Software Fault Isolation for Robust Compilation

Ana Nora Evans*
AnaNEvans@virginia.edu
University of Virginia

Memory corruption vulnerabilities are endemic to unsafe languages, such as C, and they can even be found in safe languages that themselves are implemented in unsafe languages or linked with libraries implemented in unsafe languages. *Robust compilation* mitigates the threat of linking with memory-unsafe libraries. The source language is a C-like language, enriched with a notion of a component which encapsulates data and code, exposing functionality through well-defined interfaces. Robust compilation defines what security properties a component still has, even, if one or more components are compromised. The main contribution of this work is to demonstrate that the compartmentalization necessary for a compiler that has the robust compilation property can be realized on a basic RISC processor using software fault isolation.

1 Problem and Motivation

Formal definitions of secure compilation have been proposed by Juglaret et al. [6] and, more recently, by Garg et al. [4]. This work is part of the effort to propose a new definition for robust compilation of unsafe low-level languages [3]. A compiler has the robust compilation property if any attack on a compiled variant of a program (a set of components) that can be mounted by a component linked and executed with it, can also be mounted at the source level by a component. In the source level semantics, it is impossible to write in another's component memory and only procedures exported by the callee and imported by the caller can be called. Thus, for the robust compilation property to hold, a strong machine-level separation of the compiled program and the target context is necessary. Juglaret et al.'s [7] implementation targeted a micro-policy architecture [2] with special tagging capabilities at the level of memory location. This work focuses on supporting the new definition of secure compilation on a generic RISC processor, without specialized hardware. We use software fault isolation [13] mechanisms to provide a proof-of-concept implementation of a compiler back-end to a basic RISC machine.

2 Background and Related Work

Software fault isolation was proposed in 1993 by Wahbe et al. [13]. A distrusted module is sandboxed into its own fault domain, a logical region of the address space. To prevent it from modifying data or executing code belonging to the rest of the application, its object code is instrumented. The physical address is split logically into a segment id and offset, and the introduced instrumentation does not allow writes outside the data domain and execution to escape the code domain, other than predefined exit points. Many applications that use software fault isolation followed. Google's Native Client [14] uses software fault isolation to sandbox C/C++ code in the Chrome web browser. Morrissett et al. [10] proposed a semantics of the x86 architecture and constructed a machine verified checker of Native Client. ARMor [15] is a machine verified system that uses software isolation to sandbox application code running on embedded processors. In this research, we combine ideas from this previous work and apply them to support robust compilation on a processor without specialized hardware.

Abadi [1] defined full abstraction as the property of a compiler to preserve and reflect observational equivalence. Achieving observational equivalence in the presence of side channels such as timing, is impossible. Instead, robust compilation focuses on only mapping back to the source level a context that induces a certain behavior on a program. The robust compilation property for unsafe languages proposed by Fachini et al. [3] is:

$$\forall P\ C_T\ t.\ C_T \bowtie (P{\downarrow}) \Downarrow t \Rightarrow \exists C_S\ t'.\ C_S \bowtie P \Downarrow t' \wedge t' \preccurlyeq_P t$$

That is, for all source-level programs P and all low-level contexts C_T there exists a source-level context C_S, with no undefined behavior, such that the low-level trace t of compiled P linked with C_T and source-level trace t' of P linked with C_S, match up to an undefined behavior in P.

3 Approach and Uniqueness

The work presented in this abstract is part of a project [3] that aims at defining a new security property that implements a proof-of-concept compiler from a C-like language with components to two target machines: a generic RISC processor and a micro-policy machine [2]. The generated executable runs on the bare hardware with the back-end compiler phase targeting the generic RISC processor. While promising, the micro-policy machine [2] does not exist yet. Here we target a generic load-store machine with no specialized hardware for protection. The novelty of this new software fault isolation implementation is that instead of protecting an application from one or more potentially malicious libraries, all components are potentially malicious and, thus, mutually distrustful.

*Advised by Professor Mary Lou Soffa. University of Virginia, 85 Engineer's Way, Charlottesville, VA 22904. Part of this work was performed as a visiting PhD student at Inria, Paris during the Summer of 2017, under the guidance of Cătălin Hriţcu and Marco Stronati.

In our approach, a source-level program is translated from the C-like language with components to an intermediate level language that uses a similar memory model to CompCert [9] enriched with a notion of component and interfaces between components. The addresses are not resolved and the interface calls between components are abstract. Our work implements a compiler pass in Coq [12]. It takes this intermediate program and generates a RISC assembly program that satisfies the following invariants:

1. a component can write only within its own data memory;
2. a component can only jump within its own code memory, except for predefined exit points allowed by the interface; and
3. if after a call to another component, the execution is transferred back to the callee component, then it will always return to the instruction after the call.

The assumptions in this research are that the basic RISC machine has a minimal load-store instruction set. The register file contains a set of registers dedicated to the software fault isolation instrumentation. The memory is unbounded and it is split into slots. The slots are allocated statically to each component and their type, code or data, is also statically determined. A physical address is an unbounded integer, with the bits starting from the least significant: offset with slot, component identifier, slot identifier. The offset and component are bounded, and the slot identifier is not. Thus, each component has an unbounded memory, but a limit on the contiguous memory it can allocate.

