SECOMP: Formally Secure Compilation of Compartmentalized C Programs

Jérémy Thibault[1] Roberto Blanco[1] Dongjae Lee[1,2] Sven Argo[3]
Arthur Azevedo de Amorim[4] Aïna Linn Georges[5] Cătălin Hriţcu[1] Andrew Tolmach[6]

[1]MPI-SP, Bochum, Germany [2]Seoul National University, South Korea [3]Ruhr University Bochum, Germany
[4]Rochester Institute of Technology, USA [5]MPI-SWS, Saarbrücken, Germany [6]Portland State University, USA

arXiv:2401.16277v3 [cs.PL] 23 Apr 2024

Abstract

Undefined behavior in C often causes devastating security vulnerabilities. One practical mitigation is compartmentalization, which allows developers to structure large programs into mutually distrustful compartments with clearly specified privileges and interactions. In this paper we introduce SECOMP, a compiler for compartmentalized C code that comes with machine-checked proofs guaranteeing that the scope of undefined behavior is restricted to the compartments that encounter it and become dynamically compromised. These guarantees are formalized as the preservation of safety properties against adversarial contexts, a secure compilation criterion similar to full abstraction, and this is the first time such a strong criterion is proven for a mainstream programming language. To achieve this we extend the languages of the CompCert verified C compiler with isolated compartments that can only interact via procedure calls and returns, as specified by cross-compartment interfaces. We adapt the passes and optimizations of CompCert as well as their correctness proofs to this compartment-aware setting. We then use compiler correctness as an ingredient in a larger secure compilation proof that involves several proof engineering novelties, needed to scale formally secure compilation up to a C compiler.

1 Introduction

Undefined behavior is endemic in the C language: buffer overflows, use after frees, double frees, signed integer overflows, invalid type casts, various concurrency bugs, etc., cause mainstream C compilers to produce code that can behave completely arbitrarily. This leads to devastating security vulnerabilities that are often remotely exploitable, and both Microsoft and Chrome report that around 70% of their high severity security bugs are caused by undefined behavior due to memory safety violations alone [31, 62, 84].

A strong practical mitigation against such vulnerabilities is *compartmentalization* [26, 43, 53, 88], which allows developers to structure large programs into mutually distrustful compartments that have clearly specified privileges and that can only interact via well-defined interfaces. This way, the compromise of some compartments has a limited impact on the security of the whole program. This intuitive increase in security has made compartmentalization and the compartment isolation technologies used to enforce it become widely deployed in practice; e.g., all major web browsers today use both process-level privilege separation [26, 43, 53] to isolate tabs and plugins [76], and software fault isolation (SFI) [65, 83, 91, 97, 97, 99] to sandbox WebAssembly modules [44, 49, 50, 55, 87].

In this paper, we investigate how one can provide strong *formal guarantees* for compartmentalized C source code by making the C compiler aware of compartments. We follow Abate *et al.* [8], who argue that a compartment-aware compiler for an unsafe language can restrict the scope of undefined behavior both (a) spatially to just the compartments that encounter it [51], and (b) temporally by still providing protection to each compartment up to the point in time when it encounters undefined behavior. Abate *et al.* formalize this intuition as a variant of a general *secure compilation* criterion called *Robust Safety Preservation (RSP)* [9, 10, 74, 75]. Their RSP variant ensures that any low-level attack against a compiled program's safety properties mounted by compartments dynamically compromised by undefined behavior could also have been mounted at the source level by arbitrary compartments with the same interface and privileges, while staying in the secure fragment of the source semantics, without undefined behavior. This strong guarantee allows source-level security reasoning about compartmentalized programs that have undefined behavior, and thus for which the C standard and the usual C compilers would provide no guarantees whatsoever.

Such strong formal guarantees are, however, notoriously challenging to achieve in practice and to prove mathematically. RSP [9, 10, 74, 75] belongs to the same class of secure compilation criteria as full abstraction [4, 71], for which simple and intuitive but wrong conjectures have sometimes survived for decades [33], and for which careful paper proofs can take hundreds of pages even for very simple languages and compilers [37, 51]. Such proofs are generally so challenging that no compiler for a mainstream programming language that is guaranteed to achieve any such secure compilation criterion has ever been built. Moreover, such secure compilation proofs are at the moment often only done on paper [4–6, 10, 12, 13, 29, 37, 39, 47, 51, 68, 70–75], even though at the scale of a realistic compiler, paper proofs would be impossible to trust, construct, and maintain. All this stands in stark contrast to *compiler correctness*: CompCert [59, 60]—a realistic C compiler that comes with a machine-checked correctness proof in the Coq proof assistant—has already existed for more than a decade and is used in practice in highly safety-critical applications [57].

In this paper we take an important step towards bridging this gap by devising SECOMP, a formally secure compiler for compartmentalized C code. To do this, we extend the CompCert compiler and its correctness proof to handle isolated compartments that interact only via procedure calls and returns. Although compiler correctness by itself is definitely not enough to prove secure compilation, since it gives up on programs with undefined behavior, we use it as one key ingredient for such a proof. For this we adopt the

https://t.me/learningnets

high-level proof structure proposed by Abate *et al.* [8], who showed how proving their RSP variant can be reduced to showing compiler correctness together with three security-related properties: *back-translation*, *recomposition*, and *blame* (explained in the next section, §2). Proving these properties at scale and achieving formally secure compilation for a compiler for a mainstream programming language were open research challenges, which we solve in this work by bringing the following **novel contributions**:

- We devise the SECOMP compiler for compartmentalized C programs to RISC-V assembly by extending the syntax and semantics of all the 10 languages of CompCert with the abstraction of isolated compartments that can only interact via procedure calls, as specified by cross-compartment interfaces. For CompCert's RISC-V assembly we propose an enforcement-independent characterization of C compartments that relies on a new shadow stack to ensure the well-bracketedness of cross-compartment control flow. We adapt all 19 passes and all optimizations of CompCert to this extension, except cross-compartment inlining and tail-calls, which we disallow.
- In addition to passing scalar values to each other, our compartments can also perform input and output (IO), which was not the case in the very simple languages studied by Abate *et al.* [8]. Our IO model allows pointers to global buffers of scalars to be passed to the system calls implementing IO and also allows these buffers to be changed nondeterministically by these system calls, which goes beyond what was previously possible in CompCert's IO model.
- We extend CompCert's large-scale compiler correctness proof to account for these changes so that we can use it to show secure compilation. Our extension of the correctness proof is elegant and relatively small, even though two of our changes to the semantics of the CompCert languages are substantial: (1) we extend the CompCert memory model with compartments, and (2) we extend the CompCert trace model with events recording cross-compartment calls and returns, as needed for the secure compilation proof.
- We develop a secure compilation proof for SECOMP in Coq, from Clight, the first intermediate language of CompCert and featuring a determinate [38, 63] semantics (as opposed to CompCert C), down to our extension of CompCert's RISC-V assembly. This proof shows the RSP variant of Abate *et al.* [8], capturing the secure compilation of mutually distrustful C compartments that can be dynamically compromised by undefined behavior. We are the first to prove such a strong secure compilation criterion for a mainstream programming language, which makes this a milestone for secure compilation.
- In order to scale up the secure compilation proofs to SECOMP, we introduce several proof engineering novelties: (1) Because SECOMP uses the memory model of CompCert [61] (extended with compartments), the novel simulation invariants we devise to prove security have to make use of the sophisticated *memory injections* of CompCert [61], which provide a fine-grained characterization of the way memory is transformed during compilation. (2) For back-translation, because system calls may read global buffers, we extend traces with *informative events*, which record *memory deltas*—i.e., changes to global buffers happening during silent steps—and we use those to establish memory injections to prove correctness of the system calls the back-translation generates. (3) For recomposition, we propose a more principled way of proving the required three-way simulation by defining 8 simulation diagrams and providing a general proof that together they imply recomposition. Despite the realistic RISC-V instruction set, we use these diagrams to provide a compact proof of recomposition showing that our RISC-V assembly semantics securely characterizes the compartment abstraction.
- The SECOMP secure compilation proofs end at our extension of CompCert's RISC-V assembly, which is the language where CompCert's compiler correctness proofs also end, and whose semantics still maintains CompCert's block-based memory model. As mentioned above, to this language's semantics we added the extra abstraction of isolated compartments, which formally defines *what* compartment isolation enforcement should do, but which leaves the *how* to lower-level enforcement mechanisms working with a more concrete view of memory as an array of bytes [92, 93] and potentially making use of hardware security features. We additionally show that the compartment isolation abstraction can be enforced at a lower level by designing and prototyping an unverified backend using a variant of the CHERI capability machine [94]. For this we extend a recently proposed efficient calling convention enforcing stack safety [42] to our setting of mutually distrustful compartments by introducing capability-protected wrappers to clear registers on calls and returns and to prevent capabilities from being passed between potentially compromised compartments. Various other enforcement mechanisms should be possible, including SFI [49, 55, 83, 91, 97] and tagged architectures [18, 35], as shown in a simpler setting by Abate *et al.* [8]. At the moment all these lower-level backends are, however, unverified, and extending the secure compilation proofs to cover them is a formidable research challenge that we leave as future work (§11).

Artifact. The SECOMP formally secure compiler is available at https://github.com/secure-compilation/CompCert and as a permanently archived artifact at https://doi.org/10.5281/zenodo.11007679. SECOMP adds ~37k lines of code on top of CompCert, mostly in proofs. In more detail, our extensions to CompCert and its correctness proof comprise ~5k LoC of specs and ~7k of proofs, an increase of 5.9% and 14.2% respectively. In addition to this, back-translation involves ~5k LoC of specs and ~6k of proofs; recomposition ~1k LoC of specs and ~8k of proofs; and blame ~1k LoC of specs and ~4k of proofs. The baseline for our development is CompCert version 3.12 and our experience shows that tracking mainline development should be relatively simple, as our changes are orthogonal to the usual CompCert development.

The machine-checked proofs are generally complete and include no further axioms beyond those already existing in CompCert [64], or small adaptions thereof to account for the addition of compartments to the compiler. One exception to this is an axiom assuming that CompCert can successfully compile the results of our back-translation, which would be very tedious to prove in general, but which we have instead thoroughly tested (§7). The other current gap in our Coq formalization is about connecting compiler correctness, back-translation, blame, and recomposition into a single mechanized secure compilation result (Theorem 8.1); instead at the moment the top-level proof and all the steps are basically complete, but back-translation, recomposition, and blame are still on separate branches that we are currently in the process of merging into the main branch. This is further documented in the `README.md` file.

Outline. We first review the work on which we directly build (§2) and present the key ideas of our work (§3). Then we explain

https://t.me/learningnets

how we extended CompCert and its correctness proof (§4). The following two sections detail the most interesting parts of our secure compilation proof: back-translation (§5) and recomposition (§6). Then we present the assumption that the result of back-translation compiles and how we thoroughly tested it (§7). We put these together into our secure compilation theorem (§8) and then present our lower-level, unverified capability backend (§9). We discuss related work (§10) before concluding with future work (§11). Finally, the appendices include details that we had to cut for space.

2 Background

In this section we briefly review the RSC_{MD}^{DC} secure compilation criterion of Abate *et al.* [8] as well as their high-level proof structure for this criterion, since we make use of both in this paper.

