Formally Secure Compilation

Last updated: September 1, 2017

General Information

Advisor: Cătălin Hritcu ⟨catalin.hritcu@inria.fr⟩

Institution: Inria Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Keywords:
- state-of-the-art research in programming languages;
- security foundations and/or building secure systems;
- secure compilation, full abstraction, components;
- dynamic enforcement, tag-based monitoring;
- formal verification in the Coq proof assistant;
- property-based testing (e.g., with QuickChick).

Context

Severe low-level vulnerabilities abound in today’s computer systems, allowing cyber-attackers to remotely gain full control. This happens in part because our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and too often trade off security for efficiency. The semantics of mainstream low-level languages like C is inherently insecure, and even for safer languages, establishing security with respect to a high-level semantics does not guarantee the absence of low-level attacks. Secure compilation using the coarse-grained protection mechanisms provided by mainstream hardware architectures would be too inefficient for most practical scenarios.

SECOMP\(^1\) is a new ERC-funded project aimed at leveraging emerging hardware capabilities for fine-grained protection to build the first, efficient secure compilers for realistic low-level programming languages (the C language, and Low\(^*\) [23] a safe subset of C embedded in F\(^*\) [3, 25] for verification). These compilers will provide a more secure semantics for source programs and will ensure that high-level abstractions cannot be violated even when interacting with untrusted low-level code. To achieve this level of security without sacrificing efficiency, our secure compilers target a tagged architecture [7, 10], which associates a metadata tag to each word and efficiently propagates and checks tags according to software-defined rules. We are using property-based testing and formal verification to provide high confidence that our compilers are indeed secure. Formally, we are constructing machine-checked proofs in Coq of fully abstract compilation and of a new property we call robust compilation, which implies the preservation of trace properties even against an adversarial context. These strong properties complement compiler correctness and ensure that no machine-code attacker can do more harm to securely compiled components than a component already could with respect to a secure source-level semantics.

This document is aimed at briefly presenting various topics from the SECOMP project to which an excellent student or young researcher could contribute (during a research internship, MSc or PhD thesis, PostDoc, etc, see https://secure-compilation.github.io/#positions for details). The list is not exhaustive, and if you are interested you should get in contact and we will together try to find a topic that is in sync with your interests and expertise.

Contents

1 Relating two formal notions of secure compilation: Full Abstraction and Robust Compilation 1

2 Proving secure compilation instances in Coq 2

3 Protecting C components 2

4 Micro-policies for C 3

5 Secure micro-policy composition 3

6 Verified Low\(^*\)-Vale interoperability 4

1 Relating two formal notions of secure compilation: Full Abstraction and Robust Compilation

Secure compilation denotes a class of strong security guarantees that complement compiler correctness [18, 19] and ensure that no low-level attacker can do more harm to securely compiled programs than a safe program in the source language already could. Figure 1 illustrates the intuition behind secure compilation: a compiler chain (i.e., compiler, linker, loader, runtime system, hardware taken as a whole) is secure when for each low-level context attacking a compiled component there exists a high-level context attacking the original component.

\(^1\)http://secure-compilation.github.io/
Proving this requires changing the compilation chain in order to be able to map each low-level distinguishing context to a high-level one, which intuitively means that interaction with low-level contexts is as secure as interaction with high-level contexts.

One way to formalize this intuition is as a property called full abstraction [1]. Formally, full abstraction is phrased as the distinguishability game from Figure 2, stating that low-level and high-level attackers have exactly the same distinguishing power. Since (at least intuitively) full abstraction (and its variants) ensures the preservation of secrecy and integrity properties [13], it has received increasing attention in the past years. Yet full abstraction is very challenging to achieve and prove at scale.

In ongoing work we propose a new notion of secure compilation that we call robust compilation (which others have also informally considered in the past [2, 27]). Instead of phrasing secure compilation in terms of preserving observational equivalence, robust compilation phrases it in terms of preserving trace properties (Figure 3): For any low-level attacker breaking a trace property \(\pi\) when interacting with a compiled component we can construct a safe high-level attacker breaking \(\pi\) when interacting with the original high-level component before compilation. Because it is intuitively weaker, robust compilation is easier to achieve and prove than full abstraction, especially if one wants to scale up to realistic programming languages. Still, robust compilation does capture the preservation of invariants and other integrity properties in the presence of an adversarial context, so it does provide some of the security benefits usually associated with full abstraction. And because it completely leaves out the confidentiality aspects robust compilation should also be unaffected by side-channel attacks that trivially break the full abstraction guarantees.

The main goal for this first topic is to formally relate robust compilation and full abstraction. In particular we expect full abstraction together with some compositional notion of compiler correctness to imply robust compilation. A good proxy for showing this could be the recently introduced alternative characterization of (a variant of) full abstraction as hyperproperty preservation [13]. Moreover, we are also interested in proving that robust compilation is strictly stronger than compositional compiler correctness (e.g., à la CompCert).

2 Proving secure compilation instances in Coq

To put our work on solid formal foundations we will use the Coq proof assistant to prove full abstraction in a machine-checked way. For this topic we will start simple and build a toy fully abstract compiler from a core imperative language with procedures and components to an idealized RISC machine. This compiler will protect each component from the others and we will explore different ways for proving full abstraction for it in Coq, including using interaction trace semantics [14, 15, 22], logical relations [4, 5, 21], and high-level interpreters [12]. Finally, we will investigate vertical composition for secure multi-pass compilers.

3 Protecting C components

At the lowest-level of SECOMP we want to enforce isolation and protect C and assembly components from each other,
providing a strong attacker model of mutual distrust. Mutual distrust is often justified at this level, because neither C nor assembly components are guaranteed to be memory safe, and can thus be taken over by remote attackers. This mutual distrust attacker model was recently formalized as a variant of full abstraction we call securely compartmentalizing compilation [15]. This strong property can be currently provided by process-level sandboxing (e.g., plugins and tabs in modern browsers [24]) and by software fault isolation (SFI; e.g., Google Native Client [28]). We will use our recent work on micro-policies [7, 10]—security monitors based on fine-grained metadata tags—to improve upon these currently-deployed techniques by supporting a more natural and efficient communication model of procedure calls and returns, instead of inter-process communication.

We will devise a more secure semantics for C in which a undefined behavior according to the C standard (e.g., a buffer overflow) in a program component cannot directly affect the other components. For this we will introduce a strong notion of C component that only exposes a typed interface and protects its internal representations and state, and we will enforce dynamically that all components respect the interfaces. For compilation we will modify the CompCert verified C compiler [17, 19] as well as the static linker to propagate the breakup into components and the typing information all the way down to the produced binary. We will also extend the loader to tag the memory region of each component differently, to mark procedure entry points with a tag that includes typing information about the procedure, etc. The changes to the minimalist runtime system of C will be small and only involve changing manual memory management routines like malloc and free to respect components.

Finally, we will use a new and highly non-trivial micro-policy to dynamically enforce compartment isolation [7, 16] as well as the procedure call discipline and type safety on component boundaries. Intuitively, we will tag each component differently and only allow component switching at pre-specified entry points, whose tag will also contain the type of the arguments of the called procedure, which we will dynamically check. We will tag the return address as a linear return capability, which will trigger a dynamic type check of the procedure’s return value when used. While some of this is standard for higher-order contracts [11] and gradual typing [26] in high-level languages, micro-policies allow us to do these checks at the lowest level and much more efficiently. Code pointers will be tagged with a procedure type when crossing component boundaries, and then handled similarly to direct procedure calls when invoked. We have recently started investigating such a micro-policy in the much setting [16], however, formally proving security (e.g., full abstraction or robust compilation) and gradually extending this to C are challenging open problems.

4 Micro-policies for C

We will extend the semantics of C with support for tag-based reference monitoring. These tag-based monitors—i.e., high-level micro-policies—will be written in rule-based domain-specific languages (DSLs) inspired by our rule format for micro-policies monitoring machine code [6, 7, 10]. Some parts of the micro-policy DSLs for C and machine code will be similar: for instance, we want a simple way to define the structure of tags using algebraic datatypes, sets, and maps. The kinds of tags differs from level to level though: at the machine code level we have register, program counter, and memory tags, while in C we could replace register tags with value and procedure tags. The way tags are checked and propagated also differs significantly between levels. At the machine-code level, propagation is done via rules that are invoked on each instruction, while in C we have many different operations that can be monitored, e.g., primitive operations, function calls and returns etc. Moreover, the tags of C values could be propagated automatically as values are copied around, without needing to write explicit rules for that. A continuation of this task would be to translate micro-policies from the C to the machine-code level.

5 Secure micro-policy composition

While very useful in practice, secure composition of micro-policies [7] is difficult, as one policy’s interaction with the code can break another policy’s guarantees. For example, if we wish to enforce information-flow control (IFC) together with other policies, observing the tags of these other policies can reveal sensitive information and thus break the non-interference property established for the IFC micro-policy in isolation. There are several ways in which we plan to approach this problem.

The first builds on the idea of vertical sequential composition: a linear order is carefully chosen in advance between the micro-policies to be composed. One starts with the lowest-level micro-policy and the bare hardware and proves that the policy is correct with respect to a higher-level abstract machine. This higher-level abstract machine virtualizes the tagging mechanisms in the hardware so that the second micro-policy can be implemented on top of this first abstract machine, instead of the bare hardware. We then prove the correctness of this second micro-policy with respect to a second abstract machine, and so on. While this technique is likely to work well for one-off proofs, it is dependent on the set of composed policies and the initially chosen order between them. If we want to add or remove policies we can only do that at the top of the stack; otherwise we have to redo a large number of proofs. We plan to use ideas from monad transformers [20] and algebraic effects [8] to lift this limitation.

The second approach we will consider can be called parallel composition or cross-product composition. The idea is to require each micro-policy to specify what should happen with its own tags on all other micro-policies’ monitor services and error handlers. Basically, all other micro-policies’ routines become “virtual instructions” for each micro-policy machine (thus the name cross-product composition). For instance, the IFC micro-policy could specify that the result of a monitor call returning a value’s tag in another micro-policy is as classified as the original value. For this more flexible kind of composition there is additional verification effort when composing policies, and this effort scales quadratically with the number of composed policies.
6 Verified Low*-Vale interoperability

In another thread of work, we would like to investigate the interoperability between Low* [23], a shallow embedding of a safe subset of C in F*, and Vale [9], a deep embedding of assembly for various architectures in F*. Low* and Vale are targeted at verification. As a first step we want to give a formal semantics to programs mixing Low* and Vale (i.e., C and assembly) and defining what it means for a program combining Low* parts and Vale parts to satisfy an F* specification. This interoperability model will capture in particular the C calling conventions and data representations, as well as various other invariants the Vale code should verifiably respect.

As a second step we want to prove a joint security theorem about a combined program with: (1) Low* parts that achieve secrecy using type abstraction and (2) Vale parts achieves secrecy using taint tracking and that can be given relational specifications in F*. This is a combination of separate theorems we previously proved for complete Low* programs and complete Vale programs.

As a third step we would like to prove that replacing Low* code with Vale code for efficiency is semantically justified. In particular we want to prove that replacing a Low⋆ code with Vale code for efficiency is semantically justified. In

References