_{s2}'$ and $\text{par}(c_{s1}', I_p) = \text{par}(c_{s2}', I_p)$.

By repeatedly applying these properties, we can analyze the behavior of two parallel executions $(C_S \cup P)^{\rightarrow m} \rightarrow t$ and $(C_S \cup P)^{\rightarrow t}$, with $t < m$. By unfolding the definition of $t$, we get that $3m' \leq m$, where $m' \cdot \text{Undef}(\_)$, $m' \cdot \text{Undef}(\_)$, and $m' \cdot \text{Undef}(\_)$, where $m'$ and $m$, and $m' \cdot \text{Undef}(\_)$, $m' \cdot \text{Undef}(\_)$, and $m' \cdot \text{Undef}(\_)$, we have $m \leq t$, and we are done. Otherwise, the execution of $C_S \cup P$ ended earlier because of undefined behavior. After producing prefix $m'$, $C_S \cup P'$ and $C_S \cup P$ will end up in matching states $c_{s1}$ and $c_{s2}$. Aiming for a contradiction, suppose that undefined behavior was caused by $C_S$. By the last property above, we could find a matching execution step for $C_S \cup P$ that produces the first event in $m$ that is outside of $m'$; therefore, $C_S \cup P$ cannot be stuck at $c_{s2}$. Hence $t < m$.

**Theorem 4.3 (Blame).** Assumption 5 is satisfied.

Since we have completely proved assumptions 1, 3, and 5 of Theorem 3.6 in Coq, and assumed 2 and 4, we can instantiate the generic proof technique of RSCC from §3.5 to obtain the final result.

**Theorem 4.4 (RSCC).** The compilation chain described so far in this section satisfies RSCC.

### 4.4 Software Fault Isolation Back End

The SFI back end uses a special memory layout and code instrumentation sequences to realize the desired isolation of components in the produced program. The target of the SFI back end is a bare-metal RISC processor with the same instructions as the compartmentalization machine minus Call, Return, and Alloc. The register file contains all the registers from the previous level, plus seven additional registers reserved for the SFI instrumentation.

The SFI back end maintains the following invariants: (1) a component may not write outside its own data memory; (2) a component may transfer control outside its own code memory only to entry points allowed by the interfaces or to the return address on top of the global stack; and (3) the global stack remains well formed.

Figure 9 shows the memory layout of an application with three components. The entire address space is divided in contiguous regions of equal size, which we will call slots. Each slot is assigned to a component or reserved for the use of the protection machinery. Data and code are kept in disjoint memory regions and memory writes are permitted only in data regions.

An example of a logical split of a physical address is shown in Figure 10. A logical address is a triple: offset in a slot, component identifier, and slot identifier unique per component. The slot size, as well as the maximum number of the components are constant for an application, and in Figures 9 and 10 we have 3 components and slots of size $2^{12}$ bits.

The SFI back end protects memory regions with instrumentation in the style of Wahbe et al. [82], but adapted to our component model. Each memory update is preceded by two instructions that set the component identifier to the current one, to prevent accidental or malicious writes in a different component. The instrumentation of the Jump instruction is similar. The last four bits of the offset are always zeroed and all valid targets are sixteen-word-aligned by...
our back end [60]. This mechanism, along with careful layout of instructions, ensure that the execution of instrumentation sequences always starts from the first instruction and continues until the end.

The global stack is implemented as a shadow stack [76] in memory accessible only from the SFI instrumentation sequences. Alignment of code [60] prevents corruption of the cross-component stack with prepared addresses and ROP attacks, since it is impossible to bypass the instructions in the instrumentation sequence that store the correct address in the appropriate register.

The Call instruction of the compartmentalized machine is translated to a Jal (jump and link) followed by a sequence of instructions that push the return address on the stack and then restore the values of the reserved registers for the callee component. To protect from malicious pushes that could try to use a forged address, this sequence starts with a Halt at an aligned address. Any indirect jump from the current component, will be aligned and will execute the Halt, instead of corrupting the cross-component stack. A call from a different component, will execute a direct jump, which is not subject to masking operations and can thus target an unaligned address (we check statically that it is a valid entry point). This Halt and the instructions that push on the stack are contained in the sixteen-unit block.