But first, we warm up by reviewing the compiler correctness properties this proof structure makes use of. For this we assume that both the source language (for our security proof this is Clight) and the target language (RISC-V assembly) are given trace-producing semantics. CompCert traces are composed of events recording the calls the whole program makes to system calls performing IO and the results they return. A special event $\mathrm{Undef}(k)$ terminates the trace if undefined behavior is encountered by compartment k (where k is something we added, and which we omit where it is irrelevant). We further extend these traces to cross-compartment calls and returns (Figure 2 in §4). Because the criterion of Abate *et al.* [8] focuses on safety properties [58] we only consider finite prefixes of traces. We write $\mathbf{W_S} \rightsquigarrow m$ when the whole source program $\mathbf{W_S}$ can produce the finite trace prefix m; and analogously $\mathrm{W_T} \rightsquigarrow m$ when the whole target program $\mathrm{W_T}$ can produce m. The compiler correctness guarantee of CompCert states that if a compiled whole program can produce a trace prefix m (i.e., $\mathbf{W_S}{\downarrow}{\rightsquigarrow}\ m$) then the original source program can produce a related trace $m' \preceq m$ (i.e., $\mathbf{W_S} \rightsquigarrow m'$), where the relation $m' \preceq m$ is defined as $m' = m$ when $\mathrm{Undef} \notin m'$, and as $m'_0 \cdot m_1 = m$ when $m' = m'_0 \cdot \mathrm{Undef}(k)$ for some k. Here "$\cdot$" denotes concatenation and m_1 is a completely arbitrary trace suffix that the correctly compiled program is allowed to produce when the source program encounters undefined behavior, which can lead to security vulnerabilities.

Instead of compiling only whole programs though, we assume separate compilation—as proposed by Kang *et al.* [52] and implemented in CompCert since version 2.7—and separately compile a source program $\mathbf{P}$ and a context $\mathbf{C}$, which are intuitively both formed of linked compartments, and which can be linked together to produce a whole program both before compilation using source linking ($\bowtie$), and after compilation using target linking ($\bowtie$). Using these concepts we can define the correctness of a compiler $\downarrow$ like CompCert or our variant SECOMP as follows:

Definition 1 (Backward Compiler Correctness (BCC))**.**

$$\forall \mathbf{C}\ \mathbf{P}\ m.\ (\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m \Rightarrow \exists m'.\ (\mathbf{C} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq m$$

In CompCert this backward compiler correctness (BCC) definition is proved by forward simulation [23, 60], so one also obtains a forward compiler correctness (FCC) result, that the RSC_{MD}^{DC} proof structure of Abate *et al.* [8] also makes use of. Here, instead of obtaining a related trace prefix they instead assume that the prefix one starts from in the source does not end with undefined behavior:

Definition 2 (Forward Compiler Correctness (FCC) [8])**.**

$$\forall \mathbf{C}\ \mathbf{P}.\ \forall m \not\ni \mathrm{Undef}.\ (\mathbf{C} \bowtie \mathbf{P}) \rightsquigarrow m \Rightarrow (\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m$$

All variants of C compiler correctness, including the two above, completely give up on the whole program after it encounters undefined behavior. To mitigate this issue, Abate *et al.* [8] propose a secure compilation notion that restricts the scope of undefined behavior to the compartments that encounter it. Such compromised compartments can only influence other compartments via controlled interactions respecting their interfaces and the other abstractions of the source language (e.g., the stack discipline on calls and returns). Moreover, to model dynamic compromise the scope of undefined behavior is also restricted temporally, by still providing protection to each compartment up to the point in time when it encounters undefined behavior.

Abate *et al.* [8] formalize this intuition as an iterative game in which at each step some (initially empty) set of compartments CK is already compromised and tries to attack the remaining uncompromised compartments $K \setminus CK$, for some set of compartment identifiers K defined in the original compartmentalized program with global interface I, capturing all procedure imports and exports. In each step, the uncompromised compartments are linked together into a source program $\mathbf{P}$ with interface $\lfloor I \rfloor_{K \setminus CK}$, and then $\mathbf{P}$ is compiled and linked with a target context $\mathrm{C_T}$, which puts together the compromised compartments and which has interface $\lfloor I \rfloor_{CK}$. The guarantee obtained at each step in this iterative game is formalized as a property they call RSC_{MD}^{DC}, which defined then explained below:

Definition 3. A compilation chain satisfies *Robustly Safe Compilation with Dynamic Compromise and Mutual Distrust* (RSC_{MD}^{DC}) if there exists a back-translation function $\uparrow$ that takes interface I, a target execution p producing a trace prefix m, and a compartment identifier k, and generates a source compartment such that

$$\begin{aligned}&\forall K \subseteq \mathit{CompIds}.\ \forall I{:}\mathit{Interface}(K).\ \forall CK \subseteq K.\ \forall \mathrm{C_T}{:}\lfloor I \rfloor_{CK}.\ \forall \mathbf{P}{:}\lfloor I \rfloor_{K \setminus CK}.\\ &\forall m \not\ni \mathrm{Undef}.\forall \mathrm{p} : (\mathrm{C_T} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m.\\ &\exists \mathrm{C_S}{:}\lfloor I \rfloor_{CK}.\ \mathrm{C_S} = \bowtie_{k \in CK} (I, \mathrm{p}, k)\uparrow \wedge \exists m'.(\mathrm{C_S} \bowtie \mathbf{P}) \rightsquigarrow m' \wedge m' \preceq_{K \setminus CK} m\end{aligned}$$

The premise on the first two lines states that the compound program $\mathrm{C_T} \bowtie \mathbf{P}{\downarrow}$ has an execution p in the target language producing a trace prefix m, which does not end with an undefined behavior event (i.e., for a trace $m \cdot \mathrm{Undef}(k')$ one looks only at the prefix m). The conclusion makes a step towards providing an explanation for m with respect to the source language semantics. For this it calls the back-translation function $\uparrow$ on each of the compromised compartments $k \in CK$ and it links together the generated source compartments to obtain a source context $\mathrm{C_S}$ with interface $\lfloor I \rfloor_{CK}$. The RSC_{MD}^{DC} property says that the obtained context $\mathrm{C_S}$ linked with the original source program $\mathbf{P}$ can produce a trace m' that is related to m by the formula $m' \preceq_{K \setminus CK} m$. This is a variant of the $\preceq$ relation from BCC that ensures that $m' = m$ when $\mathrm{Undef} \notin m'$, and that $m'_0 \cdot m_1 = m$ when $m' = m'_0 \cdot \mathrm{Undef}(k)$ for some *uncompromised* compartment $k \in K \setminus CK$. Intuitively either the whole target prefix m can be explained by an execution in the source language, in which case we are done; or the compromised compartments have found a way to use the interface in the source language to trigger an undefined behavior in one of the (so far) uncompromised compartments $k \in K \setminus CK$. In this second case, Abate *et al.* [8] will apply RSC_{MD}^{DC} again to an extended set of compromised compartments $CK \cup \{k\}$.

https://t.me/learningnets

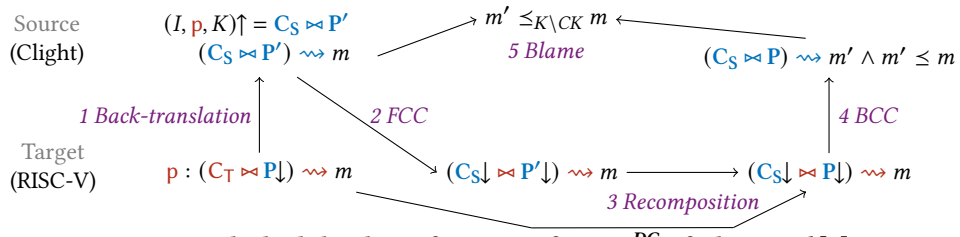


Figure 1: The high-level proof structure for RSC^{DC}_{MD} of Abate *et al.*[7]

Because the semantics is determinate [38, 63], with each iterative application of RSC^{DC}_{MD} the execution is "rewound" along the original trace prefix m and longer and longer prefixes of m are explained in the source, until the whole m is explained in terms of the source semantics and a sequence of dynamic compartment compromises.

For proving secure compilation this iterative aspect is less interesting though, and it basically suffices to show RSC^{DC}_{MD} [8]. For this, Abate *et al.* [8] propose the high-level proof structure from Figure 1 that involves compiler correctness (the FCC and BCC properties above) and three additional security-related properties: back-translation, recomposition, and blame. The high-level proof starts by back-translating a global interface I and a target execution p producing a trace prefix m repeatedly to generate each of the compartments $k \in K$ of a whole source program producing m.

Definition 4 (Back-translation)**.** There exists a function $\uparrow$ s.t.

$$\forall K.\ \forall I.\ \forall W_T{:}I.\ \forall CK{\subseteq}K.\ \forall m \not\equiv \text{Undef}.\forall p : W_T \rightsquigarrow m.$$
$$\exists C_S{:}\lfloor I \rfloor_{CK}.\ \exists P'{:}\lfloor I \rfloor_{K \setminus CK}.\ C_S \bowtie P' = \bowtie_{k \in K} (I, p, k)\uparrow \wedge (C_S \bowtie P') \rightsquigarrow m$$

Using the back-translation function $\uparrow$ to generate a *whole* source program $C_S \bowtie P'$ not only allows the conclusion of Def. 4 to be stated in terms of the usual operational semantics of whole programs ($(C_S \bowtie P') \rightsquigarrow m$), but also allows Abate *et al.* [8] to compile this whole program and make use of FCC in step 2 from Figure 1 to obtain that $(C_S\downarrow \bowtie P'\downarrow) \rightsquigarrow m$. Then, in step 3 they recompose the compartments from this execution with the ones from original execution $(C_T \bowtie P\downarrow) \rightsquigarrow m$ to obtain the execution $(C_S\downarrow \bowtie P\downarrow) \rightsquigarrow m$.

Definition 5 (Recomposition)**.** $\forall K.\ \forall I.\ \forall CK{\subseteq}K.$

$$\forall C_T, C'_T : \lfloor I \rfloor_{CK}.\ \forall P_T, P'_T : \lfloor I \rfloor_{K \setminus CK}.\ \forall m.$$
$$(C_T \bowtie P_T) \rightsquigarrow m \wedge (C'_T \bowtie P'_T) \rightsquigarrow m \Rightarrow (C'_T \bowtie P_T) \rightsquigarrow m$$

Step 4 uses BCC to turn target execution $(C_S\downarrow \bowtie P\downarrow) \rightsquigarrow m$ back into source execution $(C_S \bowtie P) \rightsquigarrow m'$, where the relation $m' \preceq m$ accounts for the possibility of undefined behavior in $C_S \bowtie P$. The context C_S is, however, generated by the back-translation and has no undefined behavior along the trace m, which one shows in step 5 of the proof. So if there is an undefined behavior in m' then this can only be blamed on an as-yet-uncompromised compartment $k \in K \setminus CK$, as required by the conclusion of RSC^{DC}_{MD} from Def. 3.

Definition 6 (Blame)**.** $\forall K.\ \forall I.\ \forall CK{\subseteq}K.\forall C_S{:}\lfloor I \rfloor_{CK}.\ \forall P, P'{:}\lfloor I \rfloor_{K \setminus CK}.$

$$\forall m.(C_S \bowtie P') \rightsquigarrow m \wedge (C_S \bowtie P) \rightsquigarrow m' \wedge m' \preceq m \Rightarrow m' \preceq_{K \setminus CK} m.$$

3 Key ideas

3.1 Compartment model

Compartmentalization [26, 43, 53, 88] allows developers to structure large programs into mutually distrustful compartments that have limited privileges, are isolated from each other, and can only interact in a controlled way. In this work, we adopt a model that statically partitions C programs into compartments. Every C definition of a procedure or a global variable belongs to a single compartment.

Any block of memory belongs to the compartment that allocated it and compartments do not share memory: each block can only be accessed by the code of the compartment it belongs to. Instead, all interactions between compartments must happen via *cross-compartment calls and returns* that respect the *interfaces* provided by the programmers: each compartment `C` comes with a set of exported function declarations (i.e., which function it makes available to other compartments), written `C.exports`, and a set of imported function declarations (i.e., which function it uses from other compartments), written `C.imports`. Compartments must respect the (global) interface defined by these declarations; otherwise they trigger undefined behavior, whose scope is restricted by our secure compilation property to just the offending compartment. Moreover, in our model compartments can only pass each other scalar values as procedure call arguments and return values. We introduced this last restriction for two reasons: first, a compartment cannot use pointers from another compartment, which defeats the purpose of passing pointers in most cases. More importantly, passing pointers to other compartments would require recording these pointers on the trace, which would significantly complicate back-translation, recomposition, and if also done for pointers to dynamically allocated memory also compiler correctness (see §11).

