When Good Components Go Bad

Formally Secure Compilation Despite Dynamic Compromise

Carmine Abate
Inria Paris and Univ. of Trento
Paris, France

Arthur Azevedo de Amorim
CMU
Pittsburgh, USA

Roberto Blanco
Inria Paris
Paris, France

Ana Nora Evans
Inria Paris and Univ. of Virginia
Charlottesville, USA

Guglielmo Fachini
Inria Paris
Paris, France

Cătălin Hriţcu
Inria Paris
Paris, France

Théo Laurent
Inria Paris and ENS Paris
Paris, France

Benjamin C. Pierce
Univ. of Pennsylvania
Philadelphia, USA

Marco Stronati
Inria Paris
Paris, France

Andrew Tolmach
Portland State Univ.
Portland, USA

ABSTRACT

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 careful proof (mostly machine-checked 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.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3243745

CCS CONCEPTS

• **Security and privacy → Security requirements**; *Formal methods and theory of security*; *Formal security models*; *Logic and verification*; *Software and application security*; • **Software and its engineering →** *Compilers*; *Modules / packages*; *Dynamic analysis*;

KEYWORDS

secure compilation; formal definition; low-level attacks; undefined behavior; compartmentalization; mutually distrustful components; dynamic compromise; software fault isolation; reference monitors; safety properties; machine-checked proofs; testing; foundations

ACM Reference Format:
Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Cătălin Hriţcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243745

1 INTRODUCTION

Compartmentalization offers a strong, practical defense against a range of devastating low-level attacks, such as control-flow hijacks exploiting buffer overflows and other vulnerabilities in C, C++, and other unsafe languages [18, 33, 81]. Widely deployed compartmentalization technologies include process-level privilege separation [18, 33, 47] (used in OpenSSH [67] and for sandboxing plugins and tabs in web browsers [69]), software fault isolation [74, 79] (e.g., Google Native Client [84]), WebAssembly modules [34] in modern web browsers, and hardware enclaves (e.g., Intel SGX [38]); many more are on the drawing boards [14, 20, 71, 81]. These mechanisms offer an attractive base for building more secure compilation chains that mitigate low-level attacks [30, 33, 44, 65, 75–77]. In particular, compartmentalization can be applied in unsafe low-level languages to structure large, performance-critical applications into mutually distrustful components that have clearly specified privileges and interact via well-defined interfaces.

Intuitively, protecting each component from all the others should bring strong security benefits, since a vulnerability in one component need not compromise the security of the whole application. Each component will be protected from all other components for as long as it remains "good." If, at some point, it encounters an internal vulnerability such as a buffer overflow, then, from this point on, it is assumed to be compromised and under the control of the attacker, potentially causing it to attack the remaining uncompromised components. The main goal of this paper is to formalize this dynamic-compromise intuition and precisely characterize what it means for a compilation chain to be secure in this setting.

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 [15, 73] (e.g., accessing an array out of bounds, or using a pointer after its memory region has been freed) and type safety violations [27, 35] (e.g., invalid unchecked casts)—cause real C compilers to produce code that behaves arbitrarily, often leading to exploitable vulnerabilities [37, 73].

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 [39] call out large numbers of undefined behaviors [36, 49] that are less familiar, even to experienced C/C++ developers [54, 80]. 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 [56, 68], providing a significantly cleaner model of undefined behavior than other C compilers [68].

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 [1, 4–8, 29, 63]. Second, unlike some recent criteria that do consider modular protection [24, 65], 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.* [43], 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 [73]. 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* [1]. To support dynamic compromise scenarios, we take an unconventional approach, dropping full abstraction and instead phrasing our criterion in terms of preserving safety properties [52] in adversarial contexts [6], 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. Finally, we expect our criterion to scale naturally from properties to hyperproperties such as confidentiality [6] (see §5 and §6).

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. [6], and extend this first to dynamic compromise (RSC^{DC}, §3.2), then mutually distrustful components (RSC^{DC}_{MD}, §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. We then use standard simulation proofs to relate our semantics for whole programs to semantics that capture the behavior of a partial program in an arbitrary context. This proof architecture yields simpler and more scalable proofs than previous work in this space [43]. One particularly important advantage is that it allows us to reuse a whole-program compiler correctness result à la CompCert [56] 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 [57] (§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 careful proofs—many of them machine-checked 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^{DC}_{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/

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 [33, 78] 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 [56]. 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

```
component C0 {
  export valid;
  valid(data) { ... }
}
component C1 {
  import E.read, C2.init, C2.process;
  main() {
    C2.init();
    x := E.read();
    y := C1.parse(x);    //(V1) can yield Undef for some x
    C2.process(x,y);
  }
  parse(x) { ... }
}
component C2 {
  import E.write, C0.valid;
  export init, process;
  init() { ... }
  process(x,y) {
    C2.prepare();        //(V2) can yield Undef if not initialized
    data := C2.handle(y); //(V3) can yield Undef for some y
    if C0.valid(data) then E.write(<data,x>)
  }
  prepare() { ... }
  handle(y) { ... }
}
```

Figure 1: Pseudocode of compartmentalized application

```
component C0 {
  import E.read, E.write, C2.init, C1.parse, C2.process;
  main() {
    C2.init();
    x := E.read();
    y := C1.parse(x);
    data := C2.process(y);
    if C0.valid(data) then E.write(<data,x>)
  }
  valid(data) { ... }
}
component C1 {
  export parse;
  parse(x) { ... }        //(V1) can yield Undef for some x
}
component C2 {
  export init, process;
  init() { ... }
  process(y) {
    C2.prepare();        //(V2) can yield Undef if not initialized
    return C2.handle(y); //(V3) can yield Undef for some y
  }
  prepare() { ... }
  handle(y) { ... }
}
```

Figure 2: More secure refactoring of the application

undefined behavior, causing it to start attacking the remaining uncompromised components. In contrast to earlier static-compromise models [43], 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 [33]. If the consequences of some plausible compromise seem too serious, developers can further reduce or separate privilege 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 ($\mathtt{C}_0$, $\mathtt{C}_1$, and $\mathtt{C}_2$) that interact with the environment `E` via input (`E.read`) and output (`E.write`) operations. Component $\mathtt{C}_1$ defines a `main()` procedure, which first invokes $\mathtt{C}_2$`.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 $\mathtt{C}_2$`.process(x,y)`. This, in turn, calls $\mathtt{C}_2$`.prepare()` and $\mathtt{C}_2$`.handle(y)`, obtaining some `data` that it validates using $\mathtt{C}_0$`.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_1) any call `E.write(<data,x>)` happens as a response to a previous `E.read()` call by $\mathtt{C}_1$ obtaining the request `x`; and
- (S_2) the application only writes valid data (i.e., data for which $\mathtt{C}_0$`.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:

- (V_1) $\mathtt{C}_1$`.parse(x)` performs complex array computations, and we do not know if it is immune to buffer overflows for all `x`;
- (V_2) $\mathtt{C}_2$`.prepare()` is intended to be called only if $\mathtt{C}_2$`.init()` has been called beforehand to set up a shared data structure; otherwise, it might dereference an undefined pointer;
- (V_3) $\mathtt{C}_2$`.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 $\mathtt{C}_1$ can get compromised and call $\mathtt{C}_2$`.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 $\mathtt{C}_1$ is not compromised, this undefined behavior cannot occur, since $\mathtt{C}_2$`.init()` will be called before $\mathtt{C}_2$`.prepare()`. Moreover, this undefined behavior cannot occur even if $\mathtt{C}_1$ *is* compromised by the undefined behavior in V_1, because that can only occur *after* $\mathtt{C}_2$`.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 $\mathtt{C}_1$ can be treated as honest and gets guarantees until it encounters undefined behavior. If instead we were only allowed to reason about $\mathtt{C}_1$'s ability to do damage based on its *interface*, as would happen in a model of static compromise [43], we wouldn't be able to conclude that $\mathtt{C}_2$ cannot be compromised: an arbitrary component with the same interface as $\mathtt{C}_1$ could indeed compromise $\mathtt{C}_2$ by calling $\mathtt{C}_2$`.process` before $\mathtt{C}_2$`.init`. Finally, if execution encounters undefined behavior in V_3, then $\mathtt{C}_2$ can get compromised irrespective of whether $\mathtt{C}_1$ is compromised beforehand, invalidating both S_1 and S_2.

Though we have not yet made it formal, this security analysis already identifies $\mathtt{C}_2$ 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 $\mathtt{C}_2$`.handle` could be hardened to reduce its chances of encountering undefined behavior, e.g. by doing better input validation. Or $\mathtt{C}_1$ could validate the values it sends to $\mathtt{C}_2$`.process`, so that an attacker would have to compromise both $\mathtt{C}_1$

and $\mathtt{C}_2$ to break the validity of writes. To ensure the correspondence of reads and writes despite the compromise of $\mathtt{C}_1$, we could make $\mathtt{C}_2$ read the request values directly from `E`, instead of via $\mathtt{C}_1$.

To achieve the best security though, we can refactor so that the read and write privileges are isolated in $\mathtt{C}_0$, which performs no complex data processing and thus is a lot less likely to be compromised by undefined behavior (Figure 2). In this variant, $\mathtt{C}_0$ reads a request, calls $\mathtt{C}_1$`.parse` on this request, passes the result to $\mathtt{C}_2$`.process`, validates the data $\mathtt{C}_2$ returns and then writes it out. This way both our desired properties hold even if both $\mathtt{C}_1$ and $\mathtt{C}_2$ 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) $\mathtt{C}_0$`.valid(data)` was called previously and returned `true`.

The refactored application in Figure 2 achieves both properties despite the compromise of both $\mathtt{C}_1$ via V_1 and $\mathtt{C}_2$ 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 $\mathtt{C}_1$`.parse(x')` *or* $\mathtt{C}_2$`.process(x,y)` was called previously with a `y` that can cause undefined behavior in $\mathtt{C}_2$`.handle(y)`, and

(W_2) $\mathtt{C}_0$`.valid(data)` was called previously and returned `true` *or* $\mathtt{C}_2$`.process(x,y)` was called previously with a `y` that can cause undefined behavior in $\mathtt{C}_2$`.handle(y)`.

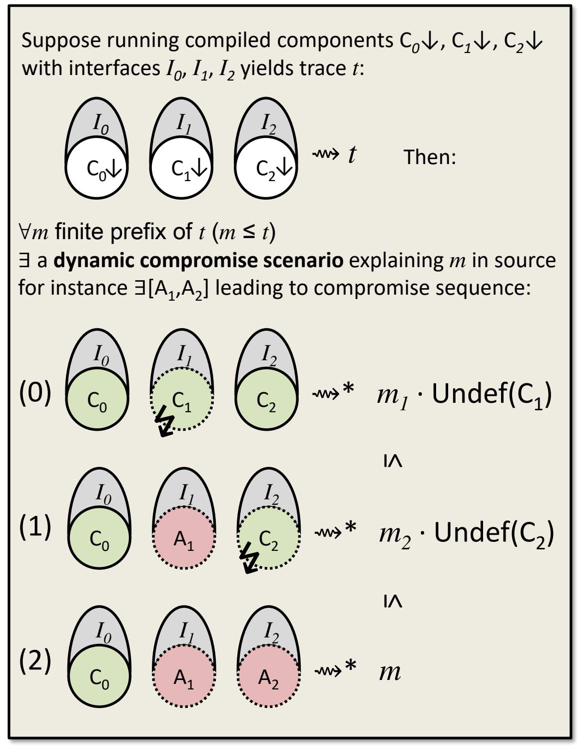
While these properties are significantly weaker (and harder to understand), they are still not trivial; in particular, they still tell us something useful under the assumption that the attacker has not actually discovered how to compromise $\mathtt{C}_1$ or $\mathtt{C}_2$.

Properties S_1, S_2, W_1, W_2 are all *safety properties* [52]—inspired, in this case, by the sorts of "correspondence assertions" used to specify authenticity in security protocols [31, 83]. 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 S_2 that includes a call to `E.write(<data,x>)` with no preceding call to $\mathtt{C}_0$`.valid(data)`:

```
[C0.main();C2.init();Ret;E.read;Ret(x);C1.parse(x);
 Ret(y);C2.process(y);Ret(data);E.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 $\mathtt{C}_0$`.main()`; this variant would invalidate safety property S_2.

Compiler correctness is often phrased in terms of preserving trace properties in general [56] (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.



The trace prefixes m, m_1, m_2 might, for instance, be:

```
m  = [C0.main();C2.init();Ret;E.read;Ret(x);C1.parse(x);
      Ret(y);C2.process(y);Ret(d);
      C0.valid(d);Ret(true);E.write(<d,x>)]

m1 = [C0.main();C2.init();Ret;E.read;Ret(x);C1.parse(x)]