The Return instruction is translated to an aligned sequence: pop from the protected stack and jump to the retrieved address. This sequence also fits entirely in a sixteen-unit block. The protection of the addresses on the stack itself is realized by the instrumentation of all the Store and Jump instructions in the program.

We used the QuickChick property-based testing tool [64] for Coq to test the three compartmentalization invariants described at the beginning of the subsection. For each invariant, we implemented a test that executes the following steps: (i) randomly generates a valid compartmentalized machine program; (ii) compiles it; (iii) executes the resulting target code in a simulator and records a property-specific trace; and (iv) analyzes the trace to verify if the property has been violated. We also manually injected faults in the compiler by mutating the instrumentation sequences of the generated output and made sure that the tests can detect these injected errors.

More importantly, we also tested two variants of the $\text{RSC}^{\text{DC}}_{\text{MD}}$ property, which consider different parts of a whole program as the adversarial context. Due to the strict memory layout and the requirement that all components are instrumented, the SFI back end cannot link with arbitrary target code, and has instead to compile a whole compartmentalized machine program. In a first test, we (1) generate a whole compartmentalized machine program $P$; (2) compile $P$; (3) run a target interpreter to obtain trace $t_T$; (4) if the trace is empty, discard the test; (5) for each component $C_P$ in the trace $t_T$ (5-1) use back-translation to replace, in the program $P$, the component $C_P$ with a component $C_T$ without undefined behavior (5-2) run the new program on the compartmentalized machine and obtain a trace $t_T$ (5-3) if the condition $t_T \leq t_T$ or $t_T < p | C_P \cup C_T$ is satisfied then the test passes, otherwise it fails. Instead of performing step (5), our second test replaces in one go all the components exhibiting undefined behavior, obtaining a compartmentalized machine program that should not have any undefined behavior.

### 4.5 Tag-based Reference Monitor

Our second back end is a novel application of a programmable tagged architecture that allows reference monitors, called micro-policies, to be defined in software but accelerated by hardware for performance [15, 27]. On a micro-policy machine, each word in memory or registers carries a metadata tag large enough to hold a pointer to an arbitrary data structure in memory. As each instruction is dispatched by the processor, the opcode of the instruction as well as the tags on the instruction, its argument registers or memory cells, and the program counter are all passed to a software monitor that decides whether to allow the instruction and, if so, produces tags for the results. The positive decisions of this monitor are cached in hardware, so that, if another instruction is executed in the near future with similarly tagged arguments, the hardware can allow the request immediately, bypassing the software monitor.

This enforcement mechanism has been shown flexible enough to implement a broad range of tag-based reference monitors, and for many of them it has a relatively modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy (typically under 60%) and chip area (10%) [27]. Moreover, the mechanism is simple enough so that the security of the reference monitors can be verified formally [13–16]. The micro-policy machine targeted by our compartmentalizing back end builds on a "symbolic machine" that Azevedo de Amorim et al. used to prove the correctness and security of several micro-policies in Coq [13, 15, 16].

The code generation and static linking parts of the micro-policy back end are much simpler than for the SFI one. The Call and Return instructions are mapped to Jal and Jump. The Alloc instruction is mapped to a monitor service that tags the allocated memory according to the calling component.

A more interesting aspect of this back end is the way memory must be tagged by the (static) loader based on metadata from previous compilation stages. Memory tags are tuples of the form $t_m := (t_v, c, cs)$. The tag $t_v$ is for the payload value. The component identifier $c$, which we call a color, establishes the component that owns the memory location. Our monitor forbids any attempt to write to memory if the color of the current instruction is different from the color of the target location. The set of colors $cs$ identifies all the components that are allowed to call to this location and is by default empty. The value tags used by our monitor distinguish cross-component return addresses from all other words in the system: $t_v := \text{Ret}(n) \upharpoonright \bot$. To enforce the cross-component stack discipline

![Figure 9: Memory layout of three user components](image)

![Figure 10: Address Example](image)
return addresses are treated as linear return capabilities, i.e., unique capabilities that cannot be duplicated [50] and that can only be used to return once. This is achieved by giving return addresses tags of the form $Ret(n)$, where the natural number $n$ represents the stack level to which this capability can return. We keep track of the current stack level using the tag of the program counter: $tpc := Level(n)$. Calls increment the counter $n$, while returns decrement it. A global invariant is that when the stack is at $Level(n)$ there is at most one capability $Ret(m)$ for any level $m$ from 0 up to $n−1$.