The compartments also interact with an external environment using *system calls*. These system calls are special, privileged functions whose semantics is axiomatized in CompCert,[1] and which might generate some events. Example of these system calls include volatile memory operations, calls to the heap allocator, or input and output (such as reading from the console). The system calls do not belong to a compartment; instead calling them is considered to be a special kind of internal call that can only change the calling compartment's memory.

3.2 Adding compartments to CompCert

Extension to CompCert's languages. Following the ideas above, we extend all of CompCert's 10 languages, from C to RISC-V assembly, adding syntax describing the compartment breakup and interfaces and semantic checks to ensure all compartments respect these interfaces. As explained above, failing a check triggers undefined behavior for the offending compartment. In particular, we update memory operations to take an additional compartment argument, ensuring compartments cannot access other compartments'

[1] Our "system calls" correspond in CompCert to the "external" functions that do not get resolved to actual C source code during linking, and that are implemented in lower-level, trusted libraries (like libc), which may include actual system calls.

https://t.me/learningnets

memories. Also, at every point where control could pass to another compartment (calls and returns at higher levels, jumps at the lowest level) we add a check that the control transfer respects the interfaces and that compartments only pass each other scalar values. Compilation preserves the program's interfaces and linking two partial programs requires that they have compatible interfaces.

To prove secure compilation (Def. 3 from §2), we also extended the trace model with two new events: `Event_call` and `Event_return`. These new events are generated by cross-compartment calls or returns (or the equivalent jumps in RISC-V assembly), and are needed to record enough information on the traces to be able to prove recomposition and back-translation. Since call and return events must not be disturbed by optimization, we disallow cross-compartment tailcall optimization and inlining, as those would substantially change the way compartments interact (e.g., would require merging stack frames belonging to different compartments).

At the RISC-V level, implementing secure compartments required even more care. Without proper protection, adversarial code could make use of the unstructured control-flow inherent to the assembly language to break the compartment abstraction. For instance, an attacker could try to jump to code to which it shouldn't have access. We modified CompCert's RISC-V assembly semantics to prevent this kind of attack, by protecting the compartment abstraction and interfaces. To do so, we observe that calls and returns are only implemented by the compiler using specific instructions—jump-and-link for calls, and indirect jumps for returns—so we forbid all other instructions from changing compartments. Then, when an execution encounters such a jump and attempts to switch to another compartment, we make use of the interfaces and of a newly added shadow stack to decide whether the switch is allowed. If the instruction is a jump-and-link, then the semantics checks whether it is an allowed call according to the interfaces, and then records the return address and the stack pointer on the shadow stack. To decide which indirect jumps to allow to return to a different compartment, we inspect the top frame of the shadow stack, and make sure that the compartment performing the jump is returning to the right address and has correctly restored the caller's stack pointer. This prevents a malicious compartment from returning using the wrong return address or confusing the caller about its stack. More details about our changes to the semantics are discussed in §4.

Correctness proofs. We updated all of CompCert's 19 passes and the simulation proofs showing FCC (Def. 2) to account for the addition of compartments and the new trace model, which by determinacy implies BCC (Def. 1). The updated proofs mainly rely on the compartment information being correctly preserved by compilation, e.g., functions do not change compartment, memory blocks that belong to different compartments are not merged, etc.

Adapting CompCert's compiler correctness Coq proof to account for our changes was a substantial amount of work. We wanted to change the proof as little as possible, but since CompCert is a realistic compiler, it was not always obvious from the start how best to do this. Several times, we made design decisions that seemed adequate, but that turned out to be inadequate much later (e.g., choosing at which precise step to insert a given check), when we discovered that they interacted poorly with some particular compilation pass (e.g., intra-compartment inlining or tail-call optimization) or language (e.g., RISC-V assembly). These issues often did not affect the correctness of the compiler, but made the proofs much more difficult, so we had to backtrack and find alternative ways to structure the changes so as to keep the proofs simple.

In the end we found elegant ways of adapting CompCert's compiler correctness Coq proof to account for all the changes above and proved FCC and BCC. Yet, while compiler correctness is an important part of our security proof, it is definitely not sufficient by itself (§1). In the remainder of this section we discuss the other components of the security proof (§2).

3.3 Back-translation from RISC-V to Clight

In our setting, the first step of the proof structure from §2 is to back-translate a finite RISC-V execution prefix into a Clight program that produces the same trace prefix. We show constructively that given a (whole) compartmentalized RISC-V program and a finite trace prefix of that program, there exists a (whole) compartmentalized Clight program with the same interface that can also produce the same trace prefix. Back-translation resembles compilation, but for RSP [9, 10, 74, 75] and variants like the one we consider, the program obtained by back-translation only needs to preserve *one single finite trace prefix*, not every possible execution of the original program.

Based on this observation, prior works [8, 11, 28, 36, 37, 72] use a simple back-translation from a trace prefix to a program. Each function of the program consists of a loop over a counter, which records how far the trace has been executed. The body of the loop is a switch over the counter value; the nth case of the switch contains code that will produce the nth event of the trace. Proving such a back-translation correct can usually be done in two steps: first, one proves that all traces generated by a target program satisfy a *well-formedness* condition, and then, that every well-formed trace can be back-translated to a program that produces that same trace. We adapt this back-translation and proof technique to our setting, but to do so, we need to make our events more informative and we devise a novel notion of well-formedness of traces made of these informative events, using an intermediate language.

First, the events we use in SECOMP (introduced informally in §3.2 and detailed in Figure 2 in §4) do not contain enough information to directly convert each trace into a Clight statement. In particular, they do not capture all the information necessary to obtain a back-translated program that produces the same events for system calls: if a RISC-V system call produces an event with memory $\mathtt{m}$ and current compartment `C`, then for the back-translated Clight program to produce the same event with memory $\mathtt{m}'$ and current compartment `C`, the CompCert-style axiomatization of system calls requires us to prove that $\mathtt{m}$ and $\mathtt{m}'$ are related by some memory injection [61] that is defined at least on the public symbols of `C`. Put simply, this means that the semantics of system calls is allowed to depend on the content of the calling compartment's global buffers. Since we restrict our semantics to ensure global buffers only contain scalars when calling system calls, this effectively means that $\mathtt{m}$ and $\mathtt{m}'$ must have the same values in `C`'s global buffers. Yet, the SECOMP events do not include the content of global buffers.

This motivates us to define more *informative events* that satisfy two requirements: (1) a RISC-V program always produces a well-formed trace of informative events, and (2) each informative event

https://t.me/learningnets

directly translates into a single Clight statement. Using these informative events, we define a novel notion of *well-formed informative trace* and a back-translation from informative traces to Clight programs, and we extend the simple technique above to prove the correctness of this back-translation. Informative events augment each system call event (and also each cross-compartment call or return event) with a list of the changes to the global buffers since the last informative event. Each of these changes is called a *memory delta*, written δ, and a list of those is written Δ, and is ordered from first to last. Our back-translation function can use these deltas to generate Clight code to produce the same changes to the global buffers before calling the system call.

But first, we define a novel notion of well-formedness of a trace of informative events. To do so, we define a new intermediate language, with a step relation that characterizes the well-formedness of such traces. In this language, states s are triples that record the currently executing function, a memory, and a cross-compartment call stack; the step relation $s \xrightarrow{\beta} s'$ relates the states s and s' and produces an informative event β. The step relation is defined by rules that record the conditions necessary for the back-translated code to be proved correct. In particular, if $s \xrightarrow{\beta} s'$, then applying the deltas Δ from β to the memory $\mathtt{m}$ of s produces the memory $\mathtt{m}'$ of s' (i.e., $\mathtt{mem\_delta\_apply}\ \Delta\ \mathtt{m} = \mathtt{m}'$).

We say that a trace of informative events is *well-formed* when it is produced by the reflexive transitive closure of this step relation. We prove that for any trace prefix m produced by a RISC-V program, there exists a well-formed trace of informative events $\overline{\beta}$ such that $\mathtt{proj}\ \overline{\beta} = m$, where $\mathtt{proj}$ is the function that removes the additional information of informative events. Thanks to this result, we can now define a back-translation function on informative traces, and completely forget about the RISC-V semantics when proving the correctness of the back-translation.

This back-translation function operates in the same standard way explained above, except that it also generates code that writes the right values inside the global buffers before performing a system call, using the memory deltas. We prove the correctness of the back-translation by induction on the intermediate language execution, using the information provided by the step relation. In order to prove system calls at the Clight level generate the same events, we maintain an invariant regarding permissions of global buffers, and prove that it can be used together with memory deltas to build a memory injection that allows us to use the axiomatization of system calls from CompCert. We describe the details of the back-translation and proof technique in §5.

3.4 Recomposition for RISC-V compartments

Recomposition is an essential part of the proof structure of §2, as it allows us to replace the arbitrary RISC-V context we started from with a context that was obtained by back-translation and then compiled. The key intuitions behind this kind of proof [8, 10, 36, 37, 51, 70, 72, 74] are as follows: (1) because of determinacy the internal behavior of a compartment only depends on its internal state, and on all the information it received from other compartments, or from system calls; (2) our traces are informative enough to capture all information that is exchanged between compartments or obtained from system calls. Hence, given a fixed execution trace, if a program takes some internal steps, then the program would take exactly the same steps even if we had changed all compartments other than the currently executing one. That is, compartment-internal steps are unaffected by the other compartments. As a consequence, whenever the execution of a compartment ought to produce a new event, whether it is a system call or a compartment boundary-crossing event, it always produces the same event no matter what the other compartments are.

Formalizing this idea in order to prove recomposition for our RISC-V semantics extended with compartments is highly complex, as it relies on many low-level details of the RISC-V semantics. For this reason, we propose a generic proof technique that elegantly splits recomposition into several self-contained parts. We introduce eight novel CompCert-like simulation diagrams (described in detail in §6) that provide a structured way to think and reason about recomposition, explicitly separating the definition of invariants, the reasoning about internal steps, and the reasoning about events and cross-compartment communication. Together, they imply the existence of a novel *three-way recomposition simulation* that itself implies the recomposition theorem (Def. 5).

We define three-way recomposition simulations in the generic setting on labeled transition systems $(S, \rightarrow)$ with *initial* states.