m2 = [C0.main();C2.init();Ret;E.read;Ret(x);C1.parse(x);
      Ret(y);C2.process(y)]
```

Figure 3: The *RSCC* dynamic compromise game for our example. We start with all components being uncompromised (in green) and incrementally replace any component that encounters undefined behavior with an arbitrary component (in red) that has the same interface and will do its part of the trace prefix *m* without causing 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 any 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 $\mathtt{C}_0\!\downarrow$, $\mathtt{C}_1\!\downarrow$ and, $\mathtt{C}_2\!\downarrow$ producing trace t in the target language, we can explain any (intuitively bad) finite prefix m of t (written $m \leq t$) 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 [56], we model undefined behavior in our source language as a special event $\mathtt{Undef}(\mathtt{C}_i)$ that terminates the trace. For instance, in step 0 of Figure 3, component $\mathtt{C}_1$ is the first to encounter undefined behavior after producing a prefix m_1 of m.

Since undefined behavior can manifest as arbitrary target-level behavior, the further actions of component $\mathtt{C}_1$ 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 $\mathtt{C}_1$, the component that encountered undefined behavior, with some other source component $\mathtt{A}_1$ 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 $\mathtt{C}_1$ with $\mathtt{A}_1$ we have to go back in time and re-execute the program from the beginning obtaining a longer trace, in this case $m_2 \cdot \mathtt{Undef}(\mathtt{C}_2)$ (where we write "$\cdot$" for appending the event $\mathtt{Undef}(\mathtt{C}_2)$ to m_2). 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 $\mathtt{C}_1$ with $\mathtt{A}_1$ and $\mathtt{C}_2$ with $\mathtt{A}_2$, after which the program can produce the whole prefix m in the source.

Let's now use this *RSCC* security game to deduce that in our example from Figure 2, even compromising both $\mathtt{C}_1$ and $\mathtt{C}_2$ does not break property S_2 at the target level. Assume, for the sake of a contradiction, that a trace of our compiled program breaks property S_2. Then there exists a finite prefix "$m \cdot \mathtt{E.write(<data,x>)}$" such that $\mathtt{C}_0$`.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 $\mathtt{C}_1$ and $\mathtt{C}_2$ as in Figure 3. In this case, replacing $\mathtt{C}_1$ and $\mathtt{C}_2$ with arbitrary $\mathtt{A}_1$ and $\mathtt{A}_2$ 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 [72], that this cannot happen, since the source code of $\mathtt{C}_0$ does call $\mathtt{C}_0$`.valid(data)` and only if it gets `true` back does it call `E.write(<data,x>)`.

While in this special case we have only used the last step in the dynamic compromise sequence, where all compromised components have already been replaced (step 2 from Figure 3), the previous steps are also useful in general for reasoning about the code our original components execute *before* they get compromised. For instance, this kind of reasoning is crucial for showing property W_2 for the original example from Figure 1. Property W_2 gives up on the validity of the written data only if $\mathtt{C}_2$ receives a `y` that exploits $\mathtt{C}_2$`.handle(y)` (vulnerability V_3). However, as discussed above, a compromised $\mathtt{C}_1$ could, in theory, try to compromise $\mathtt{C}_2$ by calling $\mathtt{C}_2$`.process` without proper initialization (exploiting vulnerability V_2). Showing that this cannot actually happen requires using step 0 of the game from Figure 3, which gives us that the original compiled program obtained by linking $\mathtt{C}_0{\downarrow}$, $\mathtt{C}_1{\downarrow}$ and, $\mathtt{C}_2{\downarrow}$ can produce the trace $m_1 \cdot \mathtt{Undef}(\mathtt{C}_1)$, for some prefix m_1 of the bad trace prefix in which $\mathtt{C}_2$`.process` is called without calling $\mathtt{C}_2$`.init` first. But it is easy to check that the straight-line code of the $\mathtt{C}_1$`.main()` procedure can only cause undefined behavior *after* it has called $\mathtt{C}_2$`.init`, contradicting the existence of a bad trace exploiting V_2.

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. [6] (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.

3.1 *RSC*: Robustly Safe Compilation

RSC [6] 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 [31, 51, 72]. A trace property (i.e., a set of potentially infinite traces built over events like I/O with the environment [56]) is a safety property [52] if any trace violating 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 $P{\rightsquigarrow}t$ 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 linking 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 ($P_T{\rightsquigarrow}t$) including for instance the target machine, loader, and deployed protection mechanisms.

Definition 3.1. A compilation chain provides *RSC* iff

$$\forall P\ C_T\ t.\ C_T[P{\downarrow}]{\rightsquigarrow}t \Rightarrow \forall m{\leq}t.\ \exists C_S\ t'.\ C_S[P]{\rightsquigarrow}t' \wedge m{\leq}t'.$$

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 ($C_T[P{\downarrow}]{\rightsquigarrow}t$), 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., $C_S[P]{\rightsquigarrow}t'$ 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 $P{\downarrow}$ 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 $C_T[P{\downarrow}]$ 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 $P{\downarrow}$ when linked with an arbitrary target context [6].

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

3.2 *RSC^{DC}*: 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 `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 `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 (RSC^{DC})* iff

$$\forall P\, C_T\, t.\, C_T[P\downarrow] \rightsquigarrow t \Rightarrow \forall m{\leq}t.\, \exists C_S\, t'.\, C_S[P] \rightsquigarrow t' \wedge (m{\leq}t' \vee t' {\prec_P} m).$$

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 $t'{\prec_P}m$ (i.e., $t'{\prec_P}m \triangleq \exists m'{\leq}m.\ t'{=}(m' \cdot \mathtt{Undef}(P))$). 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.process(x);
    E.write(y);
  }
  // can encounter Undef for some x
  process(x) { ... }
}
```

```
context C_S {
  import E.read, P.foo;
  main() {
    x := E.read();
    P.foo(x);
  }
}
```

Suppose we compile P with a compilation chain that satisfies RSC^{DC}, link the result with a target context C_T obtaining $C_T[P\downarrow]$, execute this and observe the following finite trace prefix:

```
m = [E.read(); Ret("feedbeef"); P.foo("feedbeef"); E.write("bad")]
```