Our tag-based reference monitor for compartmentalization is simple; the complete definition is given in Figure 11. For Move, Store, and Load the monitor copies the tags together with the values, but for return addresses the linear capability tag $Ret(n)$ is moved from the source to the destination. Loads from other components are allowed but prevented from stealing return capabilities. Store operations are only allowed if the color of the changed location matches the one of the currently executing instruction. Branch is restricted to the current component. Jump to a different component is only allowed if the color of the current component is included in the allowed entry points; in this case and if we are at some $Level(n)$ the machine puts the return address in register RA and the monitor gives it tag $Ret(n)$ and it increments the pc tag to $Level(n+1)$. Jump is allowed either to the current component or using a $Ret(n)$ capability, but only if we are at $Level(n+1)$; in this case the pc tag is incremented to $Level(n)$ and the $Ret(n)$ capability is destroyed. Instruction fetches are also checked to ensure that one cannot switch components by continuing to execute past the end of a code region. To make these checks as well as the ones for Jump convenient we use the next instruction tag NI directly; in reality one can encode these checks even without NI by using the program counter and current instruction tags [15]. The bigger change compared to the micro-policy mechanism of Azevedo de Amorim [15] is our overwriting of input tags in order to invalidate linear capabilities in the rules for Move, Load, and Store. For cases in which supporting this in hardware is not feasible we have also devised a compartmentalization micro-policy that does not rely on linear return capabilities but on linear entry points.

A variant of the compartmentalization micro-policy above was first studied by Juglaret et al. [46], in an unpublished technical report. Azevedo de Amorim et al. [15] also devised a micro-policy for compartmentalization, based on a rather different component model. The biggest distinction is that our micro-policy enforces the stack discipline on cross-component calls and returns. We first encountered linear capabilities in the context of the SAFE machine [50], although the idea seemed folklore. Probably even closer related to our linear return capabilities are, however, linear continuations [29, 88].

5 RELATED WORK

Fully Abstract Compilation, originally introduced in seminal work by Abadi [2], is phrased in terms of protecting two partial program variants written in a safe source language, when these are compiled and linked with a malicious target-level context that tries to distinguish the two variants. This original attacker model differs substantially from the one we consider in this paper, which protects the trace properties of multiple mutually-distrustful components written in an unsafe source language.

In this line of research, Abadi [2] and later Kennedy [48] identified failures of full abstraction in the Java and C# compilers. Abadi et al. [3] proved full abstraction of secure channel implementations using cryptography. Ahmed et al. [10–12, 63] proved the full abstraction of type-preserving compiler passes for functional languages. Abadi and Plotkin [5] and Jagadeesan et al. [42] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet et al. [31] devised a fully abstract compiler from a subset of ML to JavaScript. More recently, Patrignani et al. [65] studied fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting a hardware enclave mechanism similar to Intel SGX [40].

Modular, Fully Abstract Compilation. Patrignani et al. [67] subsequently proposed a modular extension of their compilation scheme to protecting multiple components from each other. The attacker model they consider is again different from ours: they focus on separate compilation of safe languages and aim to protect linked target-level components that are observationally equivalent to compiled components. This could be useful, for example, when hand-optimizing assembly produced by a secure compiler. In another thread of work, Devriese et al. [25] proved modular full abstraction by approximate back-translation for a compiler from simply typed to untyped $\lambda$-calculus. This work also introduces a complete Coq formalization for the original (non-modular) full abstraction proof of Devriese et al. [23].

