Portland State University

PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

2-2018

When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise

Guglielmo Fachini
Inria Paris

Cătălin Hriţcu
Inria Paris

Marco Stronati
Portland State University

Arthur Azevedo de Amorim
Carnegie Mellon University

Carmine Abate
Inria Paris

See next page for additional authors

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

 Part of the Information Security Commons, and the Programming Languages and Compilers Commons

Let us know how access to this document benefits you.

Citation Details

Fachini, Guglielmo, Catalin Hritcu, Marco Stronati, Arthur Azevedo de Amorim, Ana Nora Evans, Carmine Abate, Roberto Blanco, Théo Laurent, Benjamin C. Pierce, and Andrew Tolmach. "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise." arXiv preprint arXiv:1802.00588 (2018).

This Pre-Print is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Authors

Guglielmo Fachini, Cătălin Hriţcu, Marco Stronati, Arthur Azevedo de Amorim, Carmine Abate, Roberto Blanco, Théo Laurent, Benjamin C. Pierce, and Andrew Tolmach

This pre-print is available at PDXScholar: https://pdxscholar.library.pdx.edu/compsci_fac/183

When Good Components Go Bad

Formally Secure Compilation Despite Dynamic Compromise

Guglielmo Fachini[1] Cătălin Hriţcu[1] Marco Stronati[1] Arthur Azevedo de Amorim[2] Ana Nora Evans[1,3]
Carmine Abate[1,4] Roberto Blanco[1] Théo Laurent[1,5] Benjamin C. Pierce[6] Andrew Tolmach[7]

[1]Inria Paris [2]Carnegie Mellon University [3]University of Virginia [4]University of Trento
[5]ENS Paris [6]University of Pennsylvania [7]Portland State University

arXiv:1802.00588v1 [cs.CR] 2 Feb 2018

***Abstract*—We propose a new formal criterion for secure compilation, giving strong end-to-end security guarantees for software components written in unsafe, low-level languages with C-style undefined behavior. Our criterion is the first to model *dynamic* compromise in a system of mutually distrustful components running with least privilege. Each component is protected from all the others—in particular, from components that have encountered undefined behavior and become compromised. Each component receives secure compilation guarantees up to the point when it becomes compromised, after which an attacker can take complete control over the component and use any of its privileges to attack the remaining uncompromised components. More precisely, we ensure that dynamically compromised components cannot 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 language.**

To illustrate this model, we build a secure compilation chain for an unsafe language with buffers, procedures, and components. We compile it to a simple RISC abstract machine with built-in compartmentalization and provide thorough proofs, many of them machine-checked in Coq, showing that the 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.

1 Introduction

Compartmentalization offers a strong, practical defense against a range of devastating low-level attacks, such as control-flow hijacking exploiting buffer and integer overflow vulnerabilities in type and memory unsafe languages such as C and C++ [17], [34], [81]. A variety of compartmentalization technologies are widely deployed, including process-level privilege separation [17], [34], [47] (used in OpenSSH [65] and for sandboxing plugins and tabs in web browsers [67]), software fault isolation [73], [78] (e.g., Google Native Client [84]), modules in WebAssembly [36] (in modern web browsers), and hardware enclaves (e.g., Intel SGX [39]); many more are on the drawing boards [13], [19], [70], [81]. These mechanisms are a good basis for building more secure compilation chains that mitigate low-level attacks [30], [34], [44], [64], [75]–[77]. In particular, compartmentalization can be applied in unsafe low-level languages like C and C++ to structure large, performance-critical applications into mutually distrustful components that run with minimal privileges and interact only via well-defined interfaces.

Intuitively, protecting each component from all the others should have strong security benefits, since a vulnerability in one component need not compromise the security of the whole application. Instead, each component should be protected from all the other components until it becomes compromised by an exploit of one of its vulnerabilities, causing it to attack the remaining uncompromised components. The goal of this paper is to formalize this dynamic compromise intuition by precisely characterizing what it means for a compilation chain to be secure in this setting.

We want a characterization that supports *source-level security reasoning*, so that programmers can reason soundly about the security of their code without thinking about the complex details of the whole compilation chain (compiler, linker, loader, runtime system, system software, etc). What makes this particularly challenging for C and C++ is that their semantics, as described in standards documents and as implemented by compilers, call out a large number of "undefined behaviors" that have no source-level meaning whatsoever and are simply *assumed* never to occur. On programs that do have undefined behavior, standards-compliant compilers for these languages are allowed to generate code that does literally anything—in particular, anything a remote attacker may want. Compilers aggressively exploit the assumption of no undefined behavior to produce the fastest possible code for well-defined programs, often leading to exploitable vulnerabilities when the assumption is broken [38], [72]—and to serious confusion even among experienced C/C++ developers, who generally expect saner behavior [37], [53], [56], [68], [79], [80]. To obtain strong security guarantees, we make a worst-case assumption that *any* undefined behavior can lead to compromise.

The purpose of a compartmentalizing compilation chain is to ensure that the arbitrary effects of undefined behavior are *limited* to the component in which it occurs. For a start, we 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 high-level language (e.g., procedure calls and returns). Perhaps unsurprisingly, to support a model of dynamic compromise, in which a component receives full guarantees until it encounters undefined behavior, we also have to restrict the *temporal* scope of undefined behavior.

We do so by requiring that compiler optimizations do not cause undefined behavior to happen before earlier observable events (e.g., system calls). This additional restriction is easier to enforce than the spatial one described above—indeed, the CompCert verified C compiler [54] provably satisfies it, giving its users a saner model of undefined behavior than other C compilers [66].

We also want a characterization that is *formal*—that clarifies and brings mathematical precision to the security guarantees and attacker model of compartmentalizing compilation. In particular, we want a characterization of *sound* source-level reasoning principles that can be used to assess the security of compartmentalized applications using either formal verification tools or manual security audits. A formal characterization can further serve as a specification for formally verifying the correctness of secure compilation chains, and as useful guidance for designing and building unverified ones.

Our correctness criterion improves on existing ones in three respects. First, unlike most criteria for formally secure compilation [2], [4]–[7], [28], [29], [52], [62], it applies to *compartmentalized* programs, rather than being phrased in terms of protecting a single trusted program from an untrusted context. Second, unlike some recent characterizations that do consider modular protection [23], [64], it applies to *unsafe* languages with undefined behaviors at the source level. And third, it considers a *dynamic* compromise model—a significant advance over the proposal of Juglaret *et al.* [43], which does consider mutually distrustful components written in unsafe languages, but which only supports a static compromise model where components get no security guarantees whatsoever if they can encounter undefined behavior in *any* context.

The limitation to static compromise scenarios actually seems inherent to previous techniques, which are all based on the formal criterion of *full abstraction* [2]. We support dynamic compromise by taking the somewhat radical step of dropping full abstraction and instead phrasing security in terms of preserving safety properties [50] in adversarial contexts [29]. Moving away from full abstraction also makes our criterion easier (and more efficient) to achieve in practice and to prove at scale.

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

Our second contribution is to formalize *RSCC* (§3). We start from *Robustly Safe Compilation* (*RSC*, §3.1) a simple security criterion recently introduced by Garg *et al.* [29], and incrementally extend this to dynamic compromise (RSC^{DC}, §3.2) and mutually distrustful components (RSC^{DC}_{MD}, §3.3), which we use as a base for defining *RSCC* (§3.4). We also propose an effective and general proof technique for RSC^{DC} (§A). First, we show that *RSCC*—the most direct incarnation of our intuitive attacker model—follows from the simper RSC^{DC}_{MD}. We then reduce RSC^{DC}_{MD} to: *(1)* constructing, from any target-level finite prefix of a trace of cross-component calls and returns, a *whole* source-level program that will produce that prefix; *(2)* using standard simulation proofs to show trace decomposition and composition lemmas that relate our semantics for whole programs to generically constructed semantics that work on partial programs; and *(3)* using a whole-program compiler correctness proof à la CompCert [54] as a black-box for moving back and forth between the source and target languages. This novel proof technique yields significantly simpler and more scalable proofs than previous work in this space [43], [64] (as explained in detail in §6 and §A).

Our third contribution is a proof-of-concept secure compilation chain for an unsafe language featuring buffers, procedures, components, and a CompCert-like block-based memory model [55]. Our entire compilation chain is implemented in the Coq proof assistant. The first part of the chain compiles our source language to a simple RISC abstract machine with built-in compartmentalization (§4). We use our proof technique to construct careful proofs–many of them machine-checked in Coq—showing that this compiler satisfies *RSCC* (§B). 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, §5.1) or a tag-based reference monitor (§5.2). Both back ends are implemented in Coq. Neither has yet been verified, but we have used property-based testing to gain confidence that the SFI back end satisfies the invariants of the compartmentalized machine.

We close by discussing related work (§6) and future directions (§7). The appendices describe our general proof technique and other details that had to be omitted for space. Our Coq development (around 20,000LOC) is available as supplemental material at https://github.com/secure-compilation

2 *RSCC* By Example

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

We need not be very precise, here, about the details of the source language; we just assume that it is equipped with some compartmentalization facility [35] that allows programmers to break up security-critical applications into mutually distrustful *components* that run with minimal privileges and can only interact via well-defined *interfaces*. We also 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 our secure compilation criteria in §3 are quite generic: interfaces can include any information 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 files). We assume that the division of an application into components and the interfaces of those components are statically determined and fixed throughout execution. For the illustrative language of §4, we will use a simple and rather rigid notion of components and interfaces, where components don't directly share state, interfaces just list the procedures that components provide and those that they expect their environment to provide, and the only thing one component can do to another one is to call procedures allowed by the interfaces of both components.

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 security criteria do not fix a specific mechanism for achieving this: 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 §5 we study a compilation chain whose back ends use inline and 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 trick them into misusing their own privileges (i.e., confused deputy attacks) and to compromise them as well—e.g., by sending them malformed inputs that trigger control-hijacking attacks.

In the examples below, 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 value non-deterministically [54]. 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.

In practice, it is generally unrealistic to assume that we know in advance which components will be compromised and which ones will not. This motivates our novel model of *dynamic compromise* in which each component receives full guarantees until it becomes compromised by encountering an undefined behavior, causing it to start attacking the remaining uncompromised components. In practical attacks, these compromises are ultimately caused by malicious inputs fed to the application.

This model allows developers to reason informally about various dynamic compromise scenarios and their impact on the security of the whole application [34]. If the practical consequences of some plausible dynamic compromise scenario are too serious, developers can further reduce or separate privilege by narrowing interfaces or splitting components, or they can make components more defensive by dynamically validating the inputs they receive from other components.

As a first running example, consider the pseudocode of the simple, idealized application from Figure 1. It defines three interacting components $\texttt{C}_0$, $\texttt{C}_1$, and $\texttt{C}_2$ that use the environment `E` for input (`E.read`) and output (`E.write`). Component $\texttt{C}_1$ defines the `main`() procedure, which first invokes $\texttt{C}_2$`.init`(), then reads a request `x` from the environment (e.g., from some remote client), parses it using a private procedure (omitted here) to obtain `y`, and then invokes $\texttt{C}_2$`.process(x,y)`. This procedure then calls $\texttt{C}_2$`.prepare`() and $\texttt{C}_2$`.handle(y)`, obtaining some `data` that it validates using $\texttt{C}_0$`.valid`, and if this succeeds writes the `data` together with the original request `x` to the environment (e.g., to disk, to a database, etc).

Suppose we would like to establish two properties:

(S_1) the application only writes valid data (i.e. data for which $\texttt{C}_0$`.valid` returns true); and

(S_2) any call `E.write(<data,x>)` happens as a response to a previous `E.read`() call by $\texttt{C}_1$ obtaining the request `x`.

These can be shown to hold of executions that do not exhibit undefined behavior simply by analyzing the control flow. But what if undefined behavior does occur? Suppose that we can rule out this possibility—by careful inspection, testing, or formal verification—for simple parts of the code, but are still unsure about three subroutines:

(V_1) $\texttt{C}_1$`.parse(x)` performs complex array computations, and we do not know if it is immune to buffer overflows on all inputs `x`.

(V_2) $\texttt{C}_2$`.prepare`() is intended to be called only if $\texttt{C}_2$`.init`() has been called beforehand to set up a shared data structure; otherwise, it might dereference a pointer with an undefined value.

(V_3) $\texttt{C}_2$`.handle(y)` might integer overflow on some inputs `y`.

If the undefined behavior in V_1 occurs, then $\texttt{C}_1$ can get compromised and call $\texttt{C}_2$`.process(x,y)` with values of `x` that it hasn't received from the environment, thus invalidating S_2. 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_1 remains true.

Now consider the possible undefined behavior from V_2. If $\texttt{C}_1$ is not compromised, this undefined behavior cannot occur, since $\texttt{C}_2$`.init`() will be called before $\texttt{C}_2$`.prepare`(). Moreover, this undefined behavior cannot occur even if $\texttt{C}_1$ *is* compromised by encountering the undefined behavior in V_1, because that can only occur *after* $\texttt{C}_2$`.init`() has been called. Hence V_1 and V_2 together are no worse than V_1 alone, and property S_1 remains true. Inferring this property crucially depends on our model of dynamic compromise, in which $\texttt{C}_1$ can be treated as honest and gets full guarantees until it encounters undefined behavior. If instead we were only allowed to reason about $\texttt{C}_1$'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 $\texttt{C}_2$ cannot be compromised: an arbitrary component with the same interface as $\texttt{C}_1$ could indeed compromise $\texttt{C}_2$ by calling $\texttt{C}_2$`.process` before $\texttt{C}_2$`.init`.

Finally, if the execution encounters the undefined behavior in V_3, then $\texttt{C}_2$ can get compromised, irrespective of whether $\texttt{C}_1$ is compromised beforehand or not. The compromise of $\texttt{C}_2$ invalidates both S_1 and S_2.

Even if still informal for now, this security analysis already identifies $\texttt{C}_2$ as a single point of failure for both properties of our system. There are many ways the developers of this application could go about improving this situation: They

```
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 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 UNDEF if not initialized
    data := C2.handle(y); //(V3) can UNDEF for some y
    if C0.valid(data) then E.write(<data,x>)
  }
  prepare() { ... }
  handle(y) { ... }
}
```

Figure 1: Pseudocode of application broken into components

```
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 UNDEF for some x
}
component C2 {
  export init, process;
  init() { ... }
  process(y) {
    C2.prepare();        //(V2) can UNDEF if not initialized
    data := C2.handle(y);//(V3) can UNDEF for some y
  }
  prepare() { ... }
  handle(y) { ... }
}
```

Figure 2: More secure variant of our application

could improve the code in C2.handle to reduce the chances of encountering undefined behavior, e.g. by doing better input validation. They could also make C1 check the values it sends into C2.process, so that an attacker would have to compromise both C1 and C2 to break the validity of writes. For ensuring the correspondence of reads and writes despite the compromise of C1 they could make C2 also read the request values directly from E, instead of only via C1.

To achieve the best security though, the read and write privileges can be delegated to C0, which performs no complex data processing of its own and thus is a lot less likely to be compromised by undefined behavior. In this new variant of our application (Figure 2), component C0 reads a request, calls C1.parse on this request, passes the parse to C2.process, validates the data C2 returns and then writes it out. This way both our desired properties hold even if both C1 and C2 are compromised, since now the core application logic and privileges have been completely separated from the dangerous data processing operations that could cause vulnerabilities. Since such a neat privilege separation is not always possible in practice though, we want a security criterion that can reason about all kinds of component partitionings.

The first step towards making this more formal 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 calling a procedure from another component and returning to another component. 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 of the program then it must better be the case that:

(S_1) E.read was called before in the trace and returned x;
(S_2) C0.valid(data) was called before and returned true.

The application variant from Figure 2 achieves these strong properties despite the dynamic compromise of both C1 via V_1 and C2 via V_3, but for the variant from 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 returned x before *or* E.read returned x' before that can cause undefined behavior in C1.parse(x') *or* C2.process(x,y) was called before with a y that can cause undefined behavior in C2.handle(y);
(W_2) C0.valid(data) was called before and returned true *or* C2.process(x,y) was called before with a y that can cause undefined behavior in C2.handle(y).

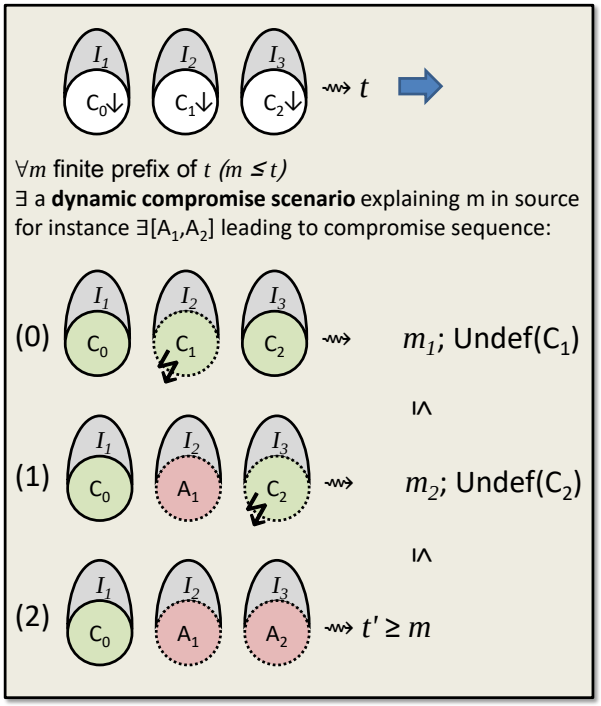
While these properties are much weaker, they are still not trivial and require an attacker to actually find and send the inputs that break the guarantees of the application by compromising C1 or C2.