According to RSC^{DC} 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' = [E.read(); Ret("feedbeef"); P.foo("feedbeef"); Undef(P)]
```

As in CompCert [56, 68], 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 [68], 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 RSC^{DC}. Using our notations this CompCert theorem looks as follows:

$$\forall P\ t.\ (P\downarrow) \rightsquigarrow t \Rightarrow \exists t'.\ P \rightsquigarrow t' \wedge (t'{=}t \vee t'{\prec}t)$$

This says that if a compiled whole program $P\downarrow$ 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 RSC^{DC} we can already imagine $P\downarrow$ 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 RSC) RSC^{DC} 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 RSC^{DC} even in adversarial contexts.

3.3 RSC^{DC}_{MD}: Mutually Distrustful Components

RSC^{DC} 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 *et al.*'s work in the full abstraction setting [43], 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 *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 (RSC^{DC}_{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,

$$\begin{aligned}&\forall P{:}I_P.\ \forall C_T{:}I_C.\ \forall t.\ (C_T \cup P\downarrow) \rightsquigarrow t \Rightarrow \forall m{\leq}t.\\ &\quad \exists t'.\ (\{(m, I_i)\uparrow \mid I_i \in I_C\} \cup P) \rightsquigarrow t' \wedge (m{\leq}t' \vee t'{\prec_{I_P}}m).\end{aligned}$$

This definition closely follows RSC^{DC}, 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-translated context components ($t'{\prec_{I_P}}m$).

3.4 Formalizing *RSCC*

Using these ideas, we now define *RSCC* by following the dynamic compromise game illustrated in Figure 3. We use the notation $P{\rightsquigarrow}^* m$ when there exists a trace t that extends m (i.e., $m \leq t$) such that $P{\rightsquigarrow}t$. 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 (RSCC)* iff ∀compatible interfaces $I_1, \ldots, I_n$,

$$\forall C_1{:}I_1, \ldots, C_n{:}I_n.\ \forall m.\ \{C_1{\downarrow}, \ldots, C_n{\downarrow}\}{\rightsquigarrow}^* m \Rightarrow$$
$$\exists A_{i_1}{:}I_{i_1}, \ldots, A_{i_k}{:}I_{i_k}.$$
$$(1)\ \forall j \in 1 \ldots k.\ \exists m_j.\ (m_j \prec_{I_{i_j}} m)\ \wedge (m_{j-1} \prec_{I_{i_{j-1}}} m_j)\ \wedge$$
$$(\{C_1, \ldots, C_n\} \backslash \{C_{i_1}, \ldots, C_{i_{j-1}}\} \cup \{A_{i_1}, \ldots, A_{i_{j-1}}\}){\rightsquigarrow}^* m_j$$
$$\wedge\ (2)\ \ (\{C_1, \ldots, C_n\} \backslash \{C_{i_1}, \ldots, C_{i_k}\} \cup \{A_{i_1}, \ldots, A_{i_k}\}){\rightsquigarrow}^* m.$$

This says that $C_{i_1}, \ldots, C_{i_k}$ constitutes a compromise sequence corresponding to finite prefix m produced by a compiled set of components $\{C_1{\downarrow}, \ldots, C_n{\downarrow}\}$. In this compromise sequence each component C_{i_j} is taken over by the already compromised components at that point in time $\{A_{i_1}, \ldots, A_{i_{j-1}}\}$ (part 1). Moreover, after replacing all the compromised components $\{C_{i_1}, \ldots, C_{i_k}\}$ with their corresponding source components $\{A_{i_1}, \ldots, A_{i_k}\}$ 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 [43] (further discussed in §5).

3.5 A Generic Proof Technique for *RSCC*

We now describe an effective and general proof technique for *RSCC*. First, we observe that the slightly simpler RSC^{DC}_{MD} implies *RSCC*. Then we provide a generic proof in Coq that any compilation chain obeys RSC^{DC}_{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 [43]. 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 [45]; 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. Finally, since only the last step of our proof technique is specific to unsafe languages, we expect it can be further simplified to provide scalable proofs of vanilla *RSC* for safe source languages [6, 66].

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

Theorem 3.5. RSC^{DC}_{MD} implies *RSCC*.

We proved this by defining a non-constructive function that produces the compromise sequence $A_{i_1}, \ldots, A_{i_1}$ 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 $t' {\prec_P} m$ 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 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 $C_T \cup P{\downarrow}$ 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 (Assumptions 2 and 3). Now from $(C_T \cup P{\downarrow}){\rightsquigarrow}^* m$ and $(C_S{\downarrow} \cup P'{\downarrow}){\rightsquigarrow}^* m$ we would like to obtain $(C_S{\downarrow} \cup P{\downarrow}){\rightsquigarrow}^* m$ by first "decomposing" (Assumption 4) separate executions for $P{\downarrow}$ and $C_S{\downarrow}$, which we can then "compose" (Assumption 5) again into a complete execution for $(C_S{\downarrow} \cup P{\downarrow})$. However, since $P{\downarrow}$ and C_S are not complete programs, how should they execute? To answer this we rely on a *partial semantics* that captures the traces of a partial program when linked with *any* context satisfying a given interface. When the partial program is running, execution is the same as in the normal operational semantics of the target language; when control is passed to the context, arbitrary actions compatible with its interface are non-deterministically executed. Using this partial semantics we can execute $C_S{\downarrow}$ with respect to the interface of $P{\downarrow}$, and $P{\downarrow}$ with respect to the interface of $C_S{\downarrow}$, as needed for the decomposition and composition steps of our proof.

Once we know that $(C_S{\downarrow} \cup P{\downarrow}){\rightsquigarrow}^* m$, we use compiler correctness again—now in the backwards direction (Assumptions 3 and 6)—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 7).

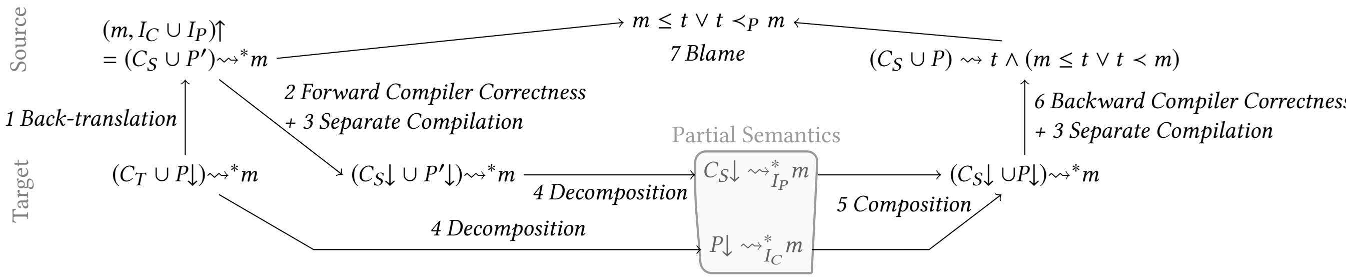


Figure 4: Outline of our generic proof technique for RSC^{DC}_{MD}

Assumptions of the RSC^{DC}_{MD} proof The generic RSC^{DC}_{MD} proof outlined above relies on assumptions about the compartmentalizing compilation chain. In the reminder of this subsection we give details about these assumptions, while still trying to stay 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., $(m, I_P)\Uparrow \rightsquigarrow^* m$).

Assumption 1 (Back-translation).

$$\exists \Uparrow.\ \forall P{:}I_P.\ \forall m \textit{ defined}.\ P\rightsquigarrow^* m \Rightarrow (m, I_P)\Uparrow : I_P \wedge (m, I_P)\Uparrow \rightsquigarrow^* m$$

Translating only finite prefixes simplifies 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 {`call`, `return`}, there is no program that produces the single event trace `return`, since every return must be preceded by a call. Thus we can 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 [41, 64] and very recently *RSC* [66] and even stronger variants [6]. 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, $(m, I_C \cup I_P)\Uparrow$, 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 [56]. It says that if the whole program P can produce the finite trace prefix m that does not end with undefined behavior (m *defined*) then P's compilation $P\downarrow$ can also produce m.

Assumption 2 (Forward Compiler Correctness).

$$\forall P.\ \forall m \textit{ defined}.\ P\rightsquigarrow^* m \Rightarrow P\downarrow \rightsquigarrow^* m$$

In Figure 4, we apply this assumption to $(C_S \cup P')\rightsquigarrow^* m$ to obtain $(C_S \cup P')\downarrow \rightsquigarrow^* m$ and then, we distribute compilation over linking using the following separate compilation assumption:

Assumption 3 (Separate Compilation).

$$\forall P\ C\ m.\ (C \cup P)\downarrow \rightsquigarrow^* m \Leftrightarrow (C\downarrow \cup P\downarrow)\rightsquigarrow^* m$$

The next assumption we make is *decomposition*, stating 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:

Assumption 4 (Decomposition).

$$\forall P_T{:}I_P.\ \forall C_T{:}I_C.\ \forall m \textit{ defined}.\ (C_T \cup P_T)\rightsquigarrow^* m \Rightarrow P_T \rightsquigarrow^*_{I_C} m$$

The converse of decomposition, *composition*, states 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:

Assumption 5 (Composition). For any I_P, I_C compatible interfaces:

$$\forall P_T{:}I_P.\ \forall C_T{:}I_C.\ \forall m.\ P_T \rightsquigarrow^*_{I_C} m \wedge C_T \rightsquigarrow^*_{I_P} m \Rightarrow (C_T \cup P_T)\rightsquigarrow^* m$$

When taken together, composition and decomposition capture that the partial semantics of the target language is adequate with respect to its complete counterpart. This adequacy notion is tailored to the *RSC* property and thus different from the requirement that a so called "trace semantics" is fully abstract [41, 64].

In order to get back to the source language our proof uses again separate compilation together with a backwards compiler correctness assumption. As also explained in §3.2, we need to take into account that a trace prefix m of $P\downarrow$ 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 6 (Backward Compiler Correctness).

$$\forall P\ m.\ P\downarrow \rightsquigarrow m \Rightarrow \exists t.\ P\rightsquigarrow t \wedge (m \leq t \vee t{<}m)$$

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

Assumption 7 (Blame). $\forall C_S : I_C.\ \forall P, P' : I_P.\ \forall m \textit{ defined}.\ \forall t.$
If $(C_S \cup P')\rightsquigarrow^* m$ and $(C_S \cup P)\rightsquigarrow t$ and $t \prec m$ then $m \leq t \vee t \prec_P m$.

We used Coq to prove the following theorem that puts together the assumptions from this subsection to show RSC^{DC}_{MD}:

Theorem 3.6. The assumptions above imply RSC^{DC}_{MD}.

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 `Alloc` instruction for dynamic storage allocation plus cross-component `Call` and `Return` instructions (§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 *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 *Micro-policies back end* (§4.5), on the other hand, relies on specialized hardware [26] 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 [26] and are a good target for formal verification [14] 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 [81] could also be used to implement the compartmentalized machine and achieve *RSCC*.

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. [59].) 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 *RSCC* in Coq (§4.3) using the proof technique of §3.5. For the SFI back end, we use property-based testing with QuickChick [62] to systematically test RSC^{DC}_{MD}.

`e ::=`	`v`	values
	`| local`	local static buffer
	`|` $e_1 \otimes e_2$	binary operations
	`|` e_1`;` e_2	sequence
	`| if` e_1 `then` e_2 `else` e_3	conditional
	`| alloc e`	memory allocation
	`| !e`	dereferencing
	`|` e_1 `:=` e_2	assignment
	`| C.P(e)`	procedure call
	`| exit`	terminate

Figure 5: Syntax of source language expressions

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 $\top$, 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 [55], our semantics propagates these $\top$ values and yields an undefined behavior if a $\top$ 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 [57]. 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(n) C'`, recording that component `C` called procedure `P` of component `C'`, 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.