Beyond Good and Evil. The closest related work is that of Juglaret et al. [45], who also aim at protecting mutually distrustful components written in an unsafe language. They adapt fully abstract compilation to components, but observe that defining observational equivalence for programs with undefined behavior is highly problematic. For instance, the partial program “int buf[5]; return buf[42]” equivalent to “int buf[5]; return buf[43]”? Both encounter undefined behavior by accessing a buffer out of bounds, so at the source level they cannot be distinguished. However, in an unsafe language, the compiled versions of these programs will likely read (out of bounds) different values and behave differently. Juglaret et al. avoid this problem by imposing a strong limitation: a set of components is protected only if it cannot encounter undefined behavior in any context. This amounts to a static model of compromise: all components that can possibly be compromised during execution have to be treated as compromised from the start. Our aim here is to show that, by moving away from full abstraction and by restricting the temporal scope of undefined behavior, we can support a more realistic dynamic compromise model. As discussed below, moving away from full abstraction also makes our secure compilation criterion easier to achieve in practice and to prove at scale.

Robustly Safe Compilation. Our criterion builds on Robustly Safe Compilation (RSC), recently proposed by Abate et al. [7], who study several secure compilation criteria that are similar to fully abstract compilation, but that are phrased in terms of preserving hyper-properties [22] (rather than observational equivalence) against an adversarial context. In particular, RSC is equivalent to preservation of robust safety, which has been previously employed for the
When Good Components Go Bad

Abate et al.

Nop : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots) \} → \{ PC \rightarrow t_{pc} \}
Const : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots) \} → \{ PC \rightarrow t_{pc}, D = \bot \}

BinOp : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), S_{i} = \ldots, S_{j} = \ldots \} → \{ PC \rightarrow t_{pc}, D = \bot \}

Mov : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), S = t_{v} \} → \{ PC \rightarrow t_{pc}, S = \bot, D = t_{v} \}
Load : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), M = (t_{v}, c\ldots) \} → \{ PC \rightarrow t_{pc}, M = (\bot, c\ldots), D = t_{v} \}
Load : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), M = (t_{v}, c\ldots) \} → \{ PC \rightarrow t_{pc}, M = (t_{v}, c\ldots), D = \bot \}

Store : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), M = (\ldots, c\ldots), S = t_{v} \} → \{ PC \rightarrow t_{pc}, M = (t_{v}, c\ldots), S = \bot \}

Bnz : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots) \} → \{ PC \rightarrow t_{pc} \}
Jal : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), P = \ldots \} → \{ PC \rightarrow t_{pc}, RA = \bot \}
Jal : \{ PC \rightarrow \text{Level}(n), C I(\ldots, c\ldots), N I(\ldots, c\ldots), P = \ldots \} → \{ PC \rightarrow \text{Level}(n + 1), RA = \text{Ret}(n) \}

Jump : \{ PC \rightarrow t_{pc}, C I(\ldots, c\ldots), N I(\ldots, c\ldots), P = \ldots \} → \{ PC \rightarrow t_{pc}, RA = \bot \}
Jump : \{ PC \rightarrow \text{Level}(n + 1), C I(\ldots, c\ldots), N I(\ldots, c\ldots), P = \ldots \} → \{ PC \rightarrow \text{Level}(n), P = \bot \}

Figure 11: Compartmentalization micro-policy rules

model checking of open systems [53], the analysis of security protocols [33], and compositional verification [75].

Though RSC is a bit less extensional than fully abstract compilation (since it is stated in terms of execution traces), it is easier to achieve. In particular, because it focuses on safety instead of confidentiality, the code and data of the protected program do not have to be hidden, allowing for more efficient enforcement, e.g., there is no need for fixed padding to hide component sizes, no cleaning of registers when passing control to the context (unless they store capabilities), and no indirection via integer handlers to hide pointers; cross-component reads can be allowed and can be used for passing large data. We believe that in the future we can obtain a more practical notion of data (but not code) confidentiality by adopting Garg et al.’s robust hypersafety preservation criterion [7].