Properties S_1, S_2, W_1 and W_2 are all *safety properties* [50], in this case inspired by the "correspondence assertions" used to specify authenticity in security protocols [31], [83]. Intuitively, a trace property is a safety property if it can be invalidated by a finite trace prefix and once it is invalidated by such a "bad prefix" it can no longer be restored. For instance here is a bad prefix for S_2 that includes a call to E.write(<data,x>) with no preceding call to C0.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>)]
```

While the program from Figure 2 cannot produce traces with this bad prefix, it could if we were to remove the validity check in C0.main(), which would invalidate safety property S_2.

Compiler correctness is most often phrased in terms of preserving trace properties in general [54], and thus safety properties as a very important special case. Compiler correctness stops applying, however, as soon the program has an undefined behavior, and all security guarantees are lost globally. Instead, we want our secure compiler to enforce that dynamically compromised components are not able to break the safety properties of the system at the target level any



Where prefixes m, m_1, m_2 could 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 running 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.

more than equally privileged components without undefined behavior already could in the source language.

We phrase *Robustly Safe Compartmentalizing Compilation* (*RSCC*) in terms of a security game that we illustrate in Figure 3 for our running example. With an *RSCC* compilation chain, given any execution of the compiled and linked components $\texttt{C}_0\downarrow$, $\texttt{C}_1\downarrow$ and, $\texttt{C}_2\downarrow$ producing trace t in the target language, we can explain any finite (bad) 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. As done by CompCert [54], we model undefined behavior in our source language as a special $\texttt{Undef}(\texttt{C}_i)$ event terminating the trace, and whatever happens afterwards at the target level for the component $\texttt{C}_i$ that encountered the undefined behavior can no longer be explained in terms of its source code. For instance, in step 0 of Figure 3 component $\texttt{C}_1$ is the first to encounter undefined behavior after producing a prefix m_1 of m.

How can we explain the rest of m in the source language? Our solution in *RSCC* is to require that one can replace $\texttt{C}_1$, the component that encountered undefined behavior, with some other source component $\texttt{A}_1$ 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 $\texttt{C}_1$ with $\texttt{A}_1$ we have to go back in time and re-execute the program from the beginning obtaining a longer trace, in this case m_2; $\texttt{Undef}(\texttt{C}_2)$ (where we write ; for appending the element $\texttt{E.write}(<\texttt{data},\texttt{x}>)$ to m). 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 trace m. In the example dynamic compromise scenario from Figure 3, this means replacing $\texttt{C}_1$ with $\texttt{A}_1$ and $\texttt{C}_2$ with $\texttt{A}_2$, after which the program can produce the whole prefix m in the source language.

Let's now use this *RSCC* security game to deduce that in our example from Figure 2, even compromising both $\texttt{C}_1$ and $\texttt{C}_2$ does not break property S_2 at the target level. Assume by contradiction that a trace of our compiled program breaks property S_2. Then there exists a minimal finite prefix "m; $\texttt{E.write}(<\texttt{data},\texttt{x}>)$" such that $\texttt{C}_0\texttt{.valid(data)}$ does not appear in m. Using *RSCC* we obtain that there exists a dynamic compromise scenario explaining m in the source. The most interesting case is when this scenario involves the compromise of both $\texttt{C}_1$ and $\texttt{C}_2$ as in Figure 3. In this case, replacing $\texttt{C}_1$ and $\texttt{C}_2$ with arbitrary $\texttt{A}_1$ and $\texttt{A}_2$ with the same interfaces allows us to reproduce the whole bad prefix m in the source (step 2). We can now reason in the source, either informally or using a program logic for robust safety [71], that this cannot happen, since the source code of $\texttt{C}_0$ does call $\texttt{C}_0\texttt{.valid(data)}$ and only if it gets `true` back does it call $\texttt{E.write}(<\texttt{data},\texttt{x}>)$.

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 becomes 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 $\texttt{C}_2$ receives a `y` that exploits $\texttt{C}_2\texttt{.handle(y)}$ (vulnerability V_3). However, as discussed earlier a compromised $\texttt{C}_1$ could, at least in theory, try to compromise $\texttt{C}_2$ by calling $\texttt{C}_2\texttt{.process}$ without proper initialization (exploiting vulnerability V_2). Showing that this cannot actually happen requires us to use step 0 of the game from Figure 3, which gives us that the original compiled program obtained by linking $\texttt{C}_0\downarrow$, $\texttt{C}_1\downarrow$ and, $\texttt{C}_2\downarrow$ can produce the trace m_1; $\texttt{Undef}(\texttt{C}_1)$, for some prefix m_1 of the bad trace prefix in which $\texttt{C}_2\texttt{.process}$ is called without calling $\texttt{C}_2\texttt{.init}$ first. However, we can easily convince ourselves (or formally verify) that the straight-line code of the $\texttt{C}_1\texttt{.main}()$ procedure can only cause undefined behavior *after* it has called $\texttt{C}_2\texttt{.init}$, which leads to a contradiction with the existence of a bad trace exploiting V_2.

3 Formally Defining *RSCC*

In the previous section we introduced the intuition behind *RSCC*, our novel criterion for secure compilation of components written in unsafe, low-level languages. We now step back and provide a formal definition for *RSCC*.

Our starting point is a simple notion of *Robustly Safe Compilation* (*RSC*) (§3.1), which we first extend to *unsafe languages with C-style undefined behavior*, providing a novel model of *dynamic compromise* in which a partial program P receives secure compilation guarantees until P encounters undefined behavior (§3.2). We then further extend this to protecting a set of *mutually distrustful components running with least privilege* from their untrusted context, obtaining a new property we call RSC_{MD}^{DC} (§3.3). Using these ideas, we formalize *RSCC* (§3.4), which (compared to RSC_{MD}^{DC}) trades off some simplicity for directly capturing the informal dynamic compromise game from §2.

While the definitions in this section are general, and could thus apply in a variety of settings, the next section will illustrate their instantiation to our concrete compilation chain.

3.1 *RSC*: Robustly Safe Compilation

We start from *RSC*, a criterion for secure compilation recently proposed by Garg *et al.* [29], which is equivalent to the preservation of all safety properties against adversarial low-level contexts, i.e., preservation of *robust safety* [32], [49], [71]. We focus on preserving robust safety since it captures many important properties of programs (e.g., robust partial correctness), while also allowing for a simple secure compilation proof technique (see Definability in §A.2). Safety properties [50] are often stated in terms of potentially infinite traces built over events such as inputs from and outputs to the environment [54]. We write $P{\rightsquigarrow}t$ to mean that the complete program P can produce trace t with respect to some operational semantics. Armed with this, *RSC* can be stated as a property of a whole compilation chain: the source language and its trace-based big-step operational semantics ($P{\rightsquigarrow}t$), compiler ($P{\downarrow}$), source and target linkers ($C_S[P]$ and $C_T[P_T]$), and target-level semantics ($P_T{\rightsquigarrow}t$) including for instance the loader, target machine, and deployed protection mechanisms:

Definition 3.1. A compilation chain provides *RSC* iff

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

For any partial source program P and any (intuitively adversarial) target context C_T where C_T linked with the compilation of P can produce a trace t in the target language ($C_T[P{\downarrow}]{\rightsquigarrow}t$), and for any (bad) finite prefix m of trace t (written $m \leq t$) we can construct a(n adversarial) 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, any finite attack m that 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 one to 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 [29].

As is the case for CompCert and our simple compiler from §4, we assume for simplicity that the traces are exactly the same in the source and target languages. However, it would be easy to extend the formal development from this section 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 protecting partial programs written in an *unsafe* source language, with C-style undefined behavior. As explained in §2, we model undefined behavior in the source language 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 undefined behaviors in the source language can be attributed to the part of the program that causes them via the $\texttt{Undef}(P)$ and $\texttt{Undef}(C)$ events (while in §3.3 we will blame the component encountering undefined behavior).

Definition 3.2. A compilation chain provides *Robustly Safe Compilation with Dynamic Compromise* (RSC^{DC}) iff

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

Instead of always requiring as in *RSC* 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., $t' {\prec_P} m \triangleq \exists m'{\leq}m.\ t'{=}(m';\texttt{Undef}(P))$). To facilitate source-level reasoning we do not allow contexts C_S to encounter undefined behaviors. However, even such a well-behaved context can sometimes trigger an undefined behavior in the protected program P, in which case there is no way to keep providing guarantees to P going forward. P does nevertheless receive secure compilation guarantees until the last trace event before the undefined behavior.

Like in CompCert [54], [66], we treat undefined behaviors as *observable* events in the execution trace, which allows the compiler to perform optimizations that move undefined behaviors earlier in the execution order past any operations that do not cause events, but prevents the compiler from moving undefined behaviors before earlier observable events. While some C compilers would need to be adapted to respect this discipline [66], limiting the temporal scope of undefined behavior is a strong 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 most cases.

Since *RSC* corresponds exactly to preserving robust safety properties [29], one might wonder what properties RSC^{DC}

preserves. We proved that *RSC*^{DC} corresponds exactly to preserving the following class Z_P against an adversarial context:

Definition 3.3. $Z_P \triangleq \mathit{Safety} \cap \mathit{Closed}_{\prec_P}$, where

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

The class of properties Z_P is defined as the intersection of *Safety* and a new class $\mathit{Closed}_{\prec_P}$ of properties closed under extension of traces with undefined behavior in P. 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., $\exists m.\ t{=}(m;\texttt{Undef}(P))$—then π should also allow any extension of the trace m—i.e., any trace t' that has m as a prefix. 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 $m;\texttt{Undef}(P)$. The intuition is simple: the secure compiler 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.

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:

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

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:

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

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 prove an equivalent *RSC*^{DC} characterization:

Theorem 3.4.
$$\textit{RSC}^{DC} \iff \left(\begin{array}{r}\forall P\ \pi{\in}Z_P.\ (\forall C_S\ t.\ C_S[P]{\rightsquigarrow}t \Rightarrow t{\in}\pi) \\ \Rightarrow (\forall C_T\ t.\ C_T[P{\downarrow}]{\rightsquigarrow}t \Rightarrow t{\in}\pi)\end{array}\right)$$

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, as was the case above, the robust safety property can explicitly talk only about the events of the program, not the ones of the context. Moreover, once we add interfaces in §3.3 we will be 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:

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

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:

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

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

RSC^{DC} provides us with a novel model of *dynamic compromise* for secure compilation, but is still phrased in terms of protecting a trusted partial program from its untrusted context. We now extend this model to one protecting any set of *mutually distrustful components running with least privilege* from their 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 and linking a program with a context to be 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* and components can only interact in accordance to 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 argued by Juglaret *et al.*, the whole idea of least privilege design crucially relies on the fact that even if a component is compromised, it does not immediately get more privilege.

Definition 3.5. 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, so that for any compatible interfaces I_P and I_C we have

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

This *RSC*$^{DC}_{MD}$ definition closely follows *RSC*^{DC}, but restricts programs and contexts to two compatible interfaces I_P and I_C. The source-level context is obtained by applying the back-translation function $\uparrow$ pointwise to all the interfaces in I_C.

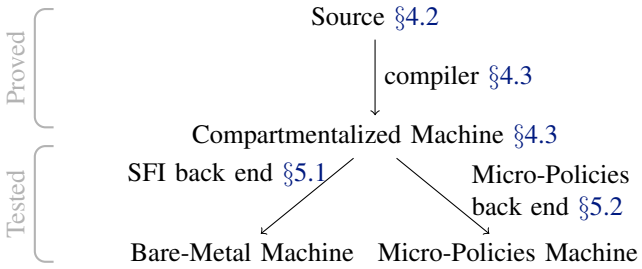


Figure 4: Our secure compilation chain

3.4 Formalizing *RSCC*