```
instr ::= Nop        |    Halt        | Jal l
        | Const i -> r               | Jump r
        | Mov r_s -> r_d             | Call C P
        | BinOp r_1⊗r_2 -> r_d       | Return
        | Load *r_p -> r_d           | Bnz r l
        | Store *r_p <- r_s          | Alloc r_1 r_2
```

Figure 6: Instructions of compartmentalized machine

$$\frac{\begin{array}{l} \mathit{fetch}[pc] = \texttt{Call}\ C'\ P \qquad C \neq C' \\ P \in C.\mathit{import} \qquad pc' = E[C'][P] \\ \mathit{reg}' = \mathit{reg}_\top[\texttt{R\_COM} \leftarrow \mathit{reg}[\texttt{R\_COM}], \texttt{R\_RA} \leftarrow pc+1] \\ \alpha = C\ \texttt{Call}(P, \mathit{reg}[\texttt{R\_COM}])\ C' \end{array}}{(C, \sigma, \mathit{mem}, \mathit{reg}, pc) \xrightarrow{\alpha} (C', (pc+1) :: \sigma, \mathit{mem}, \mathit{reg}', pc')}$$

$$\frac{\begin{array}{l} \mathit{fetch}(pc) = \texttt{Return} \qquad C \neq C' \\ \mathit{reg}[\texttt{R\_RA}] = pc' \\ \mathit{reg}' = \mathit{reg}_\top[\texttt{R\_COM} \leftarrow \mathit{reg}[\texttt{R\_COM}]] \\ \alpha = C\ \texttt{Return}(\mathit{reg}[\texttt{R\_COM}])\ C' \end{array}}{(C, pc' :: \sigma, \mathit{mem}, \mathit{reg}, pc) \xrightarrow{\alpha} (C', \sigma, \mathit{mem}, \mathit{reg}', pc')}$$

Figure 7: Compartmentalized machine 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 σ, 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 α recording the caller, the callee, the procedure and its argument, which is stored in register `R_COM`. The protected stack σ is updated with a new frame containing the next point in the code of the current component. Registers are mostly invalidated at `Call`s; $\mathit{reg}_\top$ has all registers set to $\top$ and only two registers are passed on: `R_COM` 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 compiler saves and restores the registers. Finally, there is a redundancy between the protected stack and `R_RA` because during the `Return` the protected frame is used to verify that the register is used correctly; otherwise the program has an undefined behavior.

4.3 *RSCC* Proof in Coq

We have proved that a compilation chain targeting the compartmentalized machine satisfies *RSCC*, 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 RSC^{DC}_{MD}, all the assumptions from §3.5 are proved using simulations. Most of this proof is formalized in Coq: the only non-trivial missing pieces are compiler correctness (Assumptions 2 and 6) and composition (Assumption 5). The first 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. A mechanized proof of composition is underway. Despite these missing pieces, our formalization is more detailed than previous paper proofs in the area [3, 5, 9–11, 29, 40, 41, 43, 61, 63–65]. Indeed, we are aware of only one fully mechanized proof about secure compilation: Devriese *et al.*'s [24] recent full abstraction result for a translation from the simply typed to the untyped λ-calculus in around 11KLOC of Coq.

Our Coq development comprises around 22KLOC, 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 $\uparrow$ function that takes a finite trace prefix m and a program interface I and returns a whole source program that respects I and produces m. Each generated component uses the local variable `local[0]` to track how many events it has emitted. When a procedure is invoked, it increments `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 I 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 `MainC.mainP`, with all counters set to 0, so after testing the value of `MainC.local[0]`, the program runs the first branch of `mainP`:

```
local[0]++; C.p(0); MainC.mainP(0);
```

After bumping `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`:

```
local[0]++; 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.

```
ECall MainC p      0 C
ERet  C            1 MainC
ECall MainC p      2 C
ECall C      mainP 3 MainC
```

(a) Trace of 4 events

```
MainC {
  mainP(_) {
    if (local[0] == 0) {
      local[0]++;
      C.p(0);
      MainC.mainP(0);
    } else if (local[0] == 1) {
      local[0]++;
      C.p(2);
      MainC.mainP(0);
    } else {
      exit();
    }
  }
}
```

(b) Component `MainC`

```
C {
  p(_) {
    if (local[0] == 0) {
      local[0]++;
      return 1;
    } else if (local[0] == 1) {
      local[0]++;
      MainC.mainP(3);
      C.p(0);
    } else {
      exit();
    }
  }
}
```

(c) Component `C`

Figure 8: Example of program with two components back-translated from a trace of 5 events.

Partial semantics. Our partial semantics has a simple generic definition based on the small-step operational semantics of a *whole* target program, which we denote as $\xrightarrow{\alpha}$. In this semantics, each step is labeled with an action α that is either an event or a silent action τ. The definition of the partial semantics $\overset{\alpha}{\rightharpoonup}$ uses a *partialization* function $\mathtt{par}$ that, given a complete state cs and the interface I_C of a program part C, returns a partial state ps where all information about C (such as its memory and stack frames) is erased.

$$\frac{\mathtt{par}(cs, I_C) = ps \quad \mathtt{par}(cs', I_C) = ps' \qquad cs \xrightarrow{\alpha} cs'}{ps \overset{\alpha}{\rightharpoonup} ps'}$$

The partial semantics can step with action α from the partial state ps to ps', if there exists a corresponding transition in the complete semantics whose states partialize to ps and ps'. We denote with $P{\rightsquigarrow}^*_{I_C} m$ that the partial program P produces the trace prefix m in the partial semantics after a finite execution prefix, with respect to the context interface I_C.

A consequence of abstracting away part of the program as non-deterministic actions allowed by its interface is that the abstracted part will always have actions it can do and it will never be stuck, whereas stuckness is the standard way of modeling undefined behavior [56]. Given $P{\rightsquigarrow}^*_{I_C} m$, if m ends with an undefined behavior, then this was necessarily caused by P, which is still a concrete partial program running actual code, potentially unsafe.

Our partial semantics was partially inspired by so-called "trace semantics" [41, 43, 64], where a partial program of interest is decoupled from its context, of which only the observable behavior is relevant. One important difference is that our definition of partial semantics in terms of a partialization function is generic and can be easily instantiated for different languages. On the contrary previous works defined "trace semantics" as separate relations with many rules, making the proofs to correlate partial and complete semantics more involved. Moreover, by focusing on trace properties (instead of observational equivalence) composition and decomposition can be proved using standard simulations à la CompCert, which is easier than previous proof techniques for fully abstract "trace semantics."

Theorem 4.2 (Partial Semantics). The source language and compartmentalized machine partial semantics defined as described above provide decomposition and composition (Assumptions 4 and 5).