Definition 7 (Three-way recomposition simulation)**.** Given three labeled transition systems $L_1 = (S_1, \rightarrow_1)$, $L_2 = (S_2, \rightarrow_2)$ and $L_3 = (S_3, \rightarrow_3)$, we say there exists a three-way recomposition simulation between L_1, L_2 and L_3 when there exists a relation $\mathcal{R}$ between states of L_1, L_2 and L_3 that satisfies the following properties:

(1) $\forall$ initial $s_1 \in S_1.\ \forall$ initial $s_2 \in S_2.\exists$ initial $s_3 \in S_3.\ \mathcal{R}\ s_1\ s_2\ s_3$

(2) $\forall s_1\ s_1' \in S_1.\ \forall s_2\ s_2' \in S_2.\ \forall s_3 \in S_3.\ \mathcal{R}\ s_1\ s_2\ s_3 \wedge s_1 \xrightarrow{a}_1 s_1' \wedge s_2 \xrightarrow{a} s_2' \implies \exists s_3'.\ s_3 \xrightarrow{a}{}^*_3\ s_3' \wedge \mathcal{R}\ s_1'\ s_2'\ s_3'$

(3) $\forall s_1\ s_1' \in S_1.\ \forall s_2 \in S_2.\ \forall s_3 \in S_3.\ \mathcal{R}\ s_1\ s_2\ s_3 \wedge s_1 \xrightarrow{\varepsilon}_1 s_1' \implies \exists s_3'.\ s_3 \xrightarrow{\varepsilon}{}^*_3\ s_3' \wedge \mathcal{R}\ s_1'\ s_2\ s_3'$

(4) $\forall s_1 \in S_1.\ \forall s_2\ s_2' \in S_2.\ \forall s_3 \in S_3.\ \mathcal{R}\ s_1\ s_2\ s_3 \wedge s_2 \xrightarrow{\varepsilon}_2 s_2' \implies \exists s_3'.\ s_3 \xrightarrow{\varepsilon}{}^*_3\ s_3' \wedge \mathcal{R}\ s_1\ s_2'\ s_3'$

Property (1) states that the simulation relation $\mathcal{R}$ is compatible with initial states. Property (2) states that whenever the first two executions take a step from related states producing the same observable event a, then so can the third one. Properties (3) and (4) state that silent steps in either of the first two executions preserve the simulation relation (and the third execution is allowed to take some silent steps too). We prove that, given such a three-way simulation between the semantics of three programs W_1, W_2 and W_3, then $W_1 \rightsquigarrow m \wedge W_2 \rightsquigarrow m \implies W_3 \rightsquigarrow m$, which directly implies the recomposition theorem. Simply put, it is enough to follow both executions in W_1 and W_2, and to apply the appropriate property, (2), (3), or (4) until all of m is produced. Note that we do not need a notion of decreasing measure or a notion of final states, because this simulation is only concerned with finite execution prefixes, as those are the only ones that matter in the proof of recomposition.

We instantiate relation $\mathcal{R}$ above with the conjunction of three relations, following ideas of El-Korashy*et al.* [36, 37]: a *strong* relation $\sim$, a *weak* relation $\equiv$, and a *mixed* relation $\mathcal{M}$. Intuitively, the strong relation $\sim$ relates the parts that are currently being executed, the weak relation $\equiv$ relates the parts that are not being executed, and the mixed relation $\mathcal{M}$ relates the parts shared between compartments (such as the stack). Given three states s_1, s_2, s_3 that are

https://t.me/learningnets

executing a compartment that's taken from W_1, the strong relation $\equiv$ relates s_1 to s_3, and the weak relation $\sim$ relates s_2 to s_3. Symmetrically, when the current compartment is taken from W_2 then $s_1 \sim s_3$ and $s_2 \equiv s_3$. When switching compartments, the relations may be switched as well, depending on whether the new compartment is still taken from the same side or not.

Compared to prior work [36, 37], a significant challenge in our proofs is that in our realistic setting the three relations above are parameterized by two CompCert *memory injections* [61], one for each of the original runs, whose role is to relate the memories of their respective runs to the memory in the recomposed run. These injections only relate the memory belonging to the compartments kept in their respective run, and are updated as compartments allocate more memory. Essentially, the weak relation relates only the memories using the injection of the current run, while the strong relation also relates the content of the registers. The mixed relation relates the content of the stack and cross-compartment stack, using both memory injections.

3.5 Blame for Clight semantics

The blame theorem is the final proof step of §2 and shows that the back-translated program $\mathbf{C_S} \bowtie \mathbf{P'}$ is free of undefined behavior along the given trace prefix, from which it follows that any undefined behavior in $\mathbf{C_S} \bowtie \mathbf{P}$ must come from the original (partial) program $\mathbf{P}$. To do this, blame relates the executions of two whole Clight programs that produce the same trace prefix and share a common set of compartments from $\mathbf{C_S}$—their *context side*—linked with a pair of compatible (same public symbols, imports and exports, etc.), but otherwise arbitrary *program sides* $\mathbf{P}$ and $\mathbf{P'}$, which supply the remaining compartments. Intuitively, because the two executions produce the same trace prefix, the shared context side affects the two program sides of the executions in equivalent ways, and any differences, including undefined behavior, must originate in the different program sides.

Some of the intuitions of our blame proof are similar to our recomposition proof outlined in §3.4, but carried over to Clight and to a different type of simulation involving three partial program parts arranged in two whole programs. The key challenge of the blame proof lies in the definition of the simulation invariants that relate the two executions. The shared trace prefix forces both executions to run in sync: at any point in time, either the shared context side is driving the two runs *in lockstep*, or each program side is running *independently* from the other until an observable event forces them to re-synchronize. We can apply the ideas outlined above to prove a small number of elementary simulation results for an appropriate relation $\mathcal{R}$, and use these to assemble a full proof of blame:

(1) $\mathcal{R}$ is preserved when *each* of the two whole programs takes a *single step*, each producing the *same trace*. (Proved separately starting from the program side and the context side.)
(2) $\mathcal{R}$ is preserved when *one* of the whole programs takes a *silent step* from the *program side* while the other stays put.

On top of these stepwise results, we can build three preservation properties on longer *synchronized executions* of the two programs.

(1) $\mathcal{R}$ is preserved when *each* of the two whole programs takes a sequence of *silent steps*, followed by a pair of *synchronous steps* producing the *same event*. (Proved separately starting from the program side and the context side.)
(2) $\mathcal{R}$ is preserved after any *shared trace prefix* produced by a pair of executions of the two whole programs.

Finally, synchronized executions allow us to reason about full program runs and, in particular, about the final states at the end of execution. We employ two key lemmas for this.

(1) If $\mathcal{R}$ holds, one of the runs executes *until unable to step on the context side*, and the other run is able to execute *extending the same trace to completion*, then the first run also executed to completion (i.e., it did not get stuck).
(2) If $\mathcal{R}$ holds, one of the runs executes *until unable to step*, and the second run can execute a *strict extension of the same trace*, then the first run stopped *on the program side*.

The proof of the blame theorem proceeds by case analysis on the structures of finite trace prefixes and of the CompCert small-step semantics framework. This process is somewhat technical but the assemblage of the overall simulation structure of blame out of the previous key lemmas is itself straightforward.

4 Extending CompCert with Compartments

In this section, we detail how we added compartments to CompCert's languages, including RISC-V assembly.

Memory model. We reuse the block-based memory model of CompCert, extended with compartments. Each memory block belongs to a single compartment that is assigned at allocation and cannot be changed during the execution. Memory operations (reads, writes, frees) are parameterized by the compartment performing the operation, and fail when this compartment does not own the targeted block. This means that compartments cannot share memory or pass data other than by performing calls and returns.

Calls and returns. We extend CompCert's trace model to include two more events capturing compartment transitions (see Figure 2). We write $\overline{x}$ to denote a list of x. The new events are highlighted: `Event_call C C'.f args` captures a cross-compartment call passing `args` from compartment `C` to the procedure `f` of compartment `C'`. Similarly, `Event_return C' C v` represents returning value `v` from compartment `C'` to compartment `C`.

```
a := Event_syscall name v̄ b̄s v b̄s
   | Event_vload m ch id ofs v
   | Event_vstore m ch id ofs v
   | Event_annot string v̄
   | Event_call cp cp.id v̄
   | Event_return cp cp v
```

Figure 2: SECOMP events

For all languages but RISC-V, we make use of the existing structure of the semantics to implement these new events, so the only required change is to add the events to the appropriate call and return transitions. We also insert in the semantics dynamic checks to ensure calls and returns conform to the interfaces and do not pass pointers. In these languages, semantic states are divided into three groups: *standard states*, *call-states*, and *return-states*. Standard states are used when executing a function's code, while call-states and return-states are used when the machine is preparing to execute a call or a return. We implement the new events and the dynamic checks related to calls and returns at the border between these different types of states. The dynamic checks act as follows: An internal call, i.e., a call from `C.f` to `C.g` or to a system call, is always allowed. A cross-compartment call from `C.f` to `C'.g` is allowed if (1) `g` $\in$ `C'.exports`; (2) `C'.g` $\in$ `C.imports`; (3) all of the arguments

https://t.me/learningnets

are scalar values. The dynamic checks for cross-compartment returns are similar: we check that the returned value is a scalar. In all languages before RISC-V, the control-flow well-bracketedness is ensured by the semantics. We describe in more details on an example how calls and returns are executed in SECOMP in Appendix A

Lastly, to achieve security, we made all registers caller saved, since we cannot trust the callee compartment to save and restore the caller compartment's registers, and we made the semantics invalidate non-argument registers on cross-compartment calls and non-return registers on cross-compartment returns (by making them undefined values), since recomposition requires all information passed between compartments to be captured by the trace.

Changes to RISC-V. As explained in §3.2, the changes we made to add compartments to the semantics of RISC-V assembly are more extensive. Compared to standard CompCert's RISC-V, our modified RISC-V assembly adds a boolean flag to the jump-and-link instructions $\texttt{Pjal}_\texttt{s}$ and $\texttt{Pjal}_\texttt{r}$, and to the indirect jump instruction $\texttt{Pj}_\texttt{r}$. When this tag is `true`, the instruction can be used to attempt cross-compartment calls (only for the jump-and-link instructions) or returns (only for the indirect jump). Only these instructions are allowed to change the current compartment. Our semantics also makes use of a *cross-compartment shadow stack*, which is a list of stack frames, each recording a return address, a stack pointer, and a function signature. Whenever a cross-compartment call occurs, a new shadow stack frame is allocated to record the previous return address and stack pointer. Whenever a cross-compartment return occurs, the semantics checks that the return targets the right address and that the stack pointer has been correctly restored, before executing the return. The signature is used to check that the registers used for returns only contain scalars. Importantly, the cross-compartment shadow stack is *not* used for calls and returns inside a compartment: the shadow stack captures the necessary security measure to implement in order to ensure well-bracketedness of cross-compartment control-flow [16], but leaves full liberty to chose any calling convention when staying inside a compartment. We describe one enforcement mechanism for well-bracketedness based on capabilities in §9.

Buffer-based IO. We also extended the IO model of CompCert from single-character-based to buffer-based IO. Before our change CompCert modeled only very simple IO functions. System calls and their arguments and return value, which must be scalars or pointers to globals, are recorded as events in the program trace. The behavior of all system calls is described by a single high-level axiomatization that enforces various generic properties, which are sufficient to support the compiler correctness proof. In particular, system calls are required to be *determinate* [30, 38], in the sense that any two calls with the same arguments and results have the same effect on memory, and *receptive*, meaning that any call might return an arbitrary result value of the correct type. While this is adequate to model single-character-based IO functions like `getchar` and `putchar`, it does not account properly for calls that read or write memory as a side-effect, which are very common in real C code. For example, the system call `read` takes as arguments the number of bytes to read and a buffer address, stores bytes into the buffer, and returns the actual count of bytes that were stored. Two `read` calls might return the same count but store different values in memory (violating determinacy); moreover, the count cannot exceed the requested number of bytes (violating receptivity).

We address these limitations by extending CompCert's system call events to explicitly record any bytes loaded from or stored to global memory buffers (the highlighted arguments to `Event_syscall` in Figure 2). We weaken determinacy to allow calls to store different byte values even if they return the same function result value, and we weaken receptivity to put function-specific constraints on result values and stored bytes. For the latter, we just require that the result and stored bytes might be produced by a call to the function with *some* environment and initial memory. To validate our approach, we give detailed models for `read` and `write` system calls, and show that they indeed satisfy the (weakened) properties.

5 Back-Translation Proof Details

We now detail the challenges involved in adapting the back-translation from Abate *et al.* [8] and its proof to our setting. As explained in §3.3, the main difficulty stems from the fact that, when performing a system call in RISC-V generating an event, we have to prove that it is possible to generate the exact same event in Clight. However, the axiomatization of CompCert's system calls does not allow us to do this easily. Among the axioms CompCert gives us, determinacy, which states that executing the system call in the same memory state yields the same result, is not sufficient. There is indeed little hope of perfectly reproducing the RISC-V memory in Clight.