Using these ideas, we define *RSCC* as a direct formalization of the dynamic compromise game previously 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 any component that encounters undefined behavior in the source with an arbitrary safe component that may now attack the remaining components. Formally, this is captured by the following property:

Definition 3.6. A compilation chain provides *Robustly Safe Compartmentalizing Compilation (RSCC)* iff for any compatible interfaces $I_1, ..., I_n$:

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

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

This formal definition allows us to play an iterative game in which components that encounter undefined behavior become compromised and attack the remaining uncompromised 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 §6).

4 Securely Compiling from a Simple C-like Language to a Compartmentalized Machine

We now start describing the compilation chain that we have developed in Coq to illustrate *RSCC* and that is depicted in Figure 4. The *source* language is a simple unsafe imperative language with buffers, procedures, and components that is first compiled to a variant of RISC assembly with compartmentalized memory and protected call stack, our *compartmentalized machine*. The compilation chain continues with one of two back ends that implement two different security enforcement mechanisms on two slightly different RISC machines. The *SFI Back End* targets a *Bare-Metal Machine* that implements a simple architecture without any protection mechanisms. Instead, the code generated by the SFI back end is instrumented to enforce compartmentalization. The *Micro-Policies Back End* targets a *Micro-Policies Machine* that provides hardware acceleration for tag-based reference monitors. In this section we describe in detail the compiler while the back ends are discussed in the next section §5. Despite the simplifications needed to manage the complexity of the secure compilation proof (§A and §B), we believe that this design can be scaled to a realistic compiler such as CompCert in the future.

We first introduce the definitions that are common to both source and compartmentalized machine, such as components and the block-based memory model. We then describe the details of the two languages and then the compiler. We have proved that the compiler provides *RSCC* (§B) using a generic proof technique (§A). The confidence in the security of our system comes from the fact the both the implementation of the compiler and most of the proofs are mechanized in Coq.

4.1 Common Notions

We now illustrate the common infrastructure for the source and compartmentalized machine, in particular they share the same notion of program, values, and memory model.

Programs and Interfaces A program is a triplet `(Is, procs, bufs)` composed of an interface, a set of procedures and a set of static buffers. Interfaces `Is` contain the names of the procedures that the component *exports* to and *imports* from other components. `procs` contains all the code of the component organized in procedures. `bufs` are statically allocated buffers, during execution more buffers can be allocated. We further assume that code cannot be modified during runtime. The steps for executing a linked program consist for each component in: checking that exports are matched with a procedure in its `procs`, checking that all imports are satisfied by some linked component and finally populating the memory with the static `bufs`.

Block-Based Memory Model The memory model for both source and compartmentalized machine is the same and it is a slightly simplified version of the one used in CompCert [55]. Each component has an infinite memory that is composed of finite blocks, each block being an array of values. This is reflected in the structure of pointers which are composed of three elements (C, b, o): the identifier of the component which owns the block, the unique block identifier and the offset inside the block. The system provides a special operation `alloc` for obtaining fresh blocks; this operation never fails. For simplicity there is no `free` operation, but we could add

it in the future. Pointers are unforgeable capabilities and can only be produced by `alloc`. For now our components are not allowed to exchange pointers, as a result components cannot access each others' memories at this level. In both languages arithmetic operations on pointers are limited to increasing and decreasing the offset, equality and comparison. Pointers cannot be cast to or from integers and derefencing an integer is undefined behavior. This very abstract memory model is mapped to a more realistic flat address space in the back ends.

Values The languages manipulate mathematical integers, pointers and an additional undefined value $\top$.

Definition 4.1 (Values). $v ::= n \mid (C, b, o) \mid \top$

The undefined value $\top$ is obtained when reading from an uninitialized piece of memory, as result of an erroneous binary operation or when reading a register in the compartmentalized machine after obtaining control from another component.

Events Following CompCert, we use a labeled operational semantics where events include all interactions of the program with the external world, such as system calls. We introduce extra events to keep track of any interaction between components and in particular any passing of control from one component to another. Every call to an exported procedure produces a visible event `C Call P(n) C'`, where the caller C calls procedure P of component C' passing argument n. All other computations, including calls to non-exported procedures, are invisible in the program trace and result in *silent* steps in the operational semantics. We use the same events for our source language and the compartmentalized machine.

4.2 Source Language

Our source is a simple unsafe imperative language with buffers, procedures, and components. Memory is manually managed and, like in C, out of bounds accesses lead to undefined behavior. An attacker can exploit these undefined behaviors and perform attacks lower in the compilation chain. The language is expression-based and the syntax of expressions is given in Figure 5. 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. Some aspects of the language are simplified: for instance, each component has a single static buffer `local` that is also used for the global variables of the component. More buffers can be created dynamically with `alloc`.

4.3 The Compartmentalized Machine

Our design goal for the compartmentalized machine language was to be as low-level as possible while still allowing us to target the two back ends presented later in §5. The resulting language showed in Figure 7 is a simple RISC assembly with two main abstractions: the block-based memory model and support for cross-component calls. The memory model remains the same as for the source leaving the back ends complete freedom in their layout of blocks. One important difference compared to the source is the presence of registers

`e ::=`	`v`	values
	`| local`	local static buffer
	`| e` $\otimes$ `e`	binary operations
	`| e ; e`	sequence
	`| if e then e else e`	conditional
	`| alloc e`	memory allocation
	`| !e`	dereferencing
	`| e1 := e2`	assignment
	`| C.P(e)`	procedure call
	`| exit`	terminate

Figure 5: Syntax of source language expressions

which are the only shared state between components at this level. In the syntax, l represent labels that are resolved to pointers in the next compilation phase.

Unlike in the source language, there are two kinds of call stacks: an explicit global stack for cross-component calls and an implicit one local to each component for intra-component calls. In addition to the usual `Jal` and `Jump` instructions, used to compile procedure calls and returns private to one component, we introduce two special instructions, `Call` and `Return`, for cross-component calls. These special instructions are the only ones that can manipulate the global call stack. The two back ends will implement this abstract call discipline in different ways using only standard RISC instructions.

The operational semantics rules for `call` and `return` are presented in Figure 6. 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; $reg_\top$ 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. There is a redundancy between the protected stack and `R_RA` precisely because during the `return` the protected frame is used to verify that the register is used correctly; otherwise the program has an undefined behavior. Invalidation of registers follows the intuition that nothing left from another component should be relied upon and forces the compiler to save and restore correctly the registers that are needed.

Compiler Our compiler transforms the source programs to compartmentalized machine instructions in the simplest possible way, to ease reasoning. In particular, it does not perform any optimizations.

4.4 *RSCC* Theorem

Theorem 4.2 (RSCC). This section's compiler satisfies *RSCC*.

5 Implementing Compartmentalized Machine

In this section we describe the second part of our compilation chain, the two back ends that securely enforce the abstractions

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

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

Figure 6: Compartmentalized machine 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 7: Syntax of compartmentalized machine instructions

of the compartmentalized machine against realistic machine-code-level attackers. This involves protecting the integrity of component memories as well as enforcing interfaces and the cross-component call-return discipline.

Our two back ends target variants of a simple RISC machine. In contrast to the abstract block-based memory model from the previous section, at this level memory is a single infinite array addressed via integers, not via unforgeable capabilities. There is no magic `Alloc` instruction; instead, each back end has to position the blocks in the shared address space. Without proper protection, compromised components can access buffers out-of-bounds and overwrite the code or data of other components. Similarly, while the compartmentalized machine had `Call` and `Return` instructions and a protected call stack, at the machine-code level components can jump to arbitrary places in memory, whether they are code or data, and whether they are meant to be jumped to directly or not.

While both of our back ends extend any undefined behavior in the compartmentalized machine to a trace where the machine continues execution in some way that respects high-level abstractions, they achieve this in very different ways. The *SFI back end* (§5.1) targets a *bare-metal machine* that has no protection mechanisms and implements an inline reference monitor purely in software, by rewriting 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* (§5.2), on the other hand, uses a *micro-policies machine* that adds specialized hardware that we program to implement a tag-based reference monitor for compartmentalization. These approaches have complementary advantages: SFI requires no specialized hardware. Micro-policies often incur little overhead [26] and given their simplicity are a good target for formal verification [13]. We hope that other mechanisms, like capability machines [81], could be used to implement the same compartmentalized machine in the future. The main simplification we make here is to use an infinite address space accessed by mathematical integers, but we hope to make this more realistic in the future [58].

5.1 Software Fault Isolation

SFI [78] uses software instrumentation to protect memory regions. The virtual address space is logically split into segments. The virtual address is a pair segment identifier, offset in the segment. Data and code are kept in different segments. Each memory update is preceded by a check of the address' segment identifier against the segment identifier of the data segment. If they match, the store can proceed. A similar scheme is used for the indirect transfer of control within segments. Special mechanisms, such as jump tables, are used for the cross-component communication.

The target machine of the SFI back end is a bare-metal RISC processor with the same instructions as the compartmentalization machine except for `Call`, `Return`, and `Alloc`. The register file contains all the registers from the previous level, and several registers reserved for the SFI instrumentation.

The SFI compiler back end produces machine-code programs that must satisfy the following invariants:

1) a component may not write outside its own data memory
2) a component may transfer control outside its code memory only through exit points allowed by the interface
3) the global, cross-component stack can not be corrupted

The SFI compiler back end uses a special memory layout and instrumentation sequences to realize the desired isolation of components in the produced program. The maximum number of components is statically determined. For now, we assume the output program runs directly on the hardware, without the assistance of an operating system. We do not implement dynamic linking or loading, nor system calls.

The memory is logically divided in equal size blocks, called *slots*. The address is a positive integer with the least significant bits reserved for offset, the next least significant for component identifier, and the rest are slot bits (see Figure 8).

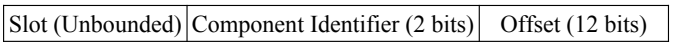


Figure 8: Address Example

The even slots are allocated for code and the odd ones for data. The component zero is reserved for the instrumentation use: code slots for initialization instructions, and data for the cross-component stack. An example with three user components is shown in Figure 9.

Every `Store` $*r_p$ `<-` r_s instruction in the input is replaced by the following instruction sequence [78]:

```
BinOp r_p & DataAnd -> RD
BinOp r_d | DataOr -> RD
Store *RD <- r_s
```

`RD`, `DataAnd`, and `DataOr` are reserved registers. The `RD` register must be set to a valid address in the data memory at every

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 program with 3 user components

component change. The registers `DataAnd` and `DataOr` are shown in Figure 10. The first instruction of the instrumentation copies the offset and slot from the register r_p. The second one sets the last bit of the slot and the component identifier bits. Lastly, the memory update is executed.

The instrumentation of the `Jump` instruction is similar, but uses different registers. The last four bits of the offset are always reset as shown by the values of registers `CodeAnd` and `CodeOr` in Figure 10. All valid `targets` are sixteen memory word aligned by our back end [57].

	Slot		Slot	LSB Slot	CID	Offset		
DataAnd	`1111`	`1111`	`1`	`0`	`00`	`1111`	`1111`	`1111`
CodeAnd	`1111`	`1111`	`1`	`0`	`00`	`1111`	`1111`	`0000`
DataOr	`0000`	`0000`	`0`	`1`	`xx`	`0000`	`0000`	`0000`
CodeOr	`0000`	`0000`	`0`	`0`	`xx`	`0000`	`0000`	`0000`

Figure 10: Masking Registers

The `Call` instruction is translated as a `Jal` (jump and link) instruction followed by a sequence of instructions that restore the values of the reserved registers. The return address is available in the `RA` register at the start of the execution of a procedure. The component-private procedures have already been translated by the previous compiler pass. Every procedure the SFI back end compiles is one that is called externally, and the return address must be first pushed on the stack. To protect from spurious pushes, the first instruction generated is a `Halt` at an aligned address, followed by a sequence of instructions that stores the return address on the cross-component stack, and another one that sets the reserved registers to valid values for the current component. The `Halt` guard and the push on stack sequence are contained in the sixteen-unit block and it is impossible to start execution in the middle. Any attempt to corrupt the stack by pushing a forged address will be thwarted by the `Halt` guard.

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, and it is impossible to prepare a corrupt address in the register, and start executing from the middle of the 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 framework [61] to test our three compartmentalization invariants: (1) no writes outside own memory; (2) no indirect transfer of control outside memory, unless it is at the address stored at the top of the control stack; (3) the cross-component stack is safe. For each property we implemented a test that executes the following steps: (i) generates a syntactically valid intermediate program; (ii) compiles it; (iii) executes the compiler's output in a simulator and records a property-specific trace; (iv) analyzes the trace to verify if the property has been violated. After all the tests passed, we manually injected faults in the compiler by mutating the instrumentation sequences of the generated output. We made sure that our testing finds these injected errors.

5.2 Tag-based Reference Monitor

Our second back end targets a programmable tagged architecture that allows reference monitors called *micro-policies* to be defined in software and accelerated by hardware [13], [25]. On the micro-policy machine each word of data stored in memory or in registers receives a large metadata tag that can reference an arbitrary data structure. On each instruction, the opcode of the instruction and the tags of the arguments, of the instruction, and of 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 software monitor are hardware cached, so if a similar instruction is executed soon enough with the same arguments then the hardware will allow the request immediately, without the overhead of running the software monitor.

This enforcement mechanism is flexible enough to allow implementing a broad range of tag-based reference monitors and for many of them imposes a relatively modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy usage (typically under 40%) and chip area (110%) [25]. Moreover, this mechanism is simple enough so that the security of the reference monitors can be formally verified [11]–[14]. The micro-policy machine targeted by this back end builds on the "symbolic machine" that Azevedo de Amorim *et al.* have defined in Coq and used to prove the correctness and security of several micro-policies [11], [13], [14]. This machine allows for micro-policies to be implemented at a high level of abstraction, using Coq datatypes for the tags and Coq functions for the behavior of the monitor.

The code generation and static linking parts of the micro-policies 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 part of the micro-policies back end is tagging memory in the (static) loader based on metadata from the 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: $t_v ::= \mathit{Ret}(n) \mid \mathit{Any}$. 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 [74]. This is achieved by giving return addresses tags of the form $\mathit{Ret}(n)$, 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: $t_{pc} ::= \mathit{Level}(n)$. Calls increment the counter n, while returns decrement it. A global invariant of the system is that when the stack is at $\mathit{Level}(n)$ there is at most one capability $\mathit{Ret}(m)$ 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 makes the tags follow the values, but for return addresses the linear capability tag $\mathit{Ret}(n)$ 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 $\mathit{Level}(n)$ the machine puts the return address in register `RA` and the monitor gives it tag $\mathit{Ret}(n)$ and it increments the pc tag to $\mathit{Level}(n+1)$. `Jump` is allowed either to the current component or using a $\mathit{Ret}(n)$ capability, but only if we are at $\mathit{Level}(n+1)$; if this is case the pc tag is decremented to $\mathit{Level}(n)$ and the $\mathit{Ret}(n)$ 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.

6 Related Work

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

Abadi [2] and later Kennedy [46] identified failures of full abstraction in the Java and C# compilers. Abadi *et al.* [3] proved full abstraction of a secure channel implementation using cryptography. Ahmed *et al.* [8]–[10], [60] 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.* [28] devised a fully abstract compiler from a subset of ML to JavaScript. More recently, Patrignani *et al.* [52], [62] 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 SGX.

Modular, Fully Abstract Compilation. Patrignani *et al.* [64] 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: instead of trying to restrict the scope of undefined behavior in the source, 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.* [21], [23] proved modular full abstraction by approximate back-translation in Coq for a compiler from simply typed to untyped λ-calculus.

Beyond Good and Evil. The work closest to ours 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, are the following partial programs observationally equivalent?

```
int buf[5]; return buf[42]  ≟  int buf[5]; return buf[43]
```

They both encounter undefined behavior by accessing a buffer out of bounds, so at the source level they cannot be distinguished. However, in a memory unsafe language, the compiled versions of these programs will very likely read (out of bounds) different values from memory and encounter different behaviors. Juglaret *et al.* avoid this problem by imposing a strong limitation on the components for which protection is guaranteed: a set of components is protected only if it *cannot* encounter undefined behavior in *any* context. This amounts to a *static* model of compromise: any component that can possibly be compromised during execution has 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 flexible *dynamic* compromise model. As discussed below, this also makes our security 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 Garg *et al.* [29], who study several secure compilation criteria that are similar to fully abstract compilation, but that are phrased in terms of preserving hyperproperties [20] (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 [49], the analysis of security protocols [32], and compositional verification [71].

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 confidentiality by adopting Garg *et al.*'s [29] robust hypersafety preservation criterion [29].

While *RSC* serves as a solid base for our work, the challenges of protecting unsafe low-level components from each other are unique to our setting, since, like fully abstract compilation, *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. Garg *et al.* [29] observe that, to prove *RSC*, it suffices to back-translate individual finite trace prefixes, but they provide no details on how to carry out such a proof. Our RSC^{DC}_{MD} proof technique from §A is thus also the first proof technique for *RSC*, although further simplifications should be possible with a safe source language. Even without further simplifications, our proof technique is simple and scalable, especially when compared to previous full abstraction proofs. While many proof techniques have been previously investigated [3], [5], [9], [10], [23], [28], [40], [60], fully abstract compilation proofs are notoriously difficult, even for very simple languages, with apparently simple conjectures surviving for decades before being finally settled [22]. 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 [63]. However, in our setting the partial semantics is given a one line generic definition and is related to the complete one by two standard simulation proofs, which is much 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 [57], [85] or tagged hardware [13]. This 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, a step in this direction is underway in ongoing work by Wilke *et al.* [82] on a variant of CompCert with SFI, based on previous work by Kroll *et al.* [48]; we believe that *RSCC* or RSC^{DC} could provide good top-level theorems for such an SFI compiler. In most work on verified compartmentalization [13], [57], [85], communication between low-level compartments is done by jumping to a specified set of entry points; the mode 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 [70]; they plan to simplify their calling convention using a notion of linear return capability [69] that seems similar in spirit to the one used in our micro-policy from §5.2.

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 [15], [16], [27], [33]. This model is also similar to Byzantine behavior in distributed systems [18], [51], 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.

7 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*, *mutual distrust*, and *privilege* 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. Though scaling up to the whole of C will certainly entail further challenges: defining a variant of C with components, efficiently enforcing compartmentalization all the way down, lowering the cost of formal verification, etc.

Verifying Compartmentalized Applications. It would also be interesting to build verification tools based on *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. Promising approaches that one could try to adapt to dynamic compromise include Jia *et al.*'s System M [42], and Devriese *et al.*'s logical relations [24], both of which allow bounding the behavior of a component based on its interface or capabilities.

Dynamic Component Creation. Another interesting extension would be 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 restricts component creation to a special initialization phase [59].

Preserving Confidentiality and Hypersafety. Extending our enforcement mechanisms from preserving robust safety to confidentiality and hypersafety [20], [29] will be challenging, especially so if low-level contexts can observe time.

More Interesting Privilege Restrictions. 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 to more interesting dynamic mechanisms such as history-based access control [1].

Acknowledgments This work is in part supported by ERC Starting Grant SECOMP (715753), by NSF award 1513854, *Micro-Policies: A Framework for Tag-Based Security Monitors* and by DARPA's *System Security Integrated Through Hardware and Firmware (SSITH)* program.

Appendix

A Generic Proof Technique for *RSCC*

We propose an effective and general proof technique for *RSCC* (§A) by showing that it follows from RSC^{DC}_{MD} (§A.1) and that RSC^{DC}_{MD} can be proved (§A.2) by: *(1)* constructing, from any target-level trace that records cross-component calls and returns, a *whole* source-level program producing that trace; *(2)* using standard simulation proofs to show trace decomposition and composition lemmas that relate our semantics for whole programs to generically constructed semantics that work on partial programs; and *(3)* using a whole-program compiler correctness proof à la CompCert [54] as a black-box for moving back and forth between the source and target languages. This yields significantly simpler proofs than previous work in this space [43], [64], which gives us hope that they can be scaled in the future to something as large as a secure variant of CompCert. Moreover, our technique for proving RSC^{DC}_{MD} is not so specific to unsafe languages; we expect it can be easily simplified in the future to provide scalable proofs of vanilla *RSC* when the source is safe [29].

A.1 RSC^{DC}_{MD} implies *RSCC*

As a first step towards proving *RSCC*, we show that *RSCC* can be obtained by iteratively applying RSC^{DC}_{MD}. This proof crucially relies on back-translation in RSC^{DC}_{MD} being performed pointwise and respecting interfaces, as explained in §3.3.

Theorem A.1. RSC^{DC}_{MD} implies *RSCC*.

We proved this by defining a non-constructive function that produces the compromise sequence $A_{i_1}, ..., A_{i_1}$ 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.

A.2 Proving RSC^{DC}_{MD}

In order to prove RSC^{DC}_{MD} we have designed a general proof strategy that relies on a few key properties of the source and target languages and their compiler. In the next section (§B) we show how to prove each property for our compiler to obtain RSC^{DC}_{MD} for it. One advantage of our proof technique is the use of a compiler that supports separate compilation but only guarantees correctness for whole programs, which is what CompCert provides for example [45]. This constraint slightly complicates the proof but will allow us to more easily tap into the CompCert infrastructure in the future.

The intuition behind RSC^{DC}_{MD} (Definition 3.5) is that a programmer working on a partial program P should not be concerned with the specific C_T that will be linked with their compiled P. Ideally they should only worry about the trace of observable events of C_T and imagine a source context C_S that, interacting with P, could produce that trace. The compiler guarantees that any target trace can be expressed in the source, or conversely that the target is limited to source traces, allowing the programmer to reason only in the source language which he is familiar with. On one side we want to abstract away a specific context C_T and only keep the finite trace prefix m it produces, on the other we want to turn m into a concrete program C_S. Following this intuition, our proof technique is based on two main ideas: a *partial semantics* to execute the target context along a trace prefix separately from the compiled program and a *definability* technique to create a new source program from that finite trace prefix. Additionally the proof uses two standard properties of correct compilers to preserve the traces of complete programs.

Partial Semantics The purpose of a *partial semantics* is to characterize the traces of a partial program P when linked with any context satisfying an interface. This corresponds to the semantics that a programmer has in mind while developing P and imagining the interactions that are possible with the context. When the program is running, the execution is the same as in the normal operational semantics of the language, but when the control is passed to the context, an action compatible with its interface is non-deterministically selected and executed. This models all possible concrete implementations of the context interface, all of which need to be taken into account to characterize the possible traces of P.

We define our partial semantics generically for any language, with respect to the *complete* small-step operational semantics of a complete program $C \cup P$, which we denote as $\xrightarrow{\alpha}$ and define over complete states cs. Each concrete execution step is labeled with an action α that is either an event or a silent action τ. We define a *partialization* function `par` that, given a complete state cs and the interface I_C of C, returns a partial state ps where information regarding C has been erased. For instance, in our particular instance from §4 partialization erases the memories and the stack frames belonging to C (among other things). Given partial states produced by `par` we define the *partial operational semantics* $\overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow}$ over them.

$$\frac{\texttt{par}(cs, I_c) = ps \quad \texttt{par}(cs', I_c) = ps' \qquad cs \xrightarrow{\alpha} cs'}{ps \overset{\alpha}{\rightarrow\!\!\!\!\!\rightarrow} ps'}$$

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 $P \rightsquigarrow_{I_C} t$ that the partial program P linked with any context with interface I_C produces the trace t in the partial semantics after a complete finite or infinite execution. The partial semantics has a different behavior depending on which part of the complete program is running. When the program has control there is a perfect match between partial and complete semantics and if the complete semantics is deterministic only one reduction is possible. On the contrary when the context is running, the partial semantics can non-deterministically pick any complete state compatible with its interfaces and perform the same action. This leads to a non-deterministic reduction that captures all possible contexts that

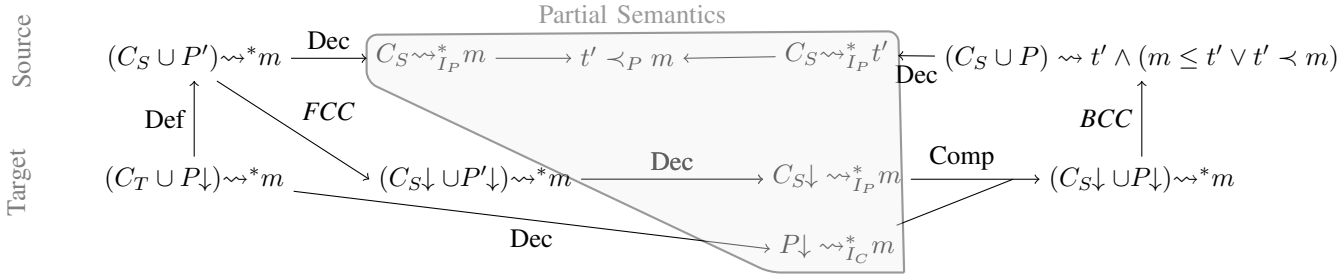


Figure 11: Our proof schema for RSC^{DC}_{MD}

the program can be linked with and their behavior. The only constraint is respecting the interfaces.

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

Decomposition is a first property that one has to prove in order to show that our partial semantics is correct with respect to its complete counterpart and in particular that the partialization function is well designed. Decomposition states that if a program obtained by linking two partial programs P and C produces a trace t in the complete semantics, then each of the two partial programs (below we take P, but the C case is symmetric) can produce a trace t' in the partial semantics so that $t \prec_{I_C} t'$. Because C has been abstracted away, its behavior is always defined in the partial semantics, so if t ended in an undefined behavior caused by C the partial semantics can continue with arbitrary events.

Definition A.2 (Decomposition).

$$\forall P{:}I_P.\ \forall C{:}I_C.\ \forall t.\ (C \cup P) \rightsquigarrow t \Rightarrow \exists t'.\ P \rightsquigarrow_{I_C} t' \wedge t \prec_{I_C} t'$$

One way to prove decomposition is to show that the partial semantics simulates the complete one.

The converse of decomposition, *composition*, states that if two partial semantics with matching interfaces produce the same trace t, then they can be linked and they produce the same t in the complete semantics.

Definition A.3 (Composition). For any two compatible interfaces I_P and I_C:

$$\forall P{:}I_P.\ \forall C{:}I_C\ \forall t.\ P \rightsquigarrow_{I_C} t \wedge C \rightsquigarrow_{I_P} t \Rightarrow (C \cup P) \rightsquigarrow t$$

Our work on partial semantics was partially inspired by a line of work on co called "trace semantics" [41], [43], [63], where similarly it is important to decouple a program of interest from a specific context, of which only the observable behavior is relevant. One important difference is that our generic definition of partial semantics in terms of a partialization function is very general and can be easily instantiated for different languages. On the contrary previous works defined "trace semantics" as a large 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 a lot easier than previous proof techniques for fully abstract "trace semantics."

Definability is the ability of a semantics to generate the behaviors specified by a suitable notion of trace. Intuitively it states that given a *viable* finite trace there should be a program that can produce it. A bit more formally, there should be a *back-translation function* $\uparrow$ that given a viable finite trace m and the interface I of the program returns a complete program such that $(m, i)\uparrow \rightsquigarrow m$. In §B we describe in more detail an instance for our source language. It should be noted that it is not always possible to take an arbitrary sequence of events and obtain a program that realizes it. For example, in a language with a call stack and events $\{\texttt{call}, \texttt{return}\}$, there is no program that produces the single event trace `return`: every return must be preceded by a call. For this reason we apply definability only on traces that are produced by one of our operational semantics, thus leading only to *viable* traces. In our case we care about back-translating complete target programs to source programs so we instantiate definability to take any finite trace produced by a target program and back-translate it to a source program that produces the same behavior.

Definition A.4 (Definability).

$$\exists \uparrow .\ \forall P{:}I_P.\ \forall m,\ P \rightsquigarrow_T m \Rightarrow (m, I_P)\uparrow \rightsquigarrow_S m$$

The fact that our back-translation function only takes finite traces as input simplifies our proof technique but at the same time limits it to only safety properties. While the other properties from this section can 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.

Compiler correctness The last property we need in order to prove RSC^{DC}_{MD} is compiler correctness. We assume we have a compiler that has whole-program compiler correctness, both forward and backward. These two properties are proved in Coq for the CompCert verified compiler [54]. Intuitively forward

compiler correctness, *FCC*, requires that the trace produced by complete source program P can also be produced by its compilation $P{\downarrow}$, and conversely for *BCC*. Additionally a compiler for an unsafe language should take into account the possibility that P might encounter an undefined behavior and that $P{\downarrow}$ at the target can produce an extended trace. Although we require correctness proof only for whole-programs, the compiler must be aware of the component structure of programs. In particular we require that any undefined behavior can be traced to the component that caused it. As usual, we denote with $\prec_C$ that a trace can be extended only if the component C of the program causes undefined behavior.

Definition A.5 (Compiler Correctness).

$$\forall P\ C\ t.\ P{\rightsquigarrow}t \Rightarrow \exists t'.\ P{\downarrow} \rightsquigarrow t' \wedge t \preccurlyeq_C t' \quad (\textit{FCC})$$
$$\forall P\ C\ t'.\ P{\downarrow} \rightsquigarrow t' \Rightarrow \exists t.\ P{\rightsquigarrow}t \wedge t \preccurlyeq_C t' \quad (\textit{BCC})$$

While we require compiler correctness only for whole-programs, we also require the compiler to support *separate compilation*, that is $\forall P\ C\ t.\ (C \cup P){\downarrow} \rightsquigarrow t \Leftrightarrow (C{\downarrow} \cup P{\downarrow}){\rightsquigarrow}t$. This is a reasonable property in a setting were components are present at all levels of the compilation chain and should not interfere significantly with the correctness proof.

The proof strategy for RSC_{MD}^{DC} We now use the definitions introduced above to explain our proof technique for RSC_{MD}^{DC}. This is depicted in the diagram from Figure 11. In the bottom left corner, we start with a complete target-level program $C_T \cup P{\downarrow}$ producing a trace with a finite prefix m. Using definability (Definition A.4) the prefix m is back-translated to a complete source program $C_S \cup P'$ that reproduces the prefix m in the source. The new source program is then compiled to a target program $(C_S \cup P'){\downarrow}$ that by forward compiler correctness (Definition A.5) produces again the same prefix m. By separate compilation we can consider $C_S{\downarrow} \cup P'{\downarrow}$ and then remove the partial program $P'{\downarrow}$ using decomposition (Definition A.2), leaving only its behavior captured by the trace m and its interface I_P. This somewhat convoluted process is necessary to guarantee the preservation of behavior of C_S while using a compiler that only guarantees whole-program correctness. Now that we managed to isolate $C_S{\downarrow}$, we can decompose $P{\downarrow}$ as well from the original program and compose the two to obtain a target program $C_S{\downarrow} \cup P{\downarrow}$ which still preserves m. Lastly we can apply separate compilation to obtain $C_S \cup P{\downarrow}$ and backward compiler correctness to obtain the source program $C_s \cup P$. During the last step however we must take care as the source program might cause an undefined behavior. For this reason the obtained trace t' can relate to m in two possible ways. Either t' is an extension of m, in which case the prefix is guaranteed, or there was a undefined behavior before m, leading to a t' shorter than m. In the second case when the program is compiled there is no guarantee that the prefix will be preserved after t'. The definition of RSC_{MD}^{DC} allows for an undefined behavior to shorten the prefix m, however only in the case that P caused it: we still need to show that C_S doesn't produce any undefined behavior before m. Again to analyze the behavior of C_S alone we apply decomposition and look at its behavior in the partial semantics where the abstract part can't produce undefined behavior. We decompose twice, in the top part of the diagram, once from $C_S \cup P'$ and once from $C_S \cup P$ and because the source language is determinate we obtain the if C_S can produce the whole m on the left, then necessarily if t' is shorter than m it must be because of P. Hence $t' \prec_P m$.

B *RSCC* Proof for the Compiler from §4

Back-translation function For the source language we have also developed a $\uparrow$ function for which we have proved the definability property (Definition A.4). The function takes a finite trace prefix m and a program interface I and returns a complete source program that respects I and produces m. The first step is to generate a skeleton of `procs` that contains all the components and their public procedures with an empty body. We then add in `bufs` a single counter per component which will keep track of which event the component should generate next; each component can access its buffer through `!(local+1)`. The function scans the trace and for every event adds a snippet of code to the body of the procedure that produced that event. More precisely, if we are in component `C1`, with current procedure `P1` and the event is `C1 Call(P2,42) C2`, then if this is the first time that `C1` performs a call in the trace t, the added snippet would be