Blame. We prove Assumption 7 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 cs_1 and cs_2 have the same context state, which, borrowing the partialization notation from above, we write as $\mathtt{par}(cs_1, I_P) = \mathtt{par}(cs_2, I_P)$. Then:

- If $cs_1 \xrightarrow{\alpha_1} cs_1'$, $cs_2 \xrightarrow{\alpha_2} cs_2'$, and C_S has control in cs_1 and cs_2, then $\alpha_1 = \alpha_2$ and $\mathtt{par}(cs_1', I_P) = \mathtt{par}(cs_2', I_P)$.
- If $cs_1 \xrightarrow{\tau} cs_1'$ and the program has control in cs_1 and cs_2, then $\mathtt{par}(cs_1', I_P) = \mathtt{par}(cs_2, I_P)$.
- If $cs_1 \xrightarrow{\alpha} cs_1'$, the program has control in cs_1 and cs_2, and $\alpha \neq \tau$, then there exists cs_2' such that $cs_2 \xrightarrow{\alpha} cs_2'$ and $\mathtt{par}(cs_1', I_P) = \mathtt{par}(cs_2', I_P)$.

By repeatedly applying these properties, we can analyze the behavior of two parallel executions $(C_S \cup P'){\rightsquigarrow}^* m$ and $(C_S \cup P) \rightsquigarrow t$, with $t \prec m$. By unfolding the definition of $t \prec m$ we get that $\exists m' {\leq} m.\ t = m' \cdot \mathtt{Undef}\ (\_)$. It suffices to show that $m {\leq} t \vee t{=}m' \cdot \mathtt{Undef}(P)$. If $m = t = m' \cdot \mathtt{Undef}\ (\_)$, 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 cs_1 and cs_2. 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 cs_2. Hence $t \prec_P m$.

Theorem 4.3 (Blame). Assumption 7 is satisfied.

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. [79], 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 [58]. 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 [73] in memory accessible only from the SFI instrumentation sequences. Alignment of code [58] 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 [62] 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

Reserved (Code)	Component Code			Protected Stack	Component Data		
	1	2	3		1	2	3
Init Code	Slot 0	Slot 0	Slot 0	**Slot 1**	Slot 1	Slot 1	Slot 1
Unused	Slot 2	Slot 2	Slot 2	**Slot 3**	Slot 3	Slot 3	Slot 3
Unused	Slot 4	Slot 4	Slot 4	**Slot 5**	Slot 5	Slot 5	Slot 5
...	...	...	...	**...**	...	...	...

Figure 9: Memory layout of three user components

Slot (Unbounded)	Component Identifier (2 bits)	Offset (12 bits)

Figure 10: Address Example

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 RSC^{DC}_{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 to 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_T in the trace t_t (5-1) use back-translation to replace, in the program P, the component C_T with a component C_S without undefined behavior (5-2) run the new program on the compartmentalized machine and obtain a trace t_s (5-3) if the condition $t_t \leq t_s$ or $t_s \prec_{P \setminus \cup C_S} t_t$ 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 [14, 26]. 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 (110%) [26]. Moreover, the mechanism is simple enough so that the security of the reference monitors can be verified formally [12–15]. 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 [12, 14, 15].

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 records of the form $\{vtag = t_v, color = c, entry = cs\}$. The *vtag* field stores the tag of the payload value. The *color* field stores a component identifier c, which we call a color, of 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 *entry* field stores a (by default empty) set of colors identifying all the components that are allowed to call to this location. The value tags used by our monitor distinguish return addresses from all other words in the system: $t_v ::= Ret(n) \mid Any$. To enforce the stack discipline return addresses are treated as *linear return capabilities*, i.e., unique capabilities that cannot be duplicated and that can only be used to return once [48]. 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 label of the program counter: $t_{pc} ::= Level(n)$. Calls increment the counter n, while returns decrement it. A global invariant of the system 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. For `Mov`, `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. `Store` operations are only allowed if the color of the changed location matches the one of the currently executing instruction. `Bnz` is restricted to the current component. `Jal` 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 decremented 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.

Azevedo de Amorim *et al.* [14] 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.

5 RELATED WORK

Fully Abstract Compilation, originally introduced in seminal work by Abadi [1], 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 [1] and later Kennedy [46] 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.* [9–11, 61] proved the full abstraction of type-preserving compiler passes for functional languages. Abadi and Plotkin [5] and Jagadeesan *et al.* [40] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet *et al.* [29] devised a fully abstract compiler from a subset of ML to JavaScript. More recently, Patrignani *et al.* [63] 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 [38].

Modular, Fully Abstract Compilation. Patrignani *et al.* [65] 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.* [24] proved modular full abstraction by approximate back-translation for a compiler from simply typed to untyped λ-calculus. This work also introduces a complete Coq formalization for the original (non-modular) full abstraction proof of Devriese *et al.* [22].

Beyond Good and Evil. The closest related work is that of Juglaret *et al.* [43], 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, is 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. [6], who study several secure compilation criteria that are similar to fully abstract compilation, but that are phrased in terms of preserving hyperproperties [21] (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 model checking of open systems [51], the analysis of security protocols [31], and compositional verification [72].

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 [6].

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 rewind the execution along the same trace.

Proof Techniques. Abate et al. [6] 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, Patrignani and Garg [66] 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, 10, 11, 24, 29, 40, 61], fully abstract compilation proofs are notoriously difficult, even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled [23]. The proofs of Juglaret *et al.* [43] are no exception: while 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 Rathke [41] and adapted to low-level target languages by Patrignani and Clarke [64]. 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 Compartmentalization. Recent successes in formal verification have focused on showing correctness of low-level compartmentalization mechanisms based on software fault isolation [58, 85] or tagged hardware [14]. 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.* [82] working on a variant of CompCert with SFI, based on previous work by Kroll *et al.* [50]; we believe *RSCC* or RSC^{DC} could provide good top-level theorems for such an SFI compiler. In most work on verified compartmentalization [14, 58, 85], 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 [71]; they plan to simplify their calling convention using a notion of linear return capability [70] 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 [16, 17, 28, 32]. This model is also similar to Byzantine behavior in distributed systems [19, 53], 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 [14, 15].

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 [42], logical relations [25], 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 [60]. 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 [33], 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 [2]. In one of its simplest form, 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 [6, 21]. 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 Juglaret first studied with us the compartmentalization micro-policy from §4.5 and described it in an unpublished technical report [44]. Jérémy Thibault helped us correctly extract the micro-policies back end to OCaml. We are also grateful to Danel Ahman, Éric Tanter, 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 RSC^{DC}

Since *RSC* corresponds exactly to preserving robust safety properties [6], one might wonder what properties RSC^{DC} preserves. In fact, RSC^{DC} corresponds exactly to preserving the following class Z_P against an adversarial context:

Definition A.1. $Z_P \triangleq \mathit{Safety} \cap \mathit{Closed}_{\prec_P}$, where

$$\begin{aligned} \mathit{Safety} &\triangleq \{\pi \mid \forall t{\notin}\pi.\ \exists m{\leq}t.\ \forall t'{\geq}m.\ t'{\notin}\pi\} \\ \mathit{Closed}_{\prec_P} &\triangleq \{\pi \mid \forall t{\in}\pi.\ \forall t'.\ t{\prec_P}t' \Rightarrow t'{\in}\pi\} \\ &= \{\pi \mid \forall t'{\notin}\pi.\ \forall t.\ t{\prec_P}t' \Rightarrow t{\notin}\pi\} \end{aligned}$$

The class of properties Z_P is defined as the intersection of *Safety* and the class $\mathit{Closed}_{\prec_P}$ of properties closed under extension of traces with undefined behavior in P [56]. If a property π is in $\mathit{Closed}_{\prec_P}$ and it allows a trace t that ends with an undefined behavior in P—i.e., $\exists m.\ t = m \cdot \texttt{Undef}(P)$—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 $\mathit{Closed}_{\prec_P}$ rejects a trace t', then for any prefix m of t' the property π should also reject the trace $m \cdot \texttt{Undef}(P)$.