To enforce the first two invariants, this work uses a strategy from Wahbe et al. [13] that has two extra instructions and three dedicated registers. Using binary bitwise operations on an address, the bits corresponding to the component identifier are set to the current one. All the data slots are odd and the instrumentation for the store instruction sets the least significant bit of the slot. All the code slots are even and the instrumentation for jump resets the least significant bit of the slot. Thus, no writes are possible in the code segment.

For the enforcement of the cross-component control flow, we use a dedicated protected control stack and a dedicated register for the stack pointer. The protected control stack is kept in a reserved memory, which can be accessed only from special instrumentation sequences. To ensure continuous execution of a certain number of instructions needed for managing the protected control stack, we align the instructions [10].

The first two sandboxing invariants do not protect the current executing component, but rather protect all other components from it. Special care must be taken to protect the control stack. First, the procedures called externally are placed at an unaligned address and are preceded by a *Halt* instruction. Thus spurious pushes onto the protected control stack are avoided. Second, to avoid the error of popping from an empty stack the execution starts with pushing the address of a *Halt* instruction on the protected control stack and, then the execution is transferred to the main function.

4 Results and Contributions

The project is implemented in Coq [12] and uses the QuickChick [11] framework to test the three invariants. A test consists of the following steps: randomly generate intermediate program using QuickChick's primitives [8], compilate with our proof-of-concept compiler, execute in simulator with recording of a log specific to each invariant using a state monad, and verify the log by a checker [8]. The intermediate programs were syntactically correct and no tests were discarded. Currently, we are working on simulating an attack by randomly injecting a change to the data memory of a component.

The robust compilation property definition cannot be directly applied at the the target level, where the addresses are resolved and a certain layout in memory and instrumentation are expected. Here, the adversarial context is linked and compiled together with the program and the robust compilation property is defined as:

$$\forall P\ C_a\ t.\ ((C_a \bowtie P)\downarrow) \Downarrow t \Rightarrow \exists S\ t'.\ S \bowtie P \Downarrow t' \wedge t' \preccurlyeq_P t \quad (1)$$

In figure 1 the program P has three components, and it's linked with the adversarial component C_a. Together, they are compiled and executed in the target machine semantic and produce the trace t. By robust compilation, there exists a component S, with no undefined behavior, such that: S together with P can be executed in the intermediate semantic, producing a trace t'. The trace t' is a prefix of trace t until S induces and undefined behavior in P.

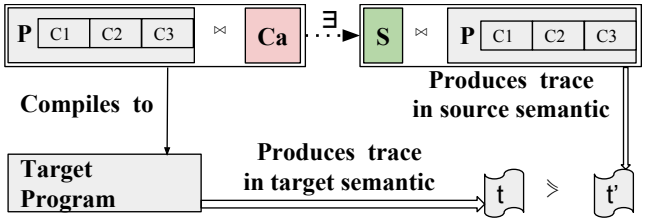

Figure 1. Robust Compilation Intermediate to Target

In conclusion, we designed and implemented a compiler transformation from a RISC-like intermediate language to a basic RISC assembly language that uses software fault isolation mechanisms to provide the memory and control flow separation required by the robust compilation property. We tested the implementation using property based testing [5] and the QuickChick framework [11].

The robust compilation property does not require specialized hardware. More work is needed to support system calls and dynamic loading, but this is an encouraging first step.

References

[1] Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. 19–34. https://doi.org/10.1007/3-540-48749-2_2

[2] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Cătălin Hriţcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE Computer Society, 813–830. https://doi.org/10.1109/SP.2015.55

[3] Guglielmo Fachini, Cătălin Hriţcu, Marco Stronati, Ana Nora Evans, Théo Laurent, Arthur Azevedo de Amorim, Benjamin C. Pierce, and Andrew Tolmach. 2017. Formally Secure Compilation of Unsafe Low-Level Components (Extended Abstract). arXiv:1710.07308. (2017). https://arxiv.org/abs/1710.07308

[4] Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Marco Stronati, and David Swasey. 2017. Robust Hyperproperty Preservation for Secure Compilation (Extended Abstract). arXiv:1710.07309. (2017). https://arxiv.org/abs/1710.07309

[5] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages (PADL'07)*. Springer-Verlag, Berlin, Heidelberg, 1–32. https://doi.org/10.1007/978-3-540-69611-7_1

[6] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 45–60. https://doi.org/10.1109/CSF.2016.11

[7] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components. *CoRR* abs/1510.00697 (2015). http://arxiv.org/abs/1510.00697

[8] Benjamin C. Pierce Leonidas Lampropoulos, Zoe Paraskevopoulou. 2018. Generating Good Generators for Inductive Relations. In *Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2018, Los Angeles, California, USA, January 10-12, 2018*.

[9] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 42–54. https://doi.org/10.1145/1111037.1111042

[10] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 395–404. https://doi.org/10.1145/2254064.2254111

[11] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin Pierce. 2015. Foundational Property-Based Testing. In *International Conference on Interactive Theorem Proving (ITP 2015)*.

[12] The Coq Development Team. [n. d.]. The Coq Proof Assistant Reference Manual. ([n. d.]). https://coq.inria.fr/distrib/current/refman/

[13] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[14] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Commun. ACM* 53, 1 (Jan. 2010), 91–99. https://doi.org/10.1145/1629175.1629203

[15] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: Fully Verified Software Fault Isolation. In *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT '11)*. ACM, New York, NY, USA, 289–298. https://doi.org/10.1145/2038642.2038687