While RSC serves as a solid base for our work, the challenges of protecting unsafe components from each other are unique to our setting, since, like full abstraction, RSC is about protecting a partial program written in a safe source language against low-level contexts. Our contribution is extending RSC to reason about the dynamic compromise of components with undefined behavior, taking advantage of the execution traces to detect the compromise of components and to rewire the execution along the same trace.

Proof Techniques. Abate et al. [7] observe that, to prove RSC, it suffices to back-translate finite execution prefixes, and recently they propose such a proof for a stronger criterion where multiple such executions are involved. In recent concurrent work, Patriziani and Garg [68] also construct such a proof for RSC. The main advantages of our RSC_{DC}^{MD} Proof are that (1) it applies to unsafe languages with undefined behavior and (2) it directly reuses a compiler correctness result à la CompCert. For safe source languages or when proof reuse is not needed our proof could be further simplified.

Even as it stands though, our proof technique is simple and scalable compared to previous full abstraction proofs. While many proof techniques have been previously investigated [3, 5, 11, 12, 25, 31, 42, 63], fully abstract compilation proofs are notoriously difficult, even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled [24]. The proofs of Juglaret et al. [45] are no exception: their compiler is similar to the one in §4, their full abstraction-based proof is significantly more complex than our RSC_{DC}^{MD} proof. Both proofs give semantics to partial programs in terms of traces, as was proposed by Jeffrey and Ratheke [43] and adapted to low-level target languages by Patriziani and Clarke [66]. However, in our setting the partial semantics is given a one line generic definition and is related to the complete one by two simulation proofs, which is simpler than proving a “trace semantics” fully abstract.

Verifying Low-Level Componentalization. Recent successes in formal verification have focused on showing correctness of low-level componentalization mechanisms based on software fault isolation [60, 89] or tagged hardware [15]. That work only considers the correctness of low-level mechanisms in isolation, not how a secure compilation chain makes use of these mechanisms to provide security reasoning principles for code written in a higher-level programming language with components. However, more work in this direction seems underway, with Wilke et al. [85] working on a variant of CompCert with SFI, based on previous work by Kroll et al. [52]; we believe RSC or RSC_{DC} could provide good top-level theorems for such an SFI compiler. In most work on verified componentalization [15, 60, 89], communication between low-level compartments is done by jumping to a specified set of entry points; the model considered here is more structured and enforces the correct return discipline. Skorstengaard et al. have also recently investigated a secure stack-based calling convention for a simple capability machine [74]; they plan to simplify their calling convention using a notion of linear return capability [73] that seems similar to the one used in our micro-policy from §4.5.

Attacker Models for Dynamic Compromise. While our model of dynamic compromise is specific to secure compilation of unsafe languages, related notions of compromise have been studied in the setting of cryptographic protocols, where, for instance, a participant’s secret keys could inadvertently be leaked to a malicious adversary, who could then use them to impersonate the victim [17, 18, 30, 34]. This model is also similar to Byzantine behavior in distributed systems [20, 55], in which the “Byzantine failure” of a node can cause it to start behaving in an arbitrary way, including generating arbitrary data, sending conflicting information to different parts of the system, and pretending to be a correct node.
6 CONCLUSION AND FUTURE WORK

We introduced RSCC, a new formal criterion for secure compilation providing strong security guarantees despite the dynamic compromise of components with undefined behavior. This criterion gives a precise meaning to informal terms like dynamic compromise and mutual distrust used by proponents of compartmentalization, and it offers a solid foundation for reasoning about security of practical compartmentalized applications and secure compiler chains.

Formally Secure Compartmentalization for C. Looking ahead, we hope to apply RSCC to the C language by developing a provably secure compartmentalizing compiler chain based on the CompCert compiler. Scaling up to the whole of C will certainly entail further challenges such as defining a variant of C with components and efficiently enforcing compartmentalization all the way down. We believe these can be overcome by building on the solid basis built by this paper: the RSCC formal security criterion, the scalable proof technique, and the proof-of-concept secure compilation chain.