For a negative example that is not in $\mathit{Closed}_{\prec_P}$, consider the following formalization of the property S_1 from §2, requiring all writes in the trace to be preceded by a corresponding read:

$$\begin{aligned} S_1{=}\{t \mid \forall m\ \texttt{d}\ \texttt{x}.\ m \cdot \texttt{E.write(<d,x>)} \leq t \\ \Rightarrow \exists m'.\ m' \cdot \texttt{E.read} \cdot \texttt{Ret(x)} \leq m\} \end{aligned}$$

While property S_1 is *Safety* it is not $\mathit{Closed}_{\prec_P}$. Consider the trace $t' = [\texttt{C}_0\texttt{.main();}\ \texttt{E.write(<d,x>)}] \notin S_1$ that does a write without a read and thus violates S_1. For S_1 to be $\mathit{Closed}_{\prec_P}$ it would have to reject not only t', but also $[\texttt{C}_0\texttt{.main();}\ \texttt{Undef}(P)]$ and $\texttt{Undef}(P)$, which it does not. One can, however, define a stronger variant of S_1 that is in Z_P:

$$\begin{aligned} S_1^{Z_P^+}{=}\{t | \forall m\ \texttt{d}\ \texttt{x}.(m \cdot \texttt{E.write(<d,x>)}{\leq}t \vee m \cdot \texttt{Undef}(P){\leq}t) \\ \Rightarrow \exists m'.\ m' \cdot \texttt{E.read} \cdot \texttt{Ret(x)} \leq m\} \end{aligned}$$

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 $C = \{C_1, C_2\}$, since we assumed that C_0 has no undefined behavior.

Using Z_P, we proved an equivalent RSC^{DC} characterization:

Theorem A.2.
$$RSC^{DC} \iff \left(\begin{aligned} \forall P\ \pi{\in}Z_P.\ (\forall C_S\ t.\ C_S[P]{\rightsquigarrow}t \Rightarrow t{\in}\pi) \\ \Rightarrow (\forall C_T\ t.\ C_T[P{\downarrow}]{\rightsquigarrow}t \Rightarrow t{\in}\pi) \end{aligned} \right)$$

This theorem shows that 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 RSC^{DC}_{MD} (§3.3) we are able to effectively restrict the context from directly performing certain events (e.g., certain system calls), and the robust safety property can then be about these privileged events that the sandboxed context cannot directly perform.

One might also wonder what stronger property does one have to prove in the source in order to obtain a certain safety property π in the target using an RSC^{DC} compiler in the case in which π is not itself in Z_P. Especially when all undefined behavior is already gone in the target language, it seems natural to look at safety properties such as $S_1 \notin Z_P$ above that do not talk at all about undefined behavior. For S_1 above, we manually defined the stronger property $S_1^{Z_P^+} \in Z_P$ that is preserved by an RSC^{DC} compiler. In fact, given any safety property π we can easily define $\pi^{Z_P^+}$ that is in Z_P, is stronger than π, and is otherwise as permissive as possible:

$$\pi^{Z_P^+} \triangleq \pi \cap \{t \mid \forall t'.\ t{<_P}t' \Rightarrow t' \in \pi\}$$

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 RSC^{DC} compiler:

$$\pi^{Z_P^-} \triangleq \pi \cup \{t' \mid \exists t \in \pi.\ t{<_P}t' \vee t' \leq t\}$$

REFERENCES

[1] Martín Abadi. 1999. Protection in Programming-Language Translations. In *Security Issues for Mobile and Distributed Objects (Lecture Notes in Computer Science)*, Vol. 1603. Springer, 19–34. https://doi.org/10.1007/3-540-48749-2_2

[2] M. Abadi and C. Fournet. 2003. Access Control Based on Execution History. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*. The Internet Society, 107–121.

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2002. Secure Implementation of Channel Abstractions. *Information and Computation* 174, 1 (2002), 37–83. https://doi.org/10.1006/inco.2002.3086

[4] Martín Abadi and Jérémy Planul. 2013. On Layout Randomization for Arrays and Functions. In *2nd International Conference on Principles of Security and Trust (Lecture Notes in Computer Science)*, Vol. 7796. Springer, 167–185. https://doi.org/10.1007/978-3-642-36830-1_9

[5] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Transactions on Information and System Security* 15, 2 (2012), 8. https://doi.org/10.1145/2240276.2240279

[6] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2018. Exploring Robust Property Preservation for Secure Compilation. arXiv:1807.04603. https://arxiv.org/abs/1807.04603

[7] Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. In *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 581–594. https://doi.org/10.1145/2676726.2676972

[8] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors. In *25th IEEE Computer Security Foundations Symposium*. IEEE, 171–185. https://doi.org/10.1109/CSF.2012.12

[9] Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (LIPIcs)*, Vol. 32. Schloss Dagstuhl, 15–31. https://doi.org/10.4230/LIPIcs.SNAPL.2015.15

[10] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *13th ACM SIGPLAN international conference on Functional programming*. ACM, 157–168. https://doi.org/10.1145/1411204.1411227

[11] Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation Via Multi-Language Semantics. In *16th International Conference on Functional Programming*. ACM, 431–444. https://doi.org/10.1145/2034773.2034830

[12] Arthur Azevedo de Amorim. 2017. *A methodology for micro-policies*. Ph.D. Dissertation. University of Pennsylvania.

[13] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-Flow Architecture. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL)*. ACM, 165–178. https://doi.org/10.1145/2535838.2535839

[14] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hriţcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, 813–830. https://doi.org/10.1109/SP.2015.55

[15] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *7th International Conference on Principles of Security and Trust (POST)*. 79–105. https://doi.org/10.1007/978-3-319-89722-6_4

[16] Michael Backes, Martin P. Grochulla, Cătălin Hriţcu, and Matteo Maffei. 2009. Achieving Security Despite Compromise Using Zero-knowledge. In *22th IEEE Symposium on Computer Security Foundations (CSF 2009)*. IEEE, 308–323.

[17] David A. Basin and Cas Cremers. 2014. Know Your Enemy: Compromising Adversaries in Protocol Analysis. *ACM Trans. Inf. Syst. Secur.* 17, 2 (2014), 7:1–7:31. https://doi.org/10.1145/2658996

[18] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *USENIX Symposium on Networked Systems Design and Implementation*. 309–322.

[19] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[20] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[21] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393

[22] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-Abstract Compilation by Approximate Back-Translation. In *43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[23] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity Versus the Universal Type. *PACMPL* 2, POPL (2018), 38:1–38:23. https://doi.org/10.1145/3158126

[24] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-abstract Compilation by Approximate Back-translation. *LMCS* 13, 4 (2017). https://doi.org/10.23638/LMCS-13(4:2)2017

[25] Dominique Devriese, Frank Piessens, and Lars Birkedal. 2016. Reasoning About Object Capabilities with Logical Relations and Effect Parametricity. In *1st IEEE European Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/EuroSP.2016.22

[26] Udit Dhawan, Cătălin Hriţcu, Rafi Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. In *ASPLOS*. ACM, 487–502. https://doi.org/10.1145/2694344.2694383

[27] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. http://arxiv.org/abs/1710.06125

[28] Cédric Fournet, Andy Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization in Distributed Systems. In *20th IEEE Computer Security Foundations Symposium, CSF*. IEEE, 31–48. https://doi.org/10.1109/CSF.2007.7

[29] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 371–384. https://doi.org/10.1145/2429069.2429114

[30] Anitha Gollamudi and Cédric Fournet. 2018. Building Secure SGX Enclaves using F*, C/C++ and X64. 2nd Workshop on Principles of Secure Compilation (PriSC).

