Nanopass Back-Translation of Multiple Traces for Secure Compilation Proofs

Jérémy Thibault
MPI-SP

Cătălin Hriţcu
MPI-SP

Abstract. Many secure compilation chains aim at ensuring that if there is no attack a source context can mount against a source program then there is no attack an adversarial target context can mount against the compiled program. One may, for instance, try to prevent attacks against trace properties (e.g., safety), hyperproperties (e.g., noninterference), or relational hyperproperties (e.g., observational equivalence).

Proving that these compilation chains are secure is, however, challenging. For any target attack one has to exhibit a source context mounting the same attack against the source program. Such back-translations can be highly complex, especially for hyperproperties and relational hyperproperties.

We describe a novel back-translation technique that results in simpler proofs, which can be more easily mechanized in a proof assistant like Coq. Given any number of finite trace prefixes, our back-translation builds a single source context producing these prefixes. We see the finite set of trace prefixes as a tree of events, and use state in the back-translated context to record the current position in this tree. The back-translation is done in many small steps, each adding to the tree new information describing how the location should change depending on how the context regains control. To prove such a back-translation correct we give semantics to every intermediate tree language and prove many small forward simulations, basically seeing the back-translation as a verified nanopass compiler.

This technique allows us to prove in Coq a strong secure compilation criterion for an existing simple compilation chain that was previously only proved to satisfy a weaker criterion.

Introduction. Modern languages provide useful abstractions for programming securely, such as modules and interfaces. However, compiling and linking with untrusted low-level code (for instance, a library) doesn't necessarily preserve the security properties of the high-level program. Indeed, such low-level code doesn't have to respect the same abstractions, and could actively attack the compiled program. The main goal of a secure compilation chain is to protect compiled code from such adversarial low-level code.

Abate et al. [1] recently introduced a range of security criteria based on the preservation of classes of trace hyperproperties against adversarial contexts. Achieving security for a realistic compiler is difficult though; and yet more difficult is proving that such a compiler is indeed secure. Yet, Abate et al. [2] showed that (a variant of) one of the weakest criteria from [1], called RSC, is amenable to formal verification in Coq, by proving the security of a compartmentalizing compiler for a small C-like language.

This and most other secure compilation proofs rely on back-translation [3, 6]: given a target context, and possibly more information such as execution traces of this target context, one must build a source context that "behaves like" the target context. We propose a technique for back-translating a finite set of finite trace prefixes into a single source context, and a convenient way to prove its correctness. It can be used to prove one of the stronger criteria of Abate et al. [1]:

$$\begin{aligned}\mathrm{RFrXC} : \forall K\ \forall P_1 \ldots P_K\ C_T\ m_1 \ldots m_K.\\ (C_T[P_1{\downarrow}] \rightsquigarrow m_1 \wedge \ldots \wedge C_T[P_K{\downarrow}] \rightsquigarrow m_K) \Rightarrow\\ \exists C_S.\, (C_S[P_1] \rightsquigarrow m_1 \wedge \ldots \wedge C_S[P_K] \rightsquigarrow m_K).\end{aligned}$$

Given K programs $(P_i)_{1\leq i\leq K}$ that, when compiled and linked with the same target context C_T, produce K finite trace prefixes $(m_i)_{1\leq i\leq K}$, we need to build a *single* source context C_S that, when linked with the K source programs, produces the same prefixes. Each prefix captures the interaction between the context and one program (calls and returns), as well as external I/O or termination events. The RFrXC criterion is a generalization of RSC to multiple programs and prefixes and also implies preservation of noninterference and in our setting also of contextual equivalence (so it implies the interesting preservation direction of full abstraction) [1].

We build upon the secure compilation proof technique from [2], as it cleanly separates the proof into several steps, the most important ones being: a back-translation step producing a source *whole* program from a single trace, a "recomposition" step allowing to split apart and recombine pieces of low-level programs, and standard compiler correctness to navigate between languages. To adapt this proof technique to our stronger RFrXC criterion, we only need to change the back-translation step. Indeed, this is the only step that needs to consider all the programs P_i at once, since we must construct *a single* context that will produce the appropriate prefixes when linked with each of the programs. The other steps, recomposition and compiler correctness, can simply be applied pointwise K times to obtain the stronger result.

The new back-translation is divided in several small simple steps, just as a nanopass compiler, and its correctness is obtained by proving simulations between the intermediate representations. As a result, we are able to mechanize this proof in the Coq proof assistant, resulting in several simple simulation proofs.

Back-translation of multiple traces. Given trace prefixes $(m_i)_{1\leq i\leq K}$ our new back-translation generates not only one source context C_S, but also several source partial programs $(P'_i)_{1\leq i\leq K}$ which together with the context produce the given prefixes. We take inspiration in the back-translation from [2], which keeps track of one private counter per component storing the current location in the trace and dictating what event the back-translated context should emit next. We also take inspiration in an RFrXC proof from [1], which devised a multiple-trace back-translation for *purely functional* languages. This time, and as in [2], we allow and rely on state to store metadata allowing the program to know what part of the trace it already executed. We describe next, on an example, a simplified version of the back-translation generating a single context that produces the given trace prefixes.

