Poster: Towards a Fully Abstract Compiler Using Micro-Policies

— Secure Compilation for Mutually Distrustful Components —

Yannis Juglaret, PhD student, Inria
yannis.juglaret@inria.fr

Compiled partial programs evolve within a low-level environment, with which they can interact. Such interaction is useful — think of high-level programs performing low-level library calls, or of a browser interacting with native code that was sent over the internet [19] — but also dangerous: parts of the environment could be malicious or compromised and try to compromise the program as well [7, 19]. Components written in unsafe languages such as C and C++ can be vulnerable to control hijacking attacks [7, 18] and be taken over by a remote attacker. When the environment can't be trusted, it is a major concern to ensure the security of running programs.

With today's compilers, low-level attackers [7] can circumvent high-level abstractions [1, 13] and are thus much more powerful than high-level attackers, which means that the security reasoning has to be done at the lowest level, which it is extremely difficult. An alternative is to build a *secure compiler* that ensures that low- and high-level attackers have exactly the same power, allowing for easier, source-level security reasoning [2, 9, 10, 16]. Formally, the notion of secure compilation is usually expressed as *full abstraction* of the translation [1]. Full abstraction is a much stronger property than just compiler correctness [14].

For a compiler targeting machine code, which lacks structure and checks, a typical low-level attacker has write access to the whole memory, and can redirect control flow to any location in memory [7]. Techniques have been developed to deal with such powerful attackers, in particular involving randomization [2] and binary code rewriting [8]. The first ones only offer weak probabilistic guarantees, while the second ones add extra software checks which often come at a high performance cost. Using additional protection in the hardware can result in secure compilers with strong guarantees [16], without sacrificing efficiency or transparency. In our work, we use a generic tag-based protection mechanism called *micro-policies* [5, 6] as the target of a secure compiler.

Micro-policies provide instruction-level monitoring based on fine-grained metadata tags. In a micro-policy machine, every word of data is augmented with a word-sized tag, and a hardware-accelerated monitor propagates these tags every time a machine instruction gets executed. Micro-policies can be described as a combination of software-defined rules and monitor services. The rules define how the monitor will perform tag propagation instruction-wise, while the services allow for direct interaction between the running code and the monitor. This mechanism comes with an efficient hardware implementation built on top of a RISC processor [6] as well as a mechanized metatheory [5], and has already been used to enforce a variety of security policies [5, 6].

Recent work [4, 16] has illustrated how *protected module architectures* — a class of hardware architectures featuring coarse-grained isolation mechanisms — can help in devising a fully abstract compilation scheme for a Java-like language. This scheme assumes the compiler knows which components in the program can be trusted and which ones cannot, and protects the trusted components from the distrusted ones by isolating them in a protected module. This kind of protection is only appropriate when all the components we want to protect can be trusted, for example because they have been verified [3]. Accounting for the cases in which this is not possible, we propose a stronger attacker model of *mutual distrust* [12]: in this setting a secure compiler should protect each component from every other component, so that whatever the compromise scenario may be, uncompromised components always get protected from the compromised ones.

This new attacker model for secure compilation extends the well-known notion of full abstraction to ensure protection for mutually distrustful components. We devised a secure compilation solution for a simple object-oriented language that defends against this strong attacker model [11, 12]. Our solution includes a simple compiler chain (compiler, linker, and loader) and a novel micro-policy that protects the ab-

stractions of our simple language—class isolation, the method call discipline, and type safety—against arbitrary low-level attackers. Enforcing a method call discipline and type safety using a micro-policy is novel and constitutes a contribution of independent interest. We have started proving that our compiler is secure, and we have good hopes in the efficiency and transparency of our solution for the protection of realistic programs. We also have ideas for mitigation when our mechanism is not transparent enough.

In independent parallel work [15, 17], Patrignani et al. are trying to extend previous results [16] to support mutual distrust using different mechanisms (e.g. multiple protected modules and randomization).

References

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

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

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

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

[5] 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. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. 2015.

[6] 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.

[7] Ú. Erlingsson. Low-level software security: Attacks and defenses. In *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, 2007.

[8] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. *OSDI*. 2006.

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

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

[11] Y. Juglaret and C. Hriţcu. Secure compilation using micro-policies (extended abstract). Workshop on Foundations of Computer Security, 2015.

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

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

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

[15] M. Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, 2015.

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

[17] M. Patrignani, D. Devriese, and F. Piessens. Multi-module fully abstract compilation (extended abstract). Workshop on Foundations of Computer Security, 2015.

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

[19] 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.