[31] Andrew D. Gordon and Alan Jeffrey. 2004. Types and Effects for Asymmetric Cryptographic Protocols. *Journal of Computer Security* 12, 3-4 (2004), 435–483.

[32] Andrew D. Gordon and Alan Jeffrey. 2005. Secrecy Despite Compromise: Types, Cryptography, and the Pi-Calculus. In *16th International Conference on Concurrency Theory, CONCUR (LNCS)*. Springer, 186–201. https://doi.org/10.1007/11539452_17

[33] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1016–1031. https://doi.org/10.1145/2810103.2813611

[34] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. 185–200. https://doi.org/10.1145/3062341.3062363

[35] István Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and*

Communications Security. 517–528. https://doi.org/10.1145/2976749.2978405
[36] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 336–345. https://doi.org/10.1145/2737924.2737979
[37] Heartbleed 2014. The Heartbleed Bug. http://heartbleed.com/.
[38] Intel 2014. Software Guard Extensions (Intel SGX) Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
[39] ISO/IEC. 2011. ISO/IEC 9899:2011 - Programming languages – C.
[40] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. 2011. Local Memory via Layout Randomization. In *24th IEEE Computer Security Foundations Symposium*. IEEE, 161–174. https://doi.org/10.1109/CSF.2011.18
[41] Alan Jeffrey and Julian Rathke. 2005. Java Jr: Fully Abstract Trace Semantics for a Core Java Language. In *14th European Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 3444. Springer, 423–438. https://doi.org/10.1007/978-3-540-31987-0_29
[42] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. 2015. A Logic of Programs with Interface-Confined Code. In *IEEE 28th Computer Security Foundations Symposium, CSF*. IEEE, 512–525. https://doi.org/10.1109/CSF.2015.38
[43] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016*. 45–60. https://doi.org/10.1109/CSF.2016.11
[44] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components. *CoRR* abs/1510.00697 (2015). http://arxiv.org/abs/1510.00697
[45] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[46] Andrew Kennedy. 2006. Securing the .NET Programming Model. *Theoretical Computer Science* 364, 3 (2006), 311–317. https://doi.org/10.1016/j.tcs.2006.08.014
[47] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*. USENIX Association, 273–284.
[48] Thomas F. Knight, Jr., André DeHon, Andrew Sutherland, Udit Dhawan, Albert Kwon, and Sumit Ray. 2012. SAFE ISA (version 3.0 with Interrupts per Thread). http://ic.ese.upenn.edu/distributions/safe_processor/
[49] Robbert Krebbers. 2015. *The C Standard Formalized in Coq*. Ph.D. Dissertation. Radboud University Nijmegen.
[50] Joshua Kroll, Gordon Stewart, and Andrew Appel. 2014. Portable Software Fault Isolation. In *27th IEEE Computer Security Foundations Symposium*. IEEE.
[51] Orna Kupferman and Moshe Y. Vardi. 1999. Robust Satisfaction. In *CONCUR '99: 10th International Conference on Concurrency Theory*. 383–398. https://doi.org/10.1007/3-540-48320-9_27
[52] Leslie Lamport and Fred B. Schneider. 1984. Formal Foundation for Specification and Verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*. 203–285. https://doi.org/10.1007/3-540-15216-4_15
[53] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176
[54] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior #1/3. LLVM Project Blog. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
[55] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. 633–647. https://doi.org/10.1145/3062341.3062343
[56] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814
[57] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-Like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0
[58] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 395–404. https://doi.org/10.1145/2254064.2254111
[59] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. 448–461. https://doi.org/10.1145/2908080.2908109
[60] Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *34th IEEE Symposium on Security and Privacy*. IEEE, 415–429.
[61] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation Via Universal Embedding. In *21st ACM SIGPLAN International Conference on Functional Programming, ICFP*. 103–116. https://doi.org/10.1145/2951913.2951941
[62] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP) (LNCS)*, Vol. 9236. Springer, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22
[63] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems* (2015).
[64] Marco Patrignani and Dave Clarke. 2015. Fully Abstract Trace Semantics for Protected Module Architectures. *Computer Languages, Systems & Structures* 42 (2015), 22–45. https://doi.org/10.1016/j.cl.2015.03.002
[65] Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully-Abstract Compilation. In *29th IEEE Computer Security Foundations Symposium*. IEEE.
[66] Marco Patrignani and Deepak Garg. 2018. Robustly Safe Compilation. *CoRR* abs/1804.00489 (2018). arXiv:1804.00489 http://arxiv.org/abs/1804.00489
[67] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *12th USENIX Security Symposium*. USENIX Association.
[68] John Regehr. 2010. A Guide to Undefined Behavior in C and C++, Part 3. Embedded in Academia blog. https://blog.regehr.org/archives/232
[69] Charles Reis and Steven D. Gribble. 2009. Isolating Web Programs in Modern Browser Architectures. In *EuroSys Conference*. ACM, 219–232. https://doi.org/10.1145/1519065.1519090
[70] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities. 2nd Workshop on Principles of Secure Compilation (PriSC).
[71] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management. In *27th European Symposium on Programming, ESOP*. 475–501. https://doi.org/10.1007/978-3-319-89884-1_17
[72] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *PACMPL* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913
[73] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 48–62. https://doi.org/10.1109/SP.2013.13
[74] Gang Tan. 2017. Principles and Implementation Techniques of Software-Based Fault Isolation. *Foundations and Trends in Privacy and Security* 1, 3 (2017), 137–198. https://doi.org/10.1561/3300000013
[75] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards Automatic Compartmentalization of C Programs on Capability Machines. In *Workshop on Foundations of Computer Security (FCS)*. 1–14.
[76] Neline van Ginkel, Raoul Strackx, Jan Tobias Muehlberg, and Frank Piessens. 2016. Towards Safe Enclaves. In *4th Workshop on Hot Issues in Security Principles and Trust (HotSpot)*.
[77] Thomas Van Strydonck, Dominique Devriese, and Frank Piessens. 2018. Linear Capabilities for Modular Fully-Abstract Compilation of Verified Code. 2nd Workshop on Principles of Secure Compilation (PriSC).
[78] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. To appear at NDSS.
[79] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 203–216.
[80] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles*. ACM, 260–275. https://doi.org/10.1145/2517349.2522728
[81] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE, 20–37. https://doi.org/10.1109/SP.2015.9
[82] Pierre Wilke, Frédéric Besson, Sandrine Blazy, and Alexandre Dang. 2017. CompCert for Software Fault Isolation. Secure Compilation Meeting (SCM).
[83] Thomas Y. C. Woo and Simon S. Lam. 1993. A Semantic Model for Authentication Protocols. In *1993 IEEE Symposium on Research in Security and Privacy*. IEEE, 178–194. https://doi.org/10.1109/RISP.1993.287633
[84] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Commun. ACM* 53, 1 (2010), 91–99. https://doi.org/10.1145/1629175.1629203
[85] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: Fully Verified Software Fault Isolation. In *11th International Conference on Embedded Software*. ACM, 289–298. https://doi.org/10.1145/2038642.2038687