Generating the context. we first observe that the set of prefixes that we have to back-translate can be represented as a tree of events. For instance, consider the three prefixes:

$$m_1 = \text{Call } C\ P\ 40; \text{Ret } P\ C\ 41; \Downarrow$$
$$m_2 = \text{Call } C\ P\ 40; \text{Ret } P\ C\ 42; \Uparrow$$
$$m_3 = \text{Call } C\ P\ 40; \text{Call } P\ C\ 43; \text{Ret } C\ P\ 44; \text{Ret } P\ C\ 41; \Uparrow$$

where $\Downarrow$ represents termination, $\Uparrow$ represents divergence, Call $C\ P\ n$ represents a call from context to program with argument n, and similarly for the other events. We use the following tree to represent them:

$$\begin{tikzcd}
 & \text{Call } C\ P\ 40 : \mathbf{1} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\text{Ret } P\ C\ 41 : \mathbf{2} \arrow[d, dash] & \text{Ret } P\ C\ 42 : \mathbf{4} \arrow[d, dash] & \text{Call } P\ C\ 43 : \mathbf{6} \arrow[d, dash] \\
\Downarrow : \mathbf{3} & \Uparrow : \mathbf{5} & \text{Ret } C\ P\ 44 : \mathbf{7} \arrow[d, dash] \\
 & & \text{Ret } P\ C\ 41 : \mathbf{8} \arrow[d, dash] \\
 & & \Uparrow : \mathbf{9}
\end{tikzcd}$$

We give a unique identifier to each node, here in red and bold, which we call the "location" of the node in the tree. In the back-translated context, we will allocate a designated memory cell `loc` that stores this number, called the location of the context. During the execution, we maintain the invariant that `loc` always stores the location of the last event executed. The context uses this information to know which event to perform next and how to update `loc`. For instance, if the current `loc` is **0** (corresponding to the initial state), then the tree tells us the context must call the program with argument 40, and update its own location to **1**.

However, the next event is controlled by the program, so the context does not know yet what location is next: **2**, **4**, or **6**. Hence, when it recovers control, it must determine what its new location is, based on both the previous location it still has stored in `loc`, and whether it gains control by being called or being returned to.

Suppose for instance that when the context recovers control, its location is still **1**. If it just got returned to, then it must be either at location **2** or **4**, depending on the value returned, and it can update its location accordingly. Note that the new location cannot be **8**: while this return corresponds to the same call as the one of location **1**, the context must have gotten the control back at least once before this point (**7**), and hence it has already updated its location to not be **1** anymore. This also explains why the context behaves differently despite being returned the same value (41) both when going from **2** to **3** and from **8** to **9**; in the latter case its state would not be the same though, having been modified in **7**. If instead of a return it receives a call at location **1**, then it has to check the argument of the call and update its location accordingly: here, the next location will be **6**.

To obtain a context with this behavior, the back-translation proceeds in many small steps, like a nanopass compiler [4]:

- It first stores the parent node's location inside every node.
- It then gathers answers to questions such as "if I'm called with argument z in location n, what is my new location?" and "if I'm returned value z in location n, what is my new location?" by performing tree traversals and storing this information inside the corresponding nodes.
- It then uses this gathered information to generate for each tree node several expressions that perform the updates to the location and that emit the events.
- Finally, the tree is flattened, and all expressions are joined into a single expression, using nested if-then-else constructs to branch on the location of the context.

Generating the programs. We also perform the same back-translation in order to obtain K source programs P_i', that will be used in the next steps of the RFrXC proof. The only difference this time is that we obtain P_i' not by back-translating the whole set of prefixes anymore, but instead by applying the previous algorithm on the singleton $\{m_i\}$.

Correctness of the back-translation. We prove this back-translation correct by giving a formal semantics to each intermediate representation of the prefixes, and using standard CompCert-style simulations [5] to relate them. In other words, we see proving that the back-translated context indeed produces the prefixes as a compiler correctness problem, where the compiler is the back-translation, the source language is the set of prefixes, and the target language is the high-level language. We believe this structure simplifies reasoning about the back-translation, as it allows us to reason in small incremental steps instead of in a single, large transformation relying on complex invariants.

The first language of trees has a very simple small-step semantics given by the following rule:

$$\frac{\text{Node } e\ T' \in T}{(T, e :: t) \to^{e} (T', t)}$$

A state consists of a remaining trace t and a list of remaining subtrees T, where Node $e\ T'$ is the tree whose root is the event e and whose children are the trees in the list T'.

At each step of the back-translation, we add more information to the trees, and at the same time, we use ghost state in the semantics to enforce explicitly the invariants of the execution. For instance, the next step of the back-translation is to add the location to each node, and the state becomes a triple that also contains this location that is updated as the execution progress through the tree. This location is updated so that it always contains the location that will be stored in the concrete state when executing the back-translated program in the source semantics. Eventually, in the last simulation proofs, we prove that this ghost counter is always equal to the concrete value stored in `loc` cell when the back-translated context is executed.

This allows us to progress step-by-step in the proof, only considering one invariant at a time, and as such considerably simplifies the mechanization of the proof.

Conclusion, future work. With this work, we aim at providing a back-translation technique that is general and modular enough to be reused in different proofs of secure compilation, in a proof assistant like Coq. We are also working on applying this back-translation technique to the compiler from [2], and hope it will soon replace the previous proof.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 256–271. https://doi.org/10.1109/CSF.2019.00025

[2] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1351–1368. https://doi.org/10.1145/3243734.3243745

[3] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-abstract Compilation by Approximate Back-translation. *Log. Methods Comput. Sci.* 13, 4 (2017). https://doi.org/10.23638/LMCS-13(4:2)2017

[4] Andy Keep. 2020. The Nanopass Framework as a Nanopass Compiler (ELS keynote). In *Proceedings of the 13th European Lisp Symposium (ELS 2020), Zurich, Switzerland, April 27-28, 2020*, Ioanna Matilde Dimitriou Henríquez (Ed.). ELSAA.

[5] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[6] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 103–116. https://doi.org/10.1145/2951913.2951941