A Trace-Based Proof Technique for Secure Compilation

Extended Abstract

Jérémy Thibault
Inria Paris
jeremy.thibault@inria.fr

Introduction

Good programming languages provide many useful abstractions, such as type or memory safety, or modules and interfaces, that can be used to achieve security. However, the security properties of a source program are not necessarily preserved when compiling it to a low-level language and linking it to adversarial code (a library, for instance). Indeed, this low-level code do not benefit from the source abstractions, and can behave in an unexpected manner. For instance, it can violate memory safety by accessing arbitrary memory locations, or pass ill-typed values to the program.

One of the goal of secure compilation is protecting compiled code from arbitrary low-level context. Traditionally, a compilation chain is proven secure by proving that it is *fully abstract* [1]: it preserves (and reflects) observational equivalence.

Recent work [3] suggested using different criteria, based on the preservation of the satisfaction of properties against adversarial contexts.

We describe one of these criteria, the *robust preservation of finite-relational safety properties*, and show that a simple compiler, between a statically typed, while language with first-order functions and its dynamically typed counterpart, satisfies this criterion. Our proof, centered around the back-translation of a finite set of finite prefixes of traces into a source context, is an adaptation of existing proof techniques for proving full abstraction.

Fully Abstract Compilation

Secure compilation is often stated as *fully abstract compilation* [1]: a fully abstract compiler is a compiler that preserves and reflects observational equivalence, where observational equivalence is instantiated with contextual equivalence. A compiler $\downarrow$ is fully abstract if:

$$\forall P_1 P_2,\ (\forall C_S, C_S\,[P_1] \approx C_S\,[P_2]) \iff (\forall C_T, C_T\,[P_1\downarrow] \approx C_T\,[P_2\downarrow]) \qquad \text{(FA)}$$

where source and target programs or contexts are indexed by S or T, and $C\,[P]$ denotes the linking of program P with context C.

However, achieving and proving a compiler fully abstract is often a very difficult task [5], and the security properties preserved by such a compiler are often hard to describe [2, 3, 6, 7].

To prove full abstraction, and in particular the security-relevant direction, i.e., the preservation of contextual equivalence (left-to-right), several proof techniques have been proposed and used, such as universal embedding [4]. Most of these proofs have in common the fact that they rely on *backtranslating* a distinguishing target context into a distinguishing source context, using the contrapositive of (FA). The source context produced this way aims at emulating the behavior of the target one.

Robust Property Preservation

The various criteria described in [3] are criteria based on the idea of preserving the satisfaction of properties or class of properties, defined over traces, against any adversarial context.

Indeed, instead of focusing on preserving observational equivalence, one might be interested in only preserving classes of properties that are security-relevant, such as safety, liveness, or hyperproperties. These classes are particularly relevant to security: indeed, they include number of well-studied properties, such as noninterference, which is a safety hyperproperty.

Full abstraction do not imply the preservation of these classes. Moreover, the kind of protections needed for full abstraction might be too powerful if one is only interested in preserving particular classes of properties. For instance, a fully abstract compilation chain has to preserve the equivalence of programs; meanwhile, a compilation chain that preserves safety properties might only have to protect some internal invariants.

We say that a (partial) program robustly satisfies a property if it satisfies this property when linked against any context. Then, the role of a compiler that robustly preserves a class of property is to ensure that any of these properties that is robustly satisfied by a program at the source level is also robustly satisfied by the same program, after compilation.

An alternative, "property-free" criterion, more suited for proofs, can be given for each of these criteria, in the same way as the contrapositive of full abstraction. Take, for instance,

the following criterion called Robust Safety Preservation:

$$\forall \pi \in \textbf{Safety}, \forall P, \qquad \text{(RSP)}$$
$$(\forall C_S, \forall t, C_S\,[P] \rightsquigarrow t \implies t \in \pi) \implies$$
$$(\forall C_T, \forall t, C_T\,[P\downarrow] \rightsquigarrow t \implies t \in \pi)$$

where π is a safety property. This criterion states that the compiler preserves the robust satisfaction of the safety properties. An equivalent way of stating it is:

$$\forall P, \forall m, \forall C_T, C_T\,[P\downarrow] \rightsquigarrow m \implies \qquad \text{(RSC)}$$
$$\exists C_S, C_S\,[P] \rightsquigarrow m$$

where m are finite prefixes. This characterization captures the fact that if a bad prefix invalidates a safety property at the target level, then the same prefix can be found at the source level.