Instead, CompCert provides another useful axiom: if a system call is executed in some memory $\texttt{m}_1$ and results in memory $\texttt{m}_1'$ while generating a trace, then if executed in some other memory $\texttt{m}_2$ that $\texttt{m}_1$ injects into, the same system call with the same arguments results in a memory $\texttt{m}_2'$ that $\texttt{m}_1'$ injects into, and crucially generate the same trace. Yet using this axiom imposes another condition on the memory injection: it must be defined at least on the public symbols of the environment (of the calling compartment), which include the global buffers. Because the global buffers only contain scalars at the point of system calls, this means we must make sure that the RISC-V memory and the memory of the program produced by back-translation contain the same values in these buffers. This motivates our usage of informative events to record the content of these global buffers inside memory deltas.

Informative events. We consider 3 kinds of informative events:

(1) $\texttt{Icall f t g}\ \overline{\texttt{v}}\ \texttt{sg}\ \Delta$ represents a cross-compartment call, where `f` is the name of the caller, `t` is the trace event produced by the call, `g` is the name of the callee, $\overline{\texttt{v}}$ are the arguments, `sg` is the signature of the callee, and Δ are the memory deltas.
(2) $\texttt{Ireturn f t v}\ \Delta$ represents a cross-compartment return, where `f` is the current function name, `t` is the trace event produced by the return, `v` is the return value, and Δ are the memory deltas.
(3) $\texttt{Isys f t ef}\ \overline{\texttt{v}}\ \Delta$ represents a system call, which is similar to the call case except that it requires `ef`, the kind of system calls, instead of `g` and `sg`.

In this definition, Δ represents a list of memory deltas. A memory delta δ records all kinds of memory operations. For instance, $\texttt{delta\_store ch b o v C}$ represents a store at location $(\texttt{b}, \texttt{o})$ of value `v` performed by compartment `C`, with memory chunk `ch`. Intuitively, given an informative event that contains a list of memory deltas Δ, this Δ records every memory operation that happened since the last

https://t.me/learningnets

```
bt_event C beta := match beta with
| Isys _ _ ef vs Delta => bt_delta C Delta; bt_sys ef vs ...
bt_delta C Delta := match Delta with
| delta_store ch b o v C :: Delta' =>
  if b maps to symbol X then X := v; bt_delta C Delta'
  else bt_delta C Delta' ...
```

Figure 3: Back-translation of an informative trace to Clight

informative event. Then, reapplying the exact same stores allows us to reconstruct the global buffers in Clight.

We capture this intuition as part of a novel notion of well-formedness of informative traces using an intermediate language.

Well-formedness of informative events. To characterize the informative traces that can be back-translated, we define a state transition relation that abstracts over the RISC-V semantics. States are triples containing parts of the RISC-V state at the points where events are generated: $\texttt{state} := (\texttt{f}, \texttt{m}, \texttt{fs})$. Intuitively, the first element is the current function; the second element is the current memory in the RISC-V execution; and the third element is a simplified view of the stack, seen as a list of functions.

The relation $s \xrightarrow{\beta} s'$ captures the conditions for a state to take a step while producing an informative event, including relations among the global environment ge and functions, memory deltas and memory updates, and well-bracketedness of cross-compartment calls and returns. However, it does not include information irrelevant to the back-translation to Clight, such as all the low-level details of the RISC-V semantics. For instance, the following (simplified) rule describes the conditions for a system call to be performed:

$$\frac{\texttt{mem\_delta\_apply ge C } \Delta\ \texttt{m}_1 = \texttt{m}_2 \qquad \texttt{globals\_scalar ge C}_1\ \texttt{m}_2 \qquad \texttt{wf\_deltas } \delta}{(\texttt{f}_1, \texttt{m}_1, \texttt{k}) \xrightarrow{\texttt{Isys f}_1\ \texttt{(Event\_syscall C ef v)\ ef v } \Delta} (\texttt{f}_1, \texttt{m}_2, \texttt{k})}$$

This rule is an abstraction of the corresponding rule in the RISC-V semantics: it keeps the conditions for the global environment, trace, shadow stack, and global buffers, and adds conditions for updating the memory according to the memory deltas. The rules for calls and returns are similar, except that they also update the stack.

An informative trace is well-formed if it satisfies the reflexive transitive closure of the step relation, starting from a state corresponding to the initial state of RISC-V. We prove that for any prefix m of a trace produced by a RISC-V program, there exists a well-formed informative trace $\overline{\beta}$, such that $\texttt{proj}(\overline{\beta}) = m$. Thanks to this result, we can forget about the RISC-V semantics, and simply write a back-translation that takes as an input a well-formed informative trace, and then prove its correctness using the well-formedness.

Back-translation function. Our back-translation function constructs a Clight program by constructing functions from the informative trace. The core definitions are given in Figure 3: `bt_event` converts each informative event into a Clight statement which produces the same trace; this is then wrapped inside a switch statement and a loop, as described in §3.3. We remark that the back-translation heavily relies on well-formedness: otherwise, it generates code that attempts to write to locations to which it doesn't have access.

Correctness of the back-translation. We prove a simulation between the intermediate language describing the well-formedness of the informative trace and the Clight semantics of the program. We prove that given two states, s_i of the intermediate language and s_C of Clight that are related by the simulation invariant, then a transition $s_i \xrightarrow{\beta} s_i'$ in the intermediate language corresponds to a sequence of transitions $s_C \xrightarrow{t}{}^{*} s_C'$ in Clight, where $t = \texttt{proj}\ \beta$. Additionally, we prove that s_i' and s_C' are related by the simulation invariant, which crucially does not require the memory of the intermediate language state to inject into the Clight memory. Instead, we only maintain the invariant that the global buffers of each compartment are writable. Then, when encountering a system call, we can use the facts stored in the rule above: $\texttt{mem\_delta\_apply ge C } \Delta\ \texttt{m}_1 = \texttt{m}_2$, $\texttt{wf\_delta } \Delta$, and $\texttt{m}_2$ only contains scalars: $\texttt{globals\_scalar ge C}_1\ \texttt{m}_2$. Using these we show that the back-translation of this informative event writes all the necessary values in the global symbols, and we can reconstruct a memory injection that is only defined on these buffers. We can then apply the CompCert axiom discussed in the 2nd paragraph to prove that the system call succeeds, and produces the same event.

Because we use a finite, `int64` counter to track how many events have been produced, we must limit the length of the trace to 2^{64} events. Moreover, because CompCert is partial, we must assume that the compilation of the back-translated program succeeds (see §7).

6 Recomposition Proof Details

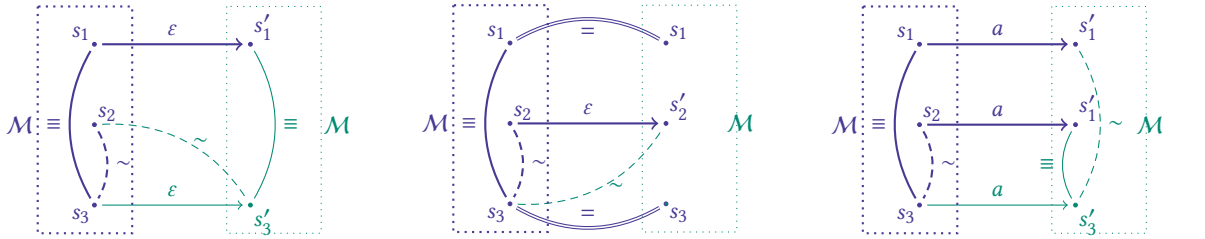
In §3.4, we described at a high level our proof technique. We also developed a collection of proof diagrams, that, together, imply the three-way simulation of recomposition. Three of these are depicted in Figure 4. In each of the diagram, each row represents one of the 3 executions. Arrows represent execution steps, and are annotated with either an event a or silence ϵ. We use thick, dark purple for the assumptions, and dark green for the conclusions we have to prove. We depict in dashed line the weak relation, and in plain line the strong relation; and we use dotted rectangles to depict the mixed relation. We denote equality via a double line in Figure 4b.

Figure 4a describes the case where s_1 and s_3 are taking silent step synchronously and can be read as follows: starting from three states s_1, s_2, and s_3 related by $\mathcal{M}$, such that $s_1 \equiv s_3$ and $s_2 \sim s_3$, and such that s_1 steps silently to s_1', we have to prove that s_3 also steps silently to another state s_3', that the weak and strong relation are reestablished for s_3' (i.e., $s_1' \equiv s_3'$ and $s_2 \sim s_3'$), and that the mixed relation is also reestablished. Similarly, Figure 4b describes the case where s_2 takes silent steps; because it is only weakly related to s_3, we do not require either s_1 or s_3 to take steps as well, but we must still reestablish the relations. Finally, Figure 4c describe the case where both executions produce an event a. Given $s_1 \equiv s_3$ and $s_2 \sim s_3$, related by $\mathcal{M}$, such that $s_1 \xrightarrow{a} s_1'$ and $s_2 \xrightarrow{a} s_2'$, one must prove the existence of s_3' such that $s_3 \xrightarrow{a} s_3'$, i.e., that the three execution advance in lockstep. If the event constitutes a change of control between the two sides (program and context) weak and strong relations are be swapped as illustrated in the diagram; but we also use a similar diagram where the relations are not swapped.

We prove these diagrams imply the existence of the above three-way simulation, and hence imply the recomposition theorem. To do so, we simply instantiate $\mathcal{R}$ with the conjunction of weak, strong, and mixed relations applied to the appropriate cases.

Applying diagrams to our setting. We now explain how we instantiate the parameters of our proof diagrams. At the RISC-V

https://t.me/learningnets



(a) Silent step in strongly related states (b) Silent step in weakly related states (c) Non-silent step with swapping relations

Figure 4: Recomposition diagrams

level, states are the disjoint union of regular states $s{=}(regs, m, st)$ and return-states $rs{=}(regs, m, st, \mathtt{C})$ where *regs* is a register set, m is a memory, st a shadow stack, and C a compartment name recording which compartment the execution is returning from. To handle the possibility of having different allocation behavior in each execution, we relate memories and values using two memory injections j_1 and j_2, one for each of the original executions. These memory injections are only defined on their executions' kept compartments, and do not describe the other compartments memory. These memory injections parameterize the relations, are kept as part of the ghost state we maintain, and are updated during the execution.

7 Compiling back-translation result

The FCC statement of Abate *et al.* [8] (reproduced in Def. 2) assumed that the compiler is a total function. While this was true of their very simple compiler, it is not the case for realistic compilers like CompCert, so our FCC theorem needs an extra assumption that the compiler can successfully compile C and P:

Theorem 7.1 (Forward Compiler Correctness (FCC)). $\forall \mathbf{C}\ \mathbf{P}.$

$$\forall m{\not\in}\mathsf{Undef}.\ (\mathbf{C}{\triangleright\!\triangleleft}\mathbf{P}) \rightsquigarrow m{\wedge}\mathbf{C}{\downarrow}\ \textit{and}\ \mathbf{P}{\downarrow}\ \textit{are defined} \Rightarrow (\mathbf{C}{\downarrow} \bowtie \mathbf{P}{\downarrow}) \rightsquigarrow m.$$

This extra assumption in FCC leads to a new assumption in our RSC^{DC}_{MD} proof, namely that the result of our back-translation can be successfully compiled. But this is not at all easy to prove: CompCert has many sources of partiality and it is not feasible to guarantee in advance that a (well-typed) C program will be successfully compiled. Two passes in CompCert, register allocation and linearization, are not verified but rely on translation validation, which can fail. The data-flow analyzer used in several optimization passes can fail if the analysis doesn't converge after a very large number of steps (e.g. 10^{12}). Several passes (notably Asmgen) use errors to rule out ill-formed code that should never have made it this far, but which is easier to recheck than to prove impossible. Finally, several passes (CminorGen, Inlining, Stacking, etc.) put constraints on the size of the generated stack frames to ensure that offsets within these frames don't overflow a machine word.