A very interesting extension is sharing memory between components. Since we already allow arbitrary reads at the lowest level, it seems appealing to also allow external reads from some of the components’ memory in the source. The simplest would be to allow certain static buffers to be shared with all other components, or only with some if we also extend the interfaces. For this extension, the back-translation would need to set the shared static buffers to the right values every time a back-translated component gives up control; for this back-translation needs to look at the read events forward in the back-translated trace prefix. More ambitious would be to allow pointers to dynamically allocated memory to be passed to other components, as a form of read capabilities. This would make pointers appear in the traces and one would need to accommodate the fact that these pointers will vary at the different levels in our compilation chain. Moreover, each component produced by the back-translation would need to record all the read capabilities it receives for later use. Finally, to safety allow write capabilities one could combine compartmentalization with memory safety [15, 16].

Verifying Compartmentalized Applications. It would also be interesting to build verification tools based on the source reasoning principles provided by RSCC and to use these tools to analyze the security of practical compartmentalized applications. Effective verification on top of RSCC will, however, require good ways for reasoning about the exponential number of dynamic compromise scenarios. One idea is to do our source reasoning with respect to a variant of our partial semantics, which would use nondeterminism to capture the compromise of components and their possible successive actions. Correctly designing such a partial semantics for a complex language is, however, challenging. Fortunately, our RSCC criterion provides a more basic, low-TCB definition against which to validate any fancier reasoning tools, like partial semantics, program logics [44], logical relations [26], etc.

Dynamic Component Creation. Another interesting extension is supporting dynamic component creation. This would make crucial use of our dynamic compromise model, since components would no longer be statically known, and thus static compromise would not apply, unless one severely restricts component creation to only a special initialization phase [62, 72]. We hope that our RSCC definition can be adapted to rewind execution to the point at which the compromised component was created, replace the component’s code with the result of our back-translation, and then re-execute. This extension could allow us to also explore moving from our current “code-based” compartmentalization model to a “data-based” one [35], e.g., one compartment per incoming network connection.

Dynamic Privilege Notions. Our proof-of-concept compilation chain used a very simple notion of interface to statically restrict the privileges of components. This could, however, be extended with dynamic notions of privilege such as capabilities and history-based access control [1]. In one of its simplest forms, allowing pointers to be passed between components and then used to write data, as discussed above, would already constitute a dynamic notion of privilege that is not captured by the static interfaces, but nevertheless needs to be enforced to achieve RSCC, in this case probably using some form of memory safety.

Preserving Confidentiality and Hypersafety. It would be interesting to extend our security criterion and enforcement mechanisms from robustly preserving safety to confidentiality and hypersafety [7, 22]. For this one needs to control the flow of information at the target level, e.g., by restricting direct reads and read capabilities, cleaning registers, etc. This becomes very challenging though, in a realistic attacker model in which low-level contexts can observe time.

Acknowledgments We thank the anonymous reviewers for their valuable feedback. Yannis Jugleret first studied with us the compartmentalization micro-policy from §4.5 and described it in an unpublished technical report [46]. Jérémy Thibault helped us correctly extract the micro-policies back end to OCaml. We are also grateful to Daniel Ahman, Eric Taner, Steven Schäfer, and William Bowman, for insightful discussions and thoughtful feedback on earlier drafts. This work is in part supported by ERC Starting Grant SECOMP (715753), by NSF award Micro-Policies (1513854), and by DARPA grant SSITH/HOPE (FA8650-15-C-7558).

A CLASS OF SAFETY PROPERTIES PRESERVED BY RSCDC

Since RSC corresponds exactly to preserving robust safety properties [7], one might wonder what properties RSCDC preserves. In fact, RSCDC corresponds exactly to preserving the following class ZP against an adversarial context.