Contribution

Compiler and Languages

We consider a source language that is a simple statically typed while language, with first-order functions. We compile it to the same language, except it is dynamically typed instead, with the help of two commands that check the type of a given expression. We restrict the languages, so that the context is initially in control, and performs a sequence of calls to functions from the program (possibly branching on returns). Finally, the traces are made of inputs and outputs to an external environment, and potentially a termination event.

The compiler's role is to introduce dynamic type checks in every function, to ensure that a function can only be called with an argument of the correct type. Otherwise, the compiler makes the function fail.

Robust Finite-Relational Safety Preservation

We define the criterion we call Robust Finite-Relational Safety Preservation. In short, this criterion states that the safety properties (that can be refuted by finite bad prefixes) that relates several partial programs (and traces) together are preserved by compilation:

$$\forall k, \forall \mathcal{R}, \forall P_1 \dots P_k, \qquad \text{(RFin-rSP)}$$
$$(\forall C_S, \forall t_1 \dots t_k, C_S\,[P_1] \rightsquigarrow t_1 \wedge \dots \wedge C_S\,[P_k] \rightsquigarrow t_k$$
$$\implies (t_1, \dots, t_k) \in \mathcal{R}) \implies$$
$$(\forall C_T, \forall t_1 \dots t_k, C_T\,[P_1\downarrow] \rightsquigarrow t_1 \wedge \dots \wedge C_T\,[P_k\downarrow] \rightsquigarrow t_k$$
$$\implies (t_1, \dots, t_k) \in \mathcal{R})$$

where $\mathcal{R}$ is a safety relation of arity k.

The criterion is equivalent to the following contrapositive form, where the m are finite prefixes of traces.

$$\forall k, \forall P_1 \dots P_k, \forall C_T, \forall m_1 \dots m_k, \qquad \text{(RFin-rSC)}$$
$$(C_T\,[P_1\downarrow] \rightsquigarrow m_1 \wedge \dots \wedge C_T\,[P_k\downarrow] \rightsquigarrow m_k) \implies$$
$$\exists C_S, (C_S\,[P_1] \rightsquigarrow m_1 \wedge \dots \wedge C_S\,[P_k] \rightsquigarrow m_k)$$

We can now use a proof technique inspired by proofs techniques used for full abstraction, to prove this result.

We consider k programs, and k finite prefixes produced by the compiled programs linked with the same context. From this set of prefixes, we construct a source context. When linked with this context, the source programs can produce the same set of finite prefixes, hence achieving the criterion.

The proof proceeds as follow:

- First, we instrument the semantics to allow the traces to keep track of more information. Indeed, the simple traces considered that only contain input and outputs do not carry enough information to backtranslate. In particular, they carry no information about the control-flow of the whole program. The informative traces we consider then also track the calls and the returns
- Then, we only consider the parts of the prefixes that can be blamed on the context, as they are the only parts relevant to the backtranslation
- We exploit the fact that the single context handles the control-flow of the whole program by calling functions from the partial programs. Hence, we identify a branching structure in the informative traces, that we reproduce in a single source context.

Our proof technique also highlights the fact that it is important that we can not blame failure on either the program or the context. Indeed, the backtranslated context needs to preemptively cause a failure when a target program would fail due to a type error; that is, a failure might be shifted from the program to the context, when going from target to source.

Future Work

This work is still in progress. We plan on improving the proof technique, to be able to solve several limitations the current compilation chain has. First and foremost, it is necessary to allow callbacks from the program to the context, in order to model a more faithful interaction between the two. Then, an extension to higher-order function seems natural.

On another direction, we will continue to study these criteria, and their relation with full abstraction in particular. We know that the criterion described here implies full abstraction in certain conditions, but we might wonder under what conditions full abstraction imply our criterion, or at least weaker ones.

Finally, we plan on studying other proof techniques, and compare them in terms of complexity, extensibility, and power.

References

[1] M. Abadi. Protection in programming-language translations. In *Security Issues for Mobile and Distributed Objects*. 1999.

[2] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic

compromise. In *25th ACM Conference on Computer and Communications Security (CCS)*. 2018.

[3] C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. arXiv:1807.04603, 2018.

[4] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *21st ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2016.

[5] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.

[6] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 2017.

[7] M. Patrignani and D. Garg. Robustly safe compilation. *CoRR*, abs/1804.00489, 2018.

Submission information

Advisor: Cătălin Hriţcu, Inria Paris
ACM student member number: 0021536
Category: Graduate