Compiling the back-translation is just an artifact of the high-level proof structure (Figure 1), which uses compiler correctness for whole programs to repeatedly move between the source and target languages. Therefore we assume as an axiom that the result of our back-translation, on any trace prefix below a certain length, *can* be successfully compiled. The length bound is needed to account for machine words being finite and other such finite resources.

Assumption 1 (Back-translation successfully compiles)**.**

$$\forall K.\ \forall I.\ \forall \mathbf{W_T}{:}I.\ \forall m \not\in \mathsf{Undef}.\ |m| \leq \mathit{MAX\_TRACE\_LENGTH} \Rightarrow \forall \mathbf{p} : \mathbf{W_T} \rightsquigarrow m.\ \forall \mathbf{C}{:}\lfloor I \rfloor_K.\ \mathbf{C}{=} \bowtie_{k \in K} (I, \mathbf{p}, k){\uparrow} \Rightarrow \mathbf{C}{\downarrow}\ \text{is defined}$$

The next paragraphs report how we have systematically tested this assumption for a large number of trace prefixes. In the future one could envision using more compositional compiler correctness results than that of CompCert, recent [54, 77, 80, 98] or upcoming, to potentially overcome the need for this assumption. For now though, we take this assumption as a reasonable cost to pay for a secure compilation proof technique that is the first to scale up to a realistic compiler like CompCert and that only requires an operational semantics for whole programs, which is not compositional, but which simplifies proofs, including CompCert's existing compiler correctness proof that we extended here to isolated compartments.

Property-based testing of Assumption 1. We systematically tested that the Clight programs constructed by our back-translation function can be compiled with CompCert again. Concretely, we experimentally test that `ccomp(bt_fun(`bt, env`))` succeeds for random but *consistent* bundled traces bt and environments env (§5). The environment defines a set of compartments, their interfaces, and the available functions that can be referenced in the traces. We generate random environments by deriving them from random, undirected and connected graphs $\mathcal{G} = (V, E)$. Each vertex $v \in V$ represents a compartment and we associate it with a random, non-empty set of functions and signatures `v.exports`. Further, for each $(u, v) \in E$ we set `u.imports` to a random, non-empty subset of `v.exports` and vice-versa (c.f. §4). The trace is generated consistent with the environment such that (1) each function call is allowed (c.f. §4); (2) two calls to the same function use the same signature and (3) all values passed as arguments or return values match the signature. For efficiency, we only generate values and in particular memory deltas that are explicitly inspected in the back-translation function and not trivially compiled to skips.

We have been able to successfully compile all Clight programs produced by back-translation for more than 100.000 pairs of generated environments and traces with up to 880 events and close to 400 events on average. Individual tests with significantly longer traces of more than 150.000 events also succeeded, but the growing computational costs make it hard to test even longer traces. In total, the traces contained over 16 million `Icall` and `Ireturn`, 7 million `Isys` and 500 million `delta_storev` instances.

https://t.me/learningnets

8 Top-level RSC_{MD}^{DC} theorem

We follow the general proof diagram from Figure 1 to assemble our previous theorems and obtain the final result that our compilation chain satisfies a variant of RSC_{MD}^{DC} from Def. 3.

Theorem 8.1 (RSC_{MD}^{DC}). *SECOMP satisfies RSC_{MD}^{DC} for all trace prefixes m such that $|m| \leq$ MAX_TRACE_LENGTH.*

The size of the trace prefixes supported by this theorem is restricted to the word size (2^{64}) by our concrete back-translation function (§5) and also by the need to successfully compile the results of the back-translation (Assumption 1), which we have systematically tested as described in §7. A second disclaimer is that we have not yet finished integrating the Coq proof of this theorem with the Coq proofs of the individual steps, since as mentioned at the end of §1, the proofs of back-translation, recomposition, and blame are basically complete too, but they were done on separate branches that we are currently in the process of merging.

9 Enforcement using capabilities

To show that the capability abstraction we added to the semantics of CompCert's RISC-V assembly is practically implementable at a lower level, we designed a capability backend for SECOMP. The backend targets an extension of the CHERI RISC-V architecture [94], which provides hardware capabilities; i.e., unforgeable pointers with base and bounds that cannot be circumvented. While various secure calling conventions targeting capabilities have been proposed in recent years [41, 78, 79, 86], our backend is based on the most recent proposal of Georges *et al.* [42], which uses not only the standard capabilities described above, but also CHERI's local capabilities [78], entry, and sealed capabilities. Additionally, this calling convention is based on two newly proposed kinds of capabilities: uninitialized [41] and directed [42]. In short, uninitialized capabilities prevent reading old values from the stack without excessive clearing [41], and directed capabilities support efficient implementation of stack safety [42]. Our backend targets a lower-level variant of CompCert's RISC-V assembly language with a flat memory model and extended with all these capabilities.

While our calling convention is inspired by Georges *et al.* [42] we had to adapt that design to our setting in two ways: First, because we only enforce compartment isolation, not memory safety, we represent pointers as offsets into a large stack capability or into per-compartment heap capabilities. By not using directed capabilities for stack pointers, we overcome a potential limitation of Georges *et al.*'s [42] calling convention and can store cyclic data structures on the stack. Second, compared to Georges *et al.* [42] we consider a stronger attacker model, in which both the caller and the callee compartments of a call can be compromised. In our model we thus need to always maintain the distinction between the caller and callee compartments and enforce that no capabilities are exchanged between the two. We achieve this by adding privileged wrappers for calls and returns, which ensure that the passed arguments/returns are not capabilities, and which clear all remaining registers.

We built a prototype implementation of this backend in Coq that can already compile simple examples, but that has not yet been thoroughly tested and is not verified. In the short run, we plan to use property-based testing to get more confidence in its correctness and security. We are also investigating the design of a second capability backend inspired by the original work of Watson *et al.* [95] and implemented in CheriBSD [1, 40], which only uses the existing features of CHERI. This second backend, however, requires a split stack layout, which is allowed by the C standard and the CompCert memory model, but which changes the RISC-V calling convention. In the long run, formally verifying such backends is a very interesting research challenge, as also discussed in §11.

10 Related work

As explained in §2, we directly build on the work of Abate *et al.* [8], in particular by reusing their RSC_{MD}^{DC} secure compilation criterion and their high-level proof structure. Scaling up these ideas from a very simple compiler for a toy programming language all the way to a verified compiler for the realistic C language was an open research challenge that we overcome in this work by providing several novel contributions that we explained throughout the paper.

The security proofs of Abate *et al.* [8] and also a later variant with pointer passing [36] (discussed in §11) are both mechanized in Coq. Even if compiler correctness is assumed, these are among the few proofs of secure compilation against adversarial contexts (i.e., for criteria like full abstraction and RSP [10]) that have been mechanized in a proof assistant, with the majority of work in this space being proved only on paper, usually for even simpler languages and compilers [4–6, 10, 12, 13, 29, 37, 39, 47, 51, 68, 70–75]. One proof that was fully mechanized in Coq is that of Devriese *et al.* [34], who prove modular full abstraction by approximate back-translation for a compiler between the simply typed to the untyped λ-calculus. Jacobs *et al.* [46] prove in Coq the purity of a Haskell-like ST monad stated as full abstraction of a translation from a pure language. Georges *et al.* [42] prove in Coq the security of their calling convention (a variant of which we also use in §9) stated as the full abstraction of the identity compiler between a secure overlay semantics and the actual semantics of a simple idealized assembly language. Finally, Abate *et al.* [9, §7.1] verify a very simple compiler in Coq illustrating secure compilation when the target language has additional trace events that are not possible in the source.

A more realistic related work is CompCertSFI [24], which builds on previous ideas by Kroll *et al.* [56] to implement portable SFI as a source-to-source transformation in Cminor, an intermediate language of CompCert that comes before optimizations. Pointers are represented as integers and masked in order to offset into a single big array representing all of the sandbox's memory. In addition to masking pointers and using trampolines for functions pointers, CompCertSFI instruments the program to prevent any undefined behavior. This is needed to properly preserve the main security result, showing that all memory accesses stay within the sandbox, down to CompCert's assembly language. An experimental evaluation shows that the overhead of CompCertSFI comes mostly from CompCert itself performing less aggressive optimizations than GCC and Clang. When the proposed SFI transformation is instead used with GCC or Clang the overheads are generally competitive with (P)NaCl [97]. By implementing SFI in an early intermediate language, CompCertSFI can take advantage of all the compiler's optimizations as well as the alignment analysis added by the authors.

Our implementation strategy is different and is not targeted specifically at SFI, but instead at being able to take advantage

https://t.me/learningnets

of hardware features for compartment isolation such as capabilities [15, 22, 94] or the recently proposed support for Hardware Fault Isolation [67]. Another difference is that SECOMP supports an arbitrary number of untrusted mutually-distrustful compartments that can interact by calling each other according to clearly specified interfaces. Finally, while one could potentially extend CompCertSFI to achieve a security notion similar to the original RSP [9, 10, 74, 75] (so without mutual distrust), this would require more work, for instance proving compiler correctness with respect to the semantics of source programs, bridging the gap between the memory model used by CompCert and the single memory block model used by CompCertSFI, and devising a verified back-translation between Cminor and higher CompCert languages like C or Clight.

Another verified SFI compiler is vWasm [27], for which the authors proved in F* [82] that Wasm code compiled to x86 can only interact with its host environment via an explicitly provided API. While this security guarantee and interaction model is similar to that of CompCertSFI, the vWasm implementation doesn't take advantage of all standard compiler optimizations, which leads to some performance loss. The security guarantee only talks about the x86 semantics and, as opposed to our work, does not aim at providing source-level security reasoning, even at the Wasm level.

Another realistic work in this space is that of Derakhshan *et al.* [32], who devise a methodology to break up Trusted Execution Environment (TEE) software into concurrently executing C compartments (überobjects [89]) whose security is compositionally verified using semi-automatic tools (Frama-C [21]) and which are then correctly compiled using a verified compiler (CASCompCert [48]). The formalization of this work is done on paper and the main assumption is that all C compartments are verified, which seems realistic only the for small, highly privileged pieces of code, like the TEEs this work considers. Our focus is instead on machine-checked proofs and on compartmentalized C code that can't be formally verified to be even free of undefined behaviors.

Our work targets a variant of RSP, but such preservation of property classes against adversarial contexts is not the only kind of formally secure compilation. Another important kind aims at preserving specific noninterference properties against passive side-channel attackers. For instance, preservation of cryptographic constant time was proved for both the CompCert [19] and Jasmin [14, 20] verified compilers. Another example is guaranteeing that protection against memory probing is preserved by CompCert [25].

Other formal verification work looks at security of low-level enforcement mechanisms, without involving a compiler from a higher-level language. For instance, SFI mechanisms for both x86 [65] and ARM [99] were proved correct in a proof assistant with respect to the semantics of these complex architectures. In these works communication between low-level compartments is done by jumping to a specified set of entry points, while we consider a more structured model that also enforces the correct return discipline. Other work in this space looks at the basic security properties of capability machines, from simpler ones [45, 81] to more realistic ones like CHERI [69] and Arm Morello [22].

11 Future Work

Building and verifying lower-level backends. Like CompCert's correctness proofs, the SECOMP security proofs currently stop at CompCert's RISC-V assembly language. We extended this language with the abstraction of isolated compartments, which formally defines *what* compartment isolation enforcement should do, but which leaves the *how* to lower-level enforcement mechanisms. Beyond the two capability backends (one already implemented and the other just planned) discussed in §9, various other enforcement mechanisms should be possible, including SFI [49, 55, 83, 91, 97] and tagged architectures [18, 35], as shown in a much simpler setting by Abate *et al.* [8]. Moreover, WebAssembly components [2, 90] could also be a good target for such a backend [27, 44, 55, 96].