Definition A.1. ZP ≜ Safety ∩ Closed−<ϕ, where

\[
\text{Safety} ≜ \{ \pi \mid \forall t \pi \cdot \exists m \leq t, \forall t' \geq m, t' \not\in \pi \}
\]

\[
\text{Closed}_{<\phi} ≜ \{ \pi \mid \forall t \in \pi, \forall t', t < pt' \Rightarrow t' \in \pi \}
\]

\[
= \{ \pi \mid \forall t' \not\in \pi, \forall t, t < pt' \Rightarrow t' \in \pi \}
\]

The class of properties ZP is defined as the intersection of Safety and the class Closed−<ϕ of properties closed under extension of traces with undefined behavior in P [58]. If a property π is in Closed−<ϕ and it allows a trace t that ends with an undefined behavior in P—i.e., ∃m. t = m · \text{Undef}(P)—then π should also allow any extension of the trace m—i.e., any trace t’ that has m as a prefix. The intuition is simple: the compilation chain is free to implement a trace with undefined behavior in P as an arbitrary trace extension, so if the property accepts traces with undefined behavior it should also accept their extensions. Conversely, if a property π in Closed−<ϕ rejects a trace t’, then for any prefix m of t’ the property π should also reject the trace m · \text{Undef}(P).
For a negative example that is not in $\mathit{Closed}_{\leq P}$, consider the following formalization of the property $S_1$ from §2, requiring all writes in the trace to be preceded by a corresponding read:

$$S_1 = \{ t \mid \forall m \; \delta \cdot x \cdot m \cdot \mathit{E.write}(<d>x) \leq t \Rightarrow \exists m'. m' \cdot \mathit{E.read}. \mathit{Ret}(x) \leq m \}$$

While property $S_1$ is Safety if it is not $\mathit{Closed}_{\leq P}$. Consider the trace $t' = [C_0.\mathit{main}; \mathit{E.write}(<d>x)] \not\in S_1$ that does a write without a read and thus violates $S_1$. For $S_1$ to be $\mathit{Closed}_{\leq P}$, it would have to reject not only $t'$, but also $[C_0.\mathit{main}; \mathit{Undef}(P)]$ and $\mathit{Undef}(P)$, which it does not. One can, however, define a stronger variant of $S_1$ that is in $Z_P$:

$$S_1^{Z_P} = \{ t \mid \forall m \; \delta \cdot x \cdot m \cdot \mathit{E.write}(<d>x) \leq t \lor m \cdot \mathit{Undef}(P) \leq t \Rightarrow \exists m'. m' \cdot \mathit{E.read}. \mathit{Ret}(x) \leq m \}$$

The property $S_1^{Z_P}$ requires any write or undefined behavior in $P$ to be preceded by a corresponding read. While this property is quite restrictive, it does hold (vacuously) for the strengthened system in Figure 2 when taking $P = \{C_0\}$ and $C = \{C_0, C_2\}$, since we assumed that $C_3$ has no undefined behavior.

Using $Z_P$, we proved an equivalent RSC-DC characterization:

**Theorem A.2.**

$$\mathit{RSC-DC} \iff \left( \forall P \pi \in Z_P. \left( \forall C_0 \; t \cdot C_0[P] \Rightarrow t \in \pi \right) \Rightarrow \left( \forall C_0 \; t \cdot C_1[P_t] \Rightarrow t \in \pi \right) \right)$$

This theorem shows that $\mathit{RSC-DC}$ is equivalent to the preservation of all properties in $Z_P$ for all $P$. One might still wonder how one obtains such robust safety properties in the source language, given that the execution traces can be influenced not only by the partial program but also by the adversarial context. In cases in which the trace records enough information so that one can determine the originator of each event, robust safety properties can explicitly talk only about the events of the program, not the ones of the context. Moreover, once we add interfaces in $\mathit{RSC}$ in the source using an $\mathit{Undef}$ call, we can obtain such robust safety properties in the source language, given that

$$\pi^{Z_P} \triangleq \pi \cap \{ t \mid \forall t' \cdot t < p t' \Rightarrow t' \in \pi \}$$

We can also easily answer the dual question asking what is left of an arbitrary safety property established in the source when looking at the target of an $\mathit{RSC-DC}$ compiler:

$$\pi^{Z_P} = \{ \{ t \mid \exists \pi \in \pi. t < p t' \lor t' \leq t \}$$

### REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. NDSS. The Internet Society, 2003.

[2] M. Abadi. Protection in programming-language translations. Secure Internet Programming, 1999.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. Information and Computation, 174(1):37–83, 2002.

