Software Assurance Tips

A product of the Software Assurance Tips Team[5]

Generated Thursday 10th June, 2021

Richard Ford

Monday 14th June, 2021

1 Secure Compilation

Ordinary compilers aim to provide correct compilation based on the rules of the source language. Most language standards allow the compiler to perform optimizations provided they follow the "as if" rule: that the external interactions of the program are the same as if the compiler had not optimized but rigidly followed the language abstract model of execution. In addition, refinement is allowed. The source language may allow several different ways that the program may be executed. The compiler is considered correct if the target code behaves in one of those ways.

Security problems arise because attackers do not play by the source language rules.[7, 4]

- A compromised operating system can inspect the memory of a running program. So, memory locations formerly occupied by out-of-scope function variables can be inspected to see the possibly sensitive data they held.
- The timing behavior of the computation can be used to leak sensitive information.

The emerging field of secure compilation seeks to define what it means for compilation to be secure.[2, 6] There is much ongoing research on secure compilation.

Whereas most compiler correctness research has been on whole-program compilation, the secure compilation studies use program fragments (called components) which are linked with a context. The context is used to model attackers. The behavior of a component is characterized by the set of traces (sequences of events) it may produce. Two components are considered contextually equivalent if, when combined with the same arbitrary context, they produced the same trace.

A key concept mentioned in many of the papers is that of fully-abstract compilation. A compilation is fully abstract if it has two properties:

1. *Contextually equivalence preservation:* if two components are contextually equivalent at the source level, then their compiled forms are also contextually equivalent.
2. *Contextually equivalent reflection:* if the compiled form of two components are contextually equivalent, then they are also contextually equivalent at the source.

Because fully abstract compilation involves quantification over contexts, it is difficult to prove. An alternate approach is to focus on properties and hyperproperties.[3]

A program property can be represented by the set of traces that satisfy that property. The language semantics specify the set of allowable traces. A program will satisfy a property if its set of traces (as defined by the language) is a subset of the property's traces.

A correct compiler, because of refinement or optimization, is allowed to produce a target program which will only produce a subset of those traces. But if the source program satisfied the property, the target program will also (assuming no attacks).

Whereas a property is characterized by a set of traces, a hyperproperty is a set of sets of traces. Hyperproperties can express requirements that cannot be expressed using only properties. For example, non-interference can be expressed by saying that for every trace involving a mix of low and high inputs and outputs there is another trace with the high inputs and outputs deleted, that has the same low inputs and outputs (thus the low inputs and outputs are not influenced by the high inputs or outputs).

The goal of secure compilation is not to remove security vulnerabilities from source programs, but to ensure that no vulnerabilities are introduced. Secure compilation, whether to achieve full abstraction or to preserve some specified properties in the face of attacks is not possible unless the target language has some means of protection. In particular, protecting data-in-use requires hardware protection mechanism. Some example hardware mechanisms that have been studied in regard to secure compilation are:[2, 6]

- Typed-assembly language
- Capability Machines (CHERI)
- Protected Module Architectures (PMA): e.g., SGX

- Micro-policies (CRASH/SAFE, or Dover CoreGuard)

Most of the secure compilation theories have been based on trace events that are the same for source and target language. But recent work has generalized this theory to use relations to relate source and target event traces.[1]

Current software development practice targets platforms that do not have the required hardware protection mechanisms and uses compilers that strive to be correct but that are unaware of the additional security requirements. Here are some possible recommendations to avoid vulnerabilities:

- Promote use of hardware protections that are currently available (e.g., SGX and CoreGuard) but not currently being used.
- Formulate and propose security contracts for programming languages and implement them in some open-source compilers. For example, an annotation could be added to the declaration of sensitive data. The compiler could then erase the data as soon as it goes out of scope.
- Use software scanners to detect vulnerabilities and try to use existing compiler features to minimize exposure.

References

[1] Carmine Abate et al. *Trace-Relating Compiler Correctness and Secure Compilation*. 2020. arXiv: 1907.05320 [cs.PL].

[2] Matteo Busi and Letterio Galletta. "A brief tour of formally secure compilation". In: *3rd Italian Conference on Cyber Security, ITASEC 2019*. Vol. 2315. CEUR-WS. 2019.

[3] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *21st IEEE Computer Security Foundations Symposium*. June 2008, pp. 51–65.

[4] Vijay D'Silva, Mathias Payer, and Dawn Song. "The Correctness-Security Gap in Compiler Optimization". In: *2015 IEEE Security and Privacy Workshops*. 2015, pp. 73–87. DOI: 10.1109/SPW.2015.33.

[5] Jon Hood, ed. *SwATips*. https://www.SwATips.com/.

[6] Marco Patrignani, Amal Ahmed, and Dave Clarke. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". In: *ACM Comput. Surv.* 51.6 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3280984. URL: https://doi.org/10.1145/3280984.

[7] Huzaifa Sidhpurwala. "Security flaws caused by compiler optimizations". In: *Red Hat Blog* (2019). URL: https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations.