Secure Composition of SPECTRE Mitigations

Matthis Kruse
CISPA Helmholtz Center for Information Security
Germany
matthis.kruse@cispa.de

Michael Backes
CISPA Helmholtz Center for Information Security
Germany
director@cispa.de

1 Introduction

Some modern programming languages enjoy strong security guarantees, for example the Rust programming languages has memory safety guarantees given by its compiler performing a semantic analysis pass. While programmers may expect that these guarantees hold even after translating their program to the target programming language, it has been shown that this expectation is false in the general case [4].

Correct compilers do not necessarily provide satisfactory guarantees [4] and thus one has to resort to secure compilers [2, 3, 8] which preserve property satisfaction even when the compiled program is linked with arbitrary target-level code, i.e., the compiled program is robust. A recent framework [7] describes how to compose secure compilers, thus allowing a divide-and-conquer approach to the engineering of secure compilers. This framework primarily focuses on compilers that do not change the traces of the original program. However, real-world compilers perform source-code transformations that may change the trace, such as applying source-code instrumentations that enhance security by, e.g., inserting bounds-checks. Other work [1] showed that there are essentially two approaches to this, where the robust preservation [2, 3, 8] statement is changed to lift the restriction of a unified trace-model as follows:

Top-down Approach	**Bottom-up Approach**
if p robustly satisfies π, then $\gamma(\text{p})$ robustly satisfies $\tau_{\sim}(\pi)$	if p robustly satisfies $\sigma_{\sim}(\pi)$, then $\gamma(\text{p})$ robustly satisfies π

Hereby, γ is a compiler, $\sim$ a cross-language trace relation from S-level traces to T-level traces describing the semantic effect of γ, and $\tau_{\sim}/\sigma_{\sim}$ project a source/target property to a target/source property. While the compositionality framework [7] does consider the top-down approach, it does not entail composition of the bottom-up one. However, the bottom-up approach is crucial for security properties that can only be described in the target-level, such as absence of SPECTRE vulnerabilities [6], since higher-level languages do not model speculation in their semantics.

It is worth noting that some compiler compositions may not give the wanted security properties, such as when composing with Index-Masking Defense (IMD) [10] that prevents SPECTREv1 attacks that exploit speculative execution of loads happening after a branch. While IMD prevents SPECTREv1 attacks, it can introduce SPECTREv4 vulnerabilities and, thus, can render a SPECTREv4 mitigation run prior to IMD useless.

Compilation passes that do not violate the security properties of earlier ones fulfill a notion of compatibility of a cross-language trace relation that describes the effect of the compiler. Because compatiblity is defined on cross-language trace relations, there is no need to reason about the concrete, syntactic changes a compilation pass does.

Definition 1.1 (Compatibility)**.** Given languages S and T, a cross-language trace relation $\sim$ between traces of S and T, and a T–level collection of properties $\mathbb{C}$, then $\sim$ is compatible with $\mathbb{C}$ iff for any $\pi \in \mathbb{C}$ it holds that $\sigma_{\sim}(\pi) \in \sigma_{\sim}(\mathbb{C})$.

This extended abstract extends prior work [7] to consider bottom-up secure compiler composition and aims to apply that theory to a selection of mitigations for SPECTRE variants. It is demonstrated that it suffices to setup a cross-language trace relation that describes the semantic effect of a secure compiler and prove compatibility with properties of interest in order to compose the secure compiler without giving up on security guarantees.

2 Composing Secure Compilers

The composition of secure compilers requires two theorems to be proven: (1) robust preservation, either with a unified trace-model [3], top-down, or bottom-up, and (2) Definition 1.1 (Compatibility). The rest of the paper assumes that the presented SPECTRE mitigations have been proven secure as in (1). The property that all mitigations aim to robustly preserve is a variant of Speculative Safety (SS) [9], which relies on a taint-tracking mechanism and taints (σ) that tag events as safe (S) or unsafe. Contrary to the original definition of SS, this paper states SS such that tags should be unequal to the tag of the kind of variant (vX) that one is interested in:

Definition 2.1 (SS for variant vX)**.** $\pi_{\mathrm{vX}} = \{\bar{\mathrm{a}} \mid \forall \mathrm{a}^{\sigma} \in \bar{\mathrm{a}}, \sigma \neq \mathrm{vX}\}$

The original SS [9], hereby named π_{ss}, can be recovered:

$$\pi_{ss} = \bigcap_{\mathrm{vX}} \pi_{\mathrm{vX}}$$

Robust preservation (only for top-down or bottom-up) and compatiblity (Definition 1.1) require a cross-language trace relation that describes the effect of a corresponding compiler semantically. Therefore, for the rest of the extended abstract, it is assumed that there are source (S) and target (T) languages, which share a large portion of their trace model. The trace models must have some kind of allocation ($\mathrm{Alloc}(\ell; n)$), memory load/store ($\mathrm{Get}(\ell; idx; n)$ and $\mathrm{Set}(\ell; idx; n; v)$), branch ($\mathrm{If}(b)$), and indirect branch events ($\mathrm{Ibranch}(v)$), jumps ($\mathrm{Jmp}(v)$), as well as a marker event for a barrier ($\times$), and a rollback event (Rlb) [9]. Trace events are annotated with taint tags and for sake of readability, trace events tagged with the secure tag (S) are written without the tag. This is enough setup to sketch the cross-language trace relations describing the semantic changes each considered mitigation does:

- Index-Masking Defense (IMD) [10] (v1)

$$\begin{aligned}\bar{\mathsf{a}} \sim^{v1}_{\texttt{IMD}} \bar{\mathsf{a}} \quad \equiv \quad & \text{if } \mathsf{Alloc}(\ell; \mathsf{n}), \mathsf{Get}(\ell; \mathsf{idx}; \mathsf{v}) \in \bar{\mathsf{a}} \\ & \text{then } \exists \ell\ \mathbf{n}\ \mathbf{idx}\