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

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

[6] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fuchini, C. Hrĳcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. CCS 2018.

[7] C. Abate, R. Blanco, D. Garg, C. Hrĳcu, M. Patrignani, and J. Thibault. Journey beyond full abstractions: Exploring robust property preservation for secure compilation. CSF, 2019.

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

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

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

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

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

[13] A. Azevedo de Amorim. A methodology for micro-policies. PhD thesis, University of Pennsylvania, 2017.

[14] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hrĳcu, D. Richarde, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. POPL. 2014.

[15] A. Azevedo de Amorim, M. Dénex, N. Giannarakis, C. Hrĳcu, B. C. Pierce, A. Spector-Zabursky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. Oakland S&P. 2015.

[16] A. Azevedo de Amorim, C. Hrĳcu, and B. C. Pierce. The meaning of memory safety. POPL. 2018.

[17] M. Backes, M. P. Grochulla, C. Hrĳcu, and M. Maffei. Achieving security despite compromise using zero-knowledge. CSF. 2009.

[18] D. A. Basin and C. Cremers. Know your enemy: Compromising adversaries in protocol analysis. TISSEC, 17(2):7:1–7:31, 2014.

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

[20] M. Castro and B. Liakov. Practical byzantine fault tolerance and proactive recovery. TOCS. 2004:399–438, 2002.

[21] D. Chinnall, 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. ASILOM. 2015.

[22] M. R. Clarkson and F. B. Schneider. Hyperproperties. JCS, 18(6):1157–1210, 2010.

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

[24] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. PACMP, 2:POPL 38:1–38:23, 2018.

[25] D. Devriese, M. Patrignani, F. Piessens, and S. Krechdel. Modular, fully-abstract compilation by approximate back-translation. LMCS, 13(4), 2017.

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

[27] U. Dhawan, C. Hrĳcu, R. Rubin, N. Vasiliakos, S. Chiricescu, J. M. Smith, T. F. Kennedy, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. ASILOM. 2015.

[28] G. J. Duck and R. H. C. Yap. EffectivelySan: Type and memory error detection using dynamically typed C/C++. PLDI. 2018.

[29] A. Filinski. Linear continuations. POPL. 1992.

[30] C. Fournet, A. D. Gordon, and S. Maffei. A type discipline for authorization policies. ACM Trans. Program. Lang. Syst., 29(5):25, 2007.

[31] C. Fournet, N. Swamy, J. Chen, P.-E. Daggad, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. POPL. 2013.

[32] A. Gollamudi and C. Fournet. Building secure SGX enclaves using $F$, C/C++ and X64 and 2nd Workshop on Principles of Secure Compilation (PrSec), 2018.

[33] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. JCS, 123(4):435–483, 2004.

[34] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. CONCUR. 2005.

[35] K. Gudka, R. N. M. Watson, J. Anderson, D. Chinnall, B. Davis, L. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. CCS. 2013.

[36] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Golman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. PLDI. 2015.
When Good Components Go Bad Abate

G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, X. Leroy. Formal verification of a realistic compiler.

J. Regehr. A guide to undefined behavior in C and C++, part 3. Embedded in T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of.

T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In Secure Compilation (PriSC), 2018.

N. van Ginkel, R. Strackx, J. T. Muehlberg, and F. Piessens. Towards safe enclaves. HotSpot, 2016.

T. Van Strydonck, D. Devriese, and F. Piessens. Linear capabilities for modular fully-abstract compilation of verified code. 2nd Workshop on Principles of Secure Compilation (PriSC), 2018.

N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. NDSS 2018.

R. Wahbe, S. Luccio, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. SOSP, 1993.

X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. SOSP, 2013.

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

P. Wilke, F. Besson, S. Blazy, and A. Dang. CompCert for software fault isolation. Secure Compilation Meeting (SCM), 2017.

T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. IEEE S&P, 1993.

B. Yee, D. Serh, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaki, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. CACM, 53(1):91–99, 2010.

S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. Higher Order and Symbolic Computation, 15:2002, 2002.

L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. EMSOFT, 2011.