```
if (local[1] == 1) { local[1]++; C2.P2(42); C1.P1(0); }
```

The code increases the counter so that at the next call, the next snippet will be executed. It then produces the event required by the trace, calling the procedure with the right argument. When returned control, after `C2.P2` is done, it calls itself again and executes the next `if` branch whose guard is now true because the counter was bumped. The resulting procedure is a concatenation of `if` statements where only one guard is true every time the component needs to generate an event, and corresponds to the branch generating exactly that event. A more complete example with a trace of 5 events and 2 components is shown in Figure 12.

In order to prove that our compiler guarantees RSC_{MD}^{DC} we show that it provides the properties presented in §A.2 and apply the general proof schema. Particular care has been taken to maintain compatibility with CompCert's proof technique and data structures and to provide a solid framework for future developments. The only large proof that we assume is the correctness of our compiler. Compiler correctness has been proved for compilers vastly more complex, such as CompCert, and repeating the exercise for our simple instance would be time consuming but not very insightful, so it is left as future work. *Assumption* (Correctness). We assume the front-end compiler $\downarrow$ provides forward and backward compiler correctness (Definition A.5).

The largest part of our proofs is devoted to the partial semantics, one for the source and one for the compartmentalized machine. For each we proved composition and decomposition. These proofs are structured as simulations, a standard technique and the main one used in CompCert, and