At the moment the existing lower-level backends are all unverified. Extending the secure compilation proofs down to cover them is a formidable research challenge that we leave as future work. All existing secure compilation proof techniques in this space [10, 71], including the one we use in the current paper [8], have their origin in proof techniques for full abstraction [71]. But once the memory layout becomes concrete [92, 93], we can no longer hide all information about the compartments' code, as would be needed for full abstraction (or in our case for recomposition), so new proof techniques will be needed for proving these backends secure.

Pointer passing and memory sharing. As with mainstream compartment isolation mechanisms (e.g., SFI or OS processes), we assume that compartments can only communicate via scalar values, but cannot pass each other pointers to share memory. While secure pointer passing is possible to implement efficiently on a capability machine like CHERI [94] or on the micro-policies tagged architecture [18] and this would allow a more efficient interaction model that is also natural for C programmers, the main challenge one still has to overcome is *proving* secure compilation at scale in the presence of such fine-grained, dynamic memory sharing.

Recent work by El-Korashy *et al.* [36, 37] in a much simpler setting shows that it is indeed possible to prove in Coq [36] the security of an extension of Abate *et al.*'s [8] compiler that allows passing secure pointers (e.g., capabilities) between compartments. With such fine-grained memory sharing, however, proofs become more challenging and the proof technique of El-Korashy *et al.* [36] led to much larger proofs and still has conceptual limitations that one would need to overcome for it to work for CompCert, in particular for supporting memory injections. In fact, even extending CompCert's compiler correctness proof to passing arbitrary pointers seems a challenge, since it would imply a significant change to CompCert's trace model. In the nearer future we will try to allow more limited forms of memory sharing between compartments, for instance of statically allocated buffers, which could be passed without significantly changing CompCert's trace model.

From safety to hypersafety. Another interesting direction is extending SECOMP to stronger criteria beyond robust preservation of safety, in particular to hypersafety [10], such as data confidentiality. We expect that SECOMP can be easily adapted to these stronger criteria, by for instance always clearing registers before changing compartments, and also that our proof technique can still apply, by only extending the back-translation step to take finite sets of trace prefixes as input [10, 85]. The challenge is actually enforcing robust preservation of hypersafety in the lower-level backends, especially with respect to side-channel attacks, including devastating micro-architectural attacks like Spectre.

https://t.me/learningnets

Dynamic compartment creation and dynamic privileges. SECOMP uses a static notion of compartments and static interfaces to restrict their privileges. SECOMP compartments are defined statically by the source program, so are a form of code-based compartmentalization. In the future one could also explore dynamic compartment creation, which would allow for data-based compartmentalization [43], e.g., one compartment per incoming network connection or one compartment per web browser tab or plugin [76]. It would also be interesting to investigate dynamic privileges for compartments, e.g., dynamically sharing memory by passing secure pointers (as discussed above), dynamically changing the compartment interfaces [66], or history-based access control [3, 17].

Acknowledgments. We thank Adrien Durier for participating in early discussions about this work. We are also grateful to the anonymous reviewers at PriSC'23 for their helpful feedback. This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), and by the Deutsche Forschungsgemeinschaft (DFG) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972.

A Calls and returns in SECOMP

Consider the example depicted in Figure 5: during the execution of a function C.f, a call instruction to C′.g is encountered. The compartmentalized semantics first compare the source and destination of the call. If the call is an internal call (bottom arrow), then the state transitions to a call-state of that compartment without generating an event. If the call is a cross-compartment call, then the semantics perform an additional check: if the call is allowed, then the state transitions to a call-state, but this time generating a event recording the call (top arrow). Otherwise, the execution must stop.

From this call-state, there are two possibilities: If g is not a system call, the call-state transitions silently to a standard state of C′ whose execution continues until reaching a return instruction, upon which the standard state silently transitions to a return-state preparing to return to C. If g is a system call, then the call-state directly transitions to a return-state preparing to return to C, according to the semantics of the system call. Now, from the return-state, and similarly to the transition to the call-state, we must again check whether the return is allowed, and transition to the state of C if it is the case. Note that in the example above, it is possible for C′ to return to C from a different function g′. This may happen when, internally, g performs a tailcall to g′.

B Background: memory injections

Memory injections [61] are a technical tool introduced by CompCert in order to deal with complex program transformations that can change the memory layout of the program. A memory injection is a partial function from block identifiers to pairs of block identifiers and offsets. Given such a memory injection j, j injects values into other values in the following sense: j injects pointer (b, o) into (b', o') if $j\ b = (b', z)$ and $o' = o + z$; j injects undefined values into any value; and j injects non-pointer values into themselves. Then, one can lift this relation to memories: j injects $\mathtt{m}$ into $\mathtt{m}'$, written $\mathtt{m} \mapsto_j \mathtt{m}'$ if, whenever $j\ b = (b', z)$ then $\mathtt{m}(b, o) = v \implies \mathtt{m}'(b', o + z) = v'$ with j injecting v into v'.

Memory injections are at the core of CompCert and satisfy a number of useful properties, in particular commutation with memory operations.

C Axiomatization of system calls

The formal definition of the CompCert axiom (extended with compartments) we use in back-translation is the following:

Axiom 1 (Commutativity of system calls and memory injections)**.** Given two environments $\mathtt{ge}_1$ and $\mathtt{ge}_2$, two memories $\mathtt{m}_1$ and $\mathtt{m}_2$ and a memory injection j that relates $\mathtt{m}_1$ and $\mathtt{m}_2$ (written $\mathtt{m}_1 \mapsto_j \mathtt{m}_2$) and that relates the symbols of the current compartment in the environments $\mathtt{ge}_1$ and $\mathtt{ge}_2$, then if a system call ef with arguments args produces a trace t in $\mathtt{ge}_1$ and $\mathtt{m}_1$, resulting in return value res and new memory $\mathtt{m}_1'$,

$$\mathtt{ge}_1, \mathtt{m}_1 \vdash \mathtt{ef}\ \mathtt{args} \xrightarrow{t} (\mathtt{res}, \mathtt{m}_1')$$

then:

$$\exists\ j'\ \mathtt{res}'\ \mathtt{m}_2'.\ \mathtt{ge}_2, \mathtt{m}_2 \vdash \mathtt{ef}\ \mathtt{args} \xrightarrow{t} (\mathtt{res}', \mathtt{m}_2')$$

with $\mathtt{m}_1' \mapsto_{j'} \mathtt{m}_2'$.

D Definition of the memory deltas

Formally, the memory deltas are defined as follows:

$$\begin{aligned} \delta := \quad & \mid \mathtt{delta\_storev}\ \mathtt{ch}\ \mathtt{p}\ \mathtt{v}\ \mathtt{C} \\ & \mid \mathtt{delta\_store}\ \mathtt{ch}\ \mathtt{b}\ \mathtt{o}\ \mathtt{v}\ \mathtt{C} \\ & \mid \mathtt{delta\_bytes}\ \mathtt{b}\ \mathtt{o}\ \overline{\mathtt{v}}\ \mathtt{C} \\ & \mid \mathtt{delta\_alloc}\ \mathtt{C}\ \mathtt{l}\ \mathtt{h} \\ & \mid \mathtt{delta\_free}\ \mathtt{b}\ \mathtt{l}\ \mathtt{h}\ \mathtt{C} \end{aligned}$$

where stores, storevs and bytes are three different kind of stores, while allocs and frees corresponds to the same memory operations.

E Blame Proof Details

The simulation invariants again combine *strong* and *weak* relations, both parameterized by a *partial memory injection* that is built throughout the runs. The strong relation holds when the shared context part is in charge of lockstep execution in both programs, and relates the various *state components* (memories, continuations, local environments, call arguments and return values) according to the injection. The weak relation, in turn, holds when each program part is in charge of execution in its own run. During this time, it relates the *context-side parts* of both memories and continuations, but ignores differences between the two running program sides.

In both cases, a partial memory injection records a mapping between the memory blocks of both executions, but disregarding most of the contents of the program side, only recording the correspondence between the *public symbols* of both programs. As noted, this partial injection does not remain constant across the runs, but needs to be updated as relevant blocks are newly allocated.

Let $\mathcal{R}_j$ denote the combined strong-weak simulation relation under injection j. The simulation invariants again combine *strong* and *weak* relations, both parameterized by a *partial memory injection* that is built throughout the runs. The strong relation holds when the shared context part is in charge of lockstep execution in both programs, and relates the various *state components* (memories, continuations, local environments, call arguments and return values) according to the injection. The weak relation, in turn, holds when each program part is in charge of execution in its own run.

https://t.me/learningnets

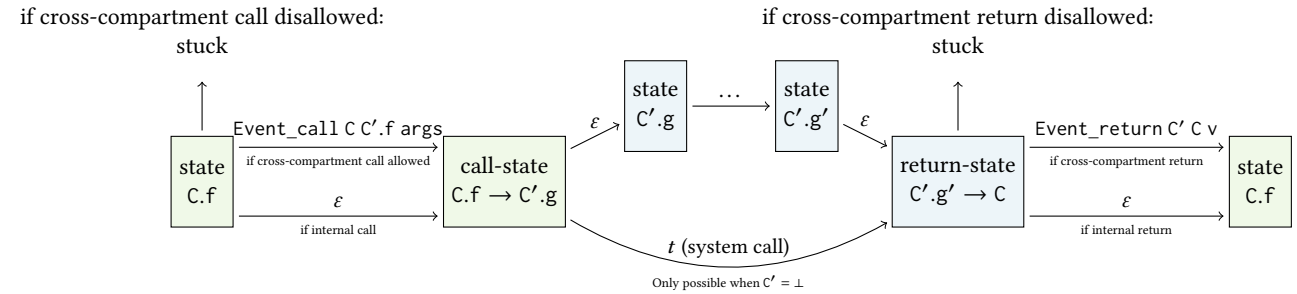


Figure 5: Call from compartment C (light green states) to compartment C′ (light blue states), and the corresponding return.

During this time, it relates the *context-side parts* of both memories and continuations, but ignores differences between the two running program sides.

In both cases, a partial memory injection records a mapping between the memory blocks of both executions, but disregarding most of the contents of the program side, only recording the correspondence between the *public symbols* of both programs. As noted, this partial injection does not remain constant across the runs, but needs to be updated as relevant blocks are newly allocated.

Let $\mathcal{R}_j$ denote the combined strong-weak simulation relation under injection j. The simulation invariants again combine *strong* and *weak* relations, both parameterized by a *partial memory injection* that is built throughout the runs. The strong relation holds when the shared context part is in charge of lockstep execution in both programs, and relates the various *state components* (memories, continuations, local environments, call arguments and return values) according to the injection. The weak relation, in turn, holds when each program part is in charge of execution in its own run. During this time, it relates the *context-side parts* of both memories and continuations, but ignores differences between the two running program sides.

In both cases, a partial memory injection records a mapping between the memory blocks of both executions, but disregarding most of the contents of the program side, only recording the correspondence between the *public symbols* of both programs. As noted, this partial injection does not remain constant across the runs, but needs to be updated as relevant blocks are newly allocated.

Let $\mathcal{R}_j$ denote the combined strong-weak simulation relation under injection j.

At the most basic level, we employ three properties based on individual Clight steps.

