Formally Secure Compilation

Last updated: August 2, 2017

General Information

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

Institition: Inria Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Existing skills or strong desire to learn about:

- 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 a proof assistant like Coq (or F⋆).

Context

Severe low-level vulnerabilities abound in today's computer systems, allowing cyber-attackers to remotely gain full control. This happens in big 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.

This new ERC-funded project called SECOMP[1] is 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⋆ a safe subset of C embedded in F⋆ for verification). These compilers will provide a secure semantics for all 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* [6,8], which associates a metadata tag to each word and efficiently propagates and checks tags according to software-defined rules. We are also using property-based testing and formal verification to provide high confidence that our compilers are indeed secure. Formally, we are constructing machine-checked proofs 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 vast SECOMP project to which a 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.

[1] `http://secure-compilation.github.io/`

Contents

1 Proving full abstraction in Coq

Full abstraction [1] is a strong security property that complements compiler correctness [16, 17] and requires enforcing all high-level language abstractions against all low-level attackers. Full abstraction states that no low-level attacker can do more harm to securely compiled programs than a program in the secure source language already could. Figure 1 illustrates the intuition behind full abstraction: a compilation scheme is fully abstract when for each low-level context attacking a compiled component there exists a high-level context attacking the original component. This means that interaction with low-level contexts is as secure as interaction with high-level contexts. 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. Proving this requires being able to map each low-level distinguishing context to a high-level one.

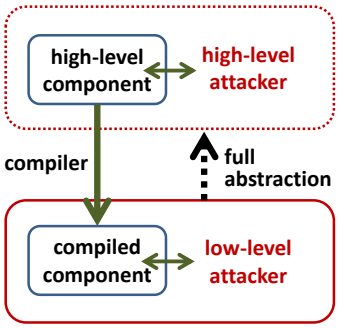

Figure 1: Full abstraction (informally)

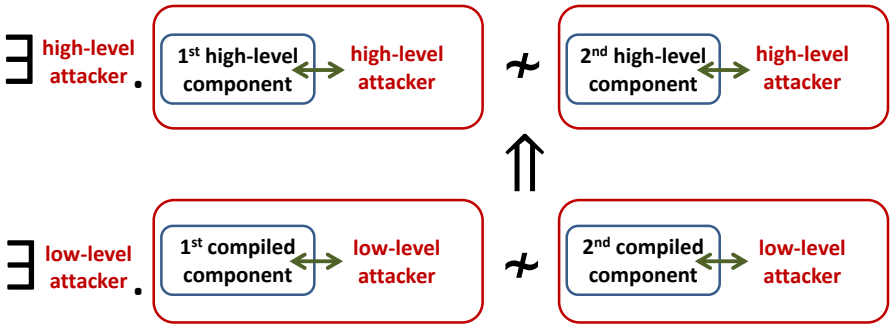

Figure 2: Full abstraction (formally)

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 [12, 13, 20], logical relations [3, 4, 19], and high-level interpreters [10]. Finally, we will investigate vertical composition for secure multi-pass compilers.

2 Formalizing robust compilation

It is folklore that "under reasonable assumptions" fully abstract compilation ensures the preservation of all confidentiality and integrity properties of the source language components. This topic is aimed at formalizing a new property we call *robust compilation* that captures the preservation of some integrity properties, while completely leaving out the confidentiality aspect. While weaker than full abstraction, robust compilation is very valuable, since it seems easier to enforce efficiently and should also be unaffected by side-channel attacks that trivially break the full abstraction guarantees.

While other researchers also seem interested in robust compilation [2,23], giving a good formal definition of this property is an open research problem we will try to solve. Intuitively, robust compilation ensures the preservation of (robust [11]) safety properties in an adversarial low-level context. In particular, we want robust compilation to preserve certain forms of integrity, such as all data invariants proved using a non-relational program logic for robust safety.

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* [13]. This strong property can be currently provided by process-level sandboxing (e.g., plugins and tabs in modern browsers [21]) and by software fault isolation (SFI; e.g., Google Native Client [24]). We will use our recent work on *micro-policies* [6,8]—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 [15, 17] 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 [6, 14] 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 [9] and gradual typing [22] 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 [14], 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 [5, 6, 8]. 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 [6] 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 noninterference 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 [18] and algebraic effects [7] 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 interop

In another thread of work, we would like to investigate the interoperability between Low*, a shallow embedding of a subset of C in F*, and VALE, 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* function with a VALE function preserves correctness and security, provided we prove that the VALE function has the reified version of the Low* function as a specification (for correctness) or as a relational specification (for security).

References

[1] M. Abadi. Protection in programming-language translations. Research Report 154, SRC, 1998.

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

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

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

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

[6] 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.

[7] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *JLAMP*, 84(1):108–123, 2015.

[8] 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.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *ICFP*. 2002.

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

[11] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3-4):435–483, 2004.

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

[13] 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.

[14] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Towards a fully abstract compiler using Micro-Policies: Secure compilation for mutually distrustful components. Technical Report, arXiv:1510.00697, 2015.

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

[16] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. *POPL*. 2014.

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

[18] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. *POPL*. 1995.

[19] M. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. To appear in 21st International Conference on Functional Programming (ICFP 2016).

[20] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42:22–45, 2015.

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

[22] A. Takikawa, D. Feltey, B. Greenman, M. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? *POPL*, 2016.

[23] N. van Ginkel, R. Strackx, J. T. Muehlberg, and F. Piessens. Towards safe enclaves. 4th Workshop on Hot Issues in Security Principles and Trust (HotSpot), 2016.

[24] 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.