```
ECall mainC P1    0 C1
ECall C1    P2    1 C2
ERet  C2          2 C1
ECall C1    P2    3 C2
ECall C2    mainP 4 mainC
```

(a) Trace of 5 events

```
C1 {
  P1(_) {
    if (local[1] == 0) {
      local[1]++;
      C2.P2(1);
      C1.P1(0);
    } else if (local[1] == 1) {
      local[1]++;
      C2.P2(3);
      C1.P1(0);
    } else {
      exit();
    }
  }
}
```

(b) Component 1

```
C2 {
  P2(_) {
    if (local[1] == 0) {
      local[1]++;
      return 2;
    } else if (local[1] == 1) {
      local[1]++;
      mainC.mainP(4);
      C2.P2(0);
    } else {
      exit();
    }
  }
}
```

(c) Component 2

Figure 12: Example of program with two components back-translated from a trace of 5 events.

require care especially when dealing with silent steps. This part of the development is very general and can easily be applied to different instances.

Theorem A.6 (Partial Semantics). The source language and compartmentalized machine partial semantics defined as in §A.2 provide decomposition (Definition A.2) and composition (Definition A.3).

We then show that the back-translation function (B) indeed respects definability. The proof is composed of two main parts, the first showing the program produces a viable trace, the second showing that the output of back-translation function can indeed reproduce any viable trace.

Theorem A.7 (Definability). The generator function ↑ provides definability (Definition A.4).

Theorem A.8 (RSCC). The compiler from §4 satisfies *RSCC*.

References

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

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

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

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

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

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

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

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

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

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

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

[12] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *POPL*. 2014.

[13] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015.

[14] A. Azevedo de Amorim, C. Hriţcu, and B. C. Pierce. The meaning of memory safety. arXiv:1705.07354, to appear at POST, 2018.

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

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

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

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.

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

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

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

[22] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL):38:1–38:23, 2018.

[23] D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, fully-abstract compilation by approximate back-translation. arXiv:1703.09988, 2017.

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

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

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

[27] C. Fournet, A. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. *CSF*. 2007.

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

[29] D. Garg, C. Hriţcu, M. Patrignani, M. Stronati, and D. Swasey. Robust hyperproperty preservation for secure compilation (extended abstract). 2nd Workshop on Principles of Secure Compilation (PriSC), 2018.

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

[31] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *TCS*, 300(1-3):379–409, 2003.

[32] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *JCS*, 12(3-4):435–483, 2004.

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

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

[35] K. Gudka, R. N. M. Watson, S. Hand, B. Laurie, and A. Madhavapeddy. Exploring compartmentalisation hypotheses with SOAAP. In *AHANS Workshop*. 2012.

[36] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. *PLDI*. 2017.

[37] C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. *PLDI*. 2015.

[38] The Heartbleed bug. http://heartbleed.com/, 2014.

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

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

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

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

[43] Y. Juglaret, C. Hritcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016.

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

[45] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *POPL*, 2016.

[46] A. Kennedy. Securing the .net programming model. *Theor. Comput. Sci.*, 364(3):311–317, 2006.

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

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

[49] O. Kupferman and M. Y. Vardi. Robust satisfaction. *CONCUR*. 1999.

[50] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In M. W. Alford, J. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider, editors, *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*. 1984.

[51] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[53] C. Lattner. What every C programmer should know about undefined behavior #1/3. LLVM Project Blog, 2011.

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

[55] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.

[56] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. *PLDI*, 2016.

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

[58] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for compcert. *PLDI*, 2016.

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

[60] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*. 2016.

[61] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.

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

[63] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *CL*, 42:22–45, 2015.

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

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

[66] J. Regehr. A guide to undefined behavior in C and C++, part 3. Embedded in Academia blog, 2010.

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

[68] L. Simon, D. Chisnall, and R. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. To appear at EuroS&P, 2018.

[69] L. Skorstengaard, D. Devriese, and L. Birkedal. Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. 2nd Workshop on Principles of Secure Compilation (PriSC), 2018.

[70] L. Skorstengaard, D. Devriese, and L. Birkedal. Reasoning about a capability machine with local capabilities: Provably safe stack and return pointer management (without OS support). Accepted at ESOP, 2018.

[71] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. To appear at OOPSLA, 2017.

[72] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE S&P*. 2013.

[73] G. Tan. Principles and implementation techniques of software-based fault isolation. *FTSEC*, 1(3):137–198, 2017.

[74] J. Thomas F. Knight, A. DeHon, A. Sutherland, U. Dhawan, A. Kwon, and S. Ray. SAFE ISA (version 3.0 with interrupts per thread), 2012.

[75] S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Towards automatic compartmentalization of C programs on capability machines. *FCS*, 2017.

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

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

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

[79] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? *APSYS*. 2012.

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

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

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

[83] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. *IEEE S&P*. 1993.

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

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