Multi-Module Fully Abstract Compilation (Extended Abstract)

Marco Patrignani, Dominique Devriese, Frank Piessens
iMinds-DistriNet, KU Leuven, Belgium

High-level languages like Java or ML support abstraction and data encapsulation through language features such as modules, objects, classes, and/or abstract data types. But traditional compilation does not preserve such abstraction boundaries. At machine code level, there is just a single address-space where all code is readable and all data is read/writable. In other words, the entire high-level program is compiled down into one single protection domain. To a large extent, this is the case because the protection domain granularity of modern execution platforms is very coarse grained: the smallest unit of protection is an operating system *process*, and most programs are compiled to a single process.

For fully safe languages, and if attackers can only provide input to and read output from programs, there is no need to preserve abstraction or protection boundaries after compilation: for such attackers, language safety is sufficient to guarantee that program abstractions are maintained. However, most compiled languages (including for instance C#, Java, Go, and Rust) are *not* fully safe: programs can contain unsafe blocks that might be subject to memory safety errors [1], or programs can interface with code written in unsafe languages through a native interface. In addition, attackers may have more powers than just the abilities to provide input and read output: for instance, programs might support binary plugins making it possible for attackers to load arbitrary machine code into a process, or kernel-level malware can inspect any user process at the machine code abstraction level. In these circumstances, mechanisms for protecting source code data encapsulation and abstractions even after compilation to machine code level are a valuable additional layer of defense.

Fortunately, in recent years there has been renewed interest in execution platforms that support fine-grained protection domains. Notable examples include protected module architectures [2], [3], [4], capability-enhanced processors [5], or general meta-data tracking processors [6]. The upcoming Intel Software Guard eXtensions (SGX) [7] will bring such fine-grained protection to mainstream processors. With the availability of fine-grained protection domains at machine code level, researchers have started exploring compilation techniques that build on such low-level protection to maintain source code level abstractions at machine code level, even in the presence of attackers that can perform machine code injection attacks.

The correctness criterion for such compilation is the notion of *full abstraction* [8]. Roughly speaking, compilation is fully abstract if any attack that an arbitrary machine code context can do against a compiled module, could also be done by a source code context of that module based on the source code semantics. More precisely, the compiler must preserve and reflect *contextual equivalence* between its source and target languages. Two modules M_1 and M_2 are contextually equivalent $M_1 \sim M_2$ if there is no context C that can distinguish M_1 and M_2 in the sense that $C[M_1]$ diverges and $C[M_2]$ does not. Compilation is *fully abstract* if it holds that: $M_1 \sim M_2$ iff $[\![M_1]\!] \sim [\![M_2]\!]$ with $[\![M]\!]$ the compilation of M.

In previous work [9], [10], we have proposed fully abstract compilation techniques for Java-like languages towards an execution platform with a protected module architecture that supports a *single* protected module living in an unprotected machine code context. One specific source code module M is compiled to a single machine code level protected module $[\![M]\!]$ with the guarantee that machine code contexts can only perform attacks against $[\![M]\!]$ that could also be done by means of a source code context against M. As a consequence, this single source code module M is protected against code injection attacks originating from outside that module.

This paper reports on our ongoing work to generalize this to compilation schemes that support compilation to multiple protected modules. Such a generalization is *useful*. The limitation to one protected module at run time makes sense in the case of a safe source level programming language, where only one module (the run time library) contains native code that could have memory safety vulnerabilities. In this case, all the safe source code modules are compiled together in the single target platform protected module. But as soon as an application has more than one module that could potentially contain safety vulnerabilities (or alternatively if some modules could have been subject to tampering with the compiled version of the module), the limitation to one target platform protected module is unsatisfactory. To protect module M_1 against exploitation of safety vulnerabilities within module M_2 at run time, $[\![M_1]\!]$ must be in a different protected module than $[\![M_2]\!]$. Our generalization makes it possible to compile *each* source code module to a corresponding target platform protected module, thus limiting the impact of code injection attacks against each individual module.

It is also a *non-trivial* generalization, as malicious machine code contexts can now try to intervene in the interactions between two protected modules. A concrete example of where the existing compilation schemes [10] fail, is the way in which object sharing is handled. Patrignani et al.'s compilation scheme protects against the context guessing private object identities by maintaining a table of all objects that have been shared with the context, and using an index into that table as the identity of an object outside the protected module. This protection is insufficient in the multi-module case, as it does not protect against a scenario where module M_1 shares an object O with M_2 but not with M_3. $[\![M_3]\!]$ can now guess the object identity of O and call methods on it, thus breaking full abstraction.

In this work-in-progress, we investigate the additional complications that arise for multi-module fully abstract compilation, show how some of the proposed execution platforms with fine-grained protection lack features needed to support such compilation, and develop two approaches where the first one achieves a form of probabilistic full abstraction on SGX-like systems, and the other one achieves full abstraction on a specific variant of capability-enhanced hardware.

REFERENCES

[1] Y. Younan, W. Joosen, and F. Piessens, “Runtime countermeasures for code injection attacks against C and C++ programs,” *ACM Computing Surveys*, vol. 44, no. 3, pp. 17:1–17:28, June 2012. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/288462

[2] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: an execution infrastructure for TCB minimization,” in *Eurosys*, 2008.

[3] R. Strackx and F. Piessens, “Fides: Selectively hardening software application components against kernel-level or process-level malware,” in *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012).* ACM Press, October 2012, pp. 2–13.

[4] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base,” in *USENIX Security*, 2013.

[5] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The CHERI capability model: Revisiting RISC in an age of risk,” in *ISCA*, 2014.

[6] U. Dhawan, C. Hritcu, 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,” in *ASPLOS*, 2015.

[7] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *HASP*, 2013.

[8] M. Abadi, “Protection in programming-language translations,” in *ICALP*, 1998.

[9] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, “Secure compilation to modern processors,” in *2012 IEEE 25th Computer Security Foundations Symposium (CSF 2012).* IEEE, August 2012, pp. 171–185.

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