(1) $\forall\ j\ s_1\ s_1'\ s_2\ s_2'\ a.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \in K \implies s_1 \xrightarrow{a} s_1' \implies s_2 \xrightarrow{a} s_2' \implies \exists j'.\ \mathcal{R}_{j'}\ s_1'\ s_2'$

(2) $\forall\ j\ s_1\ s_1'\ s_2\ s_2'\ a.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \notin K \implies s_1 \xrightarrow{a} s_1' \implies s_2 \xrightarrow{a} s_2' \implies \exists j'.\ \mathcal{R}_{j'}\ s_1'\ s_2'$

(3) $\forall\ j\ s_1\ s_1'\ s_2.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \notin K \implies s_1 \xrightarrow{\epsilon} s_1' \implies \exists j'.\ \mathcal{R}_{j'}\ s_1'\ s_2$

Properties (1) and (2) state that the relation $\mathcal{R}$ is preserved when *each* of the two programs takes a *single step*, each producing the *same trace*—respectively, from the context side or from the program side. Property (3) further shows that $\mathcal{R}$ is preserved when just *one* of the programs takes a single *silent step* from the *program side* while the other stays put.

(1) $\mathcal{R}$ is preserved when *each* of the two programs takes a *single step* from the *context side*, each producing the *same trace*.

(2) $\mathcal{R}$ is preserved when each of the two programs takes a single step from the *program side*, each producing the same trace.

(3) $\mathcal{R}$ is preserved when *one* of the programs takes a single *silent step* from the *program side* while the other stays put.

(1) $\forall\ j\ s_1\ s_1'\ s_1''\ s_2\ s_2'\ s_2''\ a.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \in K \implies s_1 \xrightarrow{\epsilon}{}^{*} s_1' \implies s_1' \xrightarrow{a} s_1'' \implies s_2 \xrightarrow{\epsilon}{}^{*} s_2' \implies s_2' \xrightarrow{a} s_2'' \implies \exists j'.\ \mathcal{R}_{j'}\ s_1''\ s_2''$

(2) $\forall\ j\ s_1\ s_1'\ s_1''\ s_2\ s_2'\ s_2''\ a.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \notin K \implies s_1 \xrightarrow{\epsilon}{}^{*} s_1' \implies s_1' \xrightarrow{a} s_1'' \implies s_2 \xrightarrow{\epsilon}{}^{*} s_2' \implies s_2' \xrightarrow{a} s_2'' \implies \exists j'.\ \mathcal{R}_{j'}\ s_1''\ s_2''$

(3) $\forall\ j\ s_1\ s_1''\ s_2\ s_2''\ t\ t_1\ t_2.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \xrightarrow{t\cdot t_1}{}^{*} s_1'' \implies s_2 \xrightarrow{t\cdot t_2}{}^{*} s_2'' \implies \exists j'\ s_1'\ s_2'.\ s_1 \xrightarrow{t}{}^{*} s_1' \wedge s_1' \xrightarrow{t_1}{}^{*} s_1'' \wedge s_2 \xrightarrow{t}{}^{*} s_2' \wedge s_2' \xrightarrow{t_2}{}^{*} s_2'' \wedge \mathcal{R}_{j'}\ s_1'\ s_2'$

(1) $\mathcal{R}$ is preserved when *each* of the two programs takes an arbitrary sequence of *silent steps* from the *context side*, followed by a pair of *synchronous steps* producing the *same event*. *same trace*.

(2) $\mathcal{R}$ is preserved when each of the two programs takes an arbitrary sequence of silent steps from the *program side*, followed by a pair of synchronous steps producing the same event. *same trace*.

(3) $\mathcal{R}$ is preserved after any *shared trace prefix* produced by a pair of executions of the two programs.

(1) $\forall\ j\ s_1\ s_1'\ s_2\ s_2'\ t\ t'.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \xrightarrow{t\cdot t'}{}^{*} s_1' \implies s_2 \xrightarrow{t}{}^{*} s_2' \implies s_1'\text{final} \implies s_2' \not\to \implies s_2' \in K \implies s_2'\text{final}$

(2) $\forall\ j\ s_1\ s_1'\ s_2\ s_2'\ t\ a\ t'.\ \mathcal{R}_j\ s_1\ s_2 \implies s_1 \xrightarrow{t\cdot a::t'}{}^{*} s_1' \implies s_2 \xrightarrow{t}{}^{*} s_2' \implies s_2' \not\to \implies s_2' \notin K$

All our definitions are specialized to work with our compartmentalized version of the Clight language. Program states in Clight always include the current memory and continuation (which abstracts the call stack), with additional information depending on the type of state: standard states add the executing function, the language statement being evaluated and local variable environments; call-states replace those with information about the callee and its arguments; and return-states with information about the return value.

https://t.me/learningnets

References

[1] CheriBSD.
[2] The WebAssembly component model.
[3] M. Abadi and C. Fournet. Access control based on execution history. *NDSS*. The Internet Society, 2003.
[4] M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999.
[5] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.
[6] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2):8, 2012.
[7] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, J. Thibault, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. Extended preprint on arXiv:1802.00588v5, 2019.
[8] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018.
[9] C. Abate, R. Blanco, Ş. Ciobâcă, A. Durier, D. Garg, C. Hriţcu, M. Patrignani, É. Tanter, and J. Thibault. An extended account of trace-relating compiler correctness and secure compilation, 2021.
[10] C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *CSF*, 2019.
[11] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012.
[12] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *ICFP*. 2008.
[13] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011.
[14] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020.
[15] S. Amar, D. Chisnall, T. Chen, N. W. Filardo, B. Laurie, K. Liu, R. Norton, S. W. Moore, Y. Tao, R. N. M. Watson, and H. Xia. CHERIoT: Complete memory safety for embedded devices. To appear at MICRO, 2023.
[16] S. N. Anderson, R. Blanco, L. Lampropoulos, B. C. Pierce, and A. Tolmach. Formalizing stack safety as a security property. *CSF*. 2023.
[17] C.-C. Andrici, Ştefan Ciobâcă, C. Hriţcu, G. Martínez, E. Rivas, Éric Tanter, and T. Winterhalter. Securing verified IO programs against unverified code in F*. To appear at POPL, 2024.
[18] 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.
[19] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020.
[20] G. Barthe, B. Grégoire, V. Laporte, and S. Priya. Structured leakage and applications to cryptographic constant-time and cost. *CCS*. 2021.
[21] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, 2021.
[22] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. M. Watson, and P. Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. *ESOP*. 2022.
[23] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. *ESOP*, 2014.
[24] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke. Compiling sandboxes: Formally verified software fault isolation. *ESOP*, 2019.
[25] F. Besson, A. Dang, and T. P. Jensen. Information-flow preservation in compiler optimisations. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 2019.
[26] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008.
[27] J. Bosamiya, W. S. Lim, and B. Parno. Provably-safe multilingual software sandboxing using WebAssembly. *USENIX Security*. 2022.
[28] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. 2020.
[29] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Securing interruptible enclaved execution on small microprocessors. *ACM Trans. Program. Lang. Syst.*, 43(3):12:1–12:77, 2021.
[30] V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *TCS*, 492:1–39, 2013.
[31] C. Cimpanu. Chrome: 70% of all security bugs are memory safety issues. ZDNet, 2020.
[32] F. Derakhshan, Z. Zhang, A. Vasudevan, and L. Jia. Towards end-to-end verified TEEs via verified interface conformance and certified compilers. *CSF*. 2023.
[33] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL):38:1–38:23, 2018.
[34] D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, fully-abstract compilation by approximate back-translation. *LMCS*, 13(4), 2017.
[35] 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.
[36] A. El-Korashy, R. Blanco, J. Thibault, A. Durier, D. Garg, and C. Hriţcu. SecurePtrs: Proving secure compilation with data-flow back-translation and turn-taking simulation. *CSF*, 2022.
[37] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. *CSF*. 2021.
[38] J. Engelfriet. Determinacy implies (observation equivalence = trace equivalence). *TCS*, 36:21–25, 1985.
[39] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013.
[40] D. Gao. Compartmentalisation models. Principles of Capability Languages workshop, 2024.
[41] A. L. Georges, A. Guéneau, T. V. Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *PACMPL*, 5(POPL):1–30, 2021.
[42] A. L. Georges, A. Trieu, and L. Birkedal. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *PACMPL*, 6(OOPSLA):1–30, 2022.
[43] 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.
[44] 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.
[45] S. Huyghebaert, S. Keuchel, C. D. Roover, and D. Devriese. Formalizing, verifying and applying ISA security guarantees as universal contracts. In W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. 2023.
[46] K. Jacobs, D. Devriese, and A. Timany. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022.
[47] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011.
[48] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng. Towards certified separate compilation for concurrent programs. *PLDI*. 2019.
[49] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. *IEEE S&P*. 2023.
[50] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. SFI safety for native-compiled Wasm. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. 2021.
[51] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016.
[52] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *POPL*, 2016.
[53] D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003.
[54] J. Koenig and Z. Shao. CompCertO: compiling certified open C components. *PLDI*. 2021.
[55] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI. *PACMPL*, 6(POPL):1–30, 2022.
[56] J. Kroll, G. Stewart, and A. Appel. Portable software fault isolation. *CSF*. 2014.
[57] D. Kästner, U. Wünsche, J. Barrho, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. *ERTS*. 2018.
[58] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, 1984.
[59] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
[60] X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.
[61] 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.

https://t.me/learningnets

[62] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat IL, 2019.

[63] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.

[64] D. Monniaux and S. Boulmé. The trusted computing base of the CompCert verified compiler. *ESOP*. 2022.

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

[66] 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.

[67] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. M. Tullsen, and D. Stefan. Going beyond the limits of SFI: flexible and secure hardware-assisted in-process isolation with HFI. *ASPLOS*. 2023.

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

[69] K. Nienhuis, A. Joannou, T. Bauereiss, A. C. J. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. *IEEE S&P*. 2020.

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

[71] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.

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

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

[74] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*, 2017.

[75] M. Patrignani and D. Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1), 2021.

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

[77] M. Sammler, S. Spies, Y. Song, E. D'Osualdo, R. Krebbers, D. Garg, and D. Dreyer. DimSum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7(POPL):775–805, 2023.

[78] L. Skorstengaard, D. Devriese, and L. Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *TOPLAS*, 42(1):5:1–5:53, 2020.

[79] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *JFP*, 31:e9, 2021.

[80] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C. Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL):23:1–23:31, 2020.

[81] T. V. Strydonck, A. L. Georges, A. Guéneau, A. Trieu, A. Timany, F. Piessens, L. Birkedal, and D. Devriese. Proving full-system security properties under multiple attacker models on capability machines. *CSF*. 2022.

[82] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

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

[84] The Chromium Project. Memory safety. chromium.org.

[85] J. Thibault and C. Hriţcu. Nanopass back-translation of multiple traces for secure compilation proofs. PriSC, 2021.

[86] S. Tsampas, D. Devriese, and F. Piessens. Temporal safety for stack allocated memory on capability machines. *CSF*. 2019.

[87] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown. Lightweight, modular verification for WebAssembly-to-Native instruction selection. To appear at ASPLOS'24.

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

[89] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016.

[90] L. Wagner. What is a WebAssembly component (and why?). WebAssembly Workshop (WAW), 2024.

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

[92] Y. Wang, P. Wilke, and Z. Shao. An abstract stack based approach to verified compositional compilation to machine code. *PACMPL*, 3(POPL):62:1–62:30, 2019.

[93] Y. Wang, X. Xu, P. Wilke, and Z. Shao. CompCertELF: verified separate compilation of C programs into ELF object files. *Proc. ACM Program. Lang.*, 4(OOPSLA):197:1–197:28, 2020.

[94] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020.

[95] 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. *S&P*. 2015.

[96] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner. Two mechanisations of webassembly 1.0. *FM*. 2021.

[97] 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.

[98] L. Zhang, Y. Wang, J. Wu, J. Koenig, and Z. Shao. Fully composable and adequate verified compilation with direct refinements between open modules. To appear at POPL, 2024.

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

https://t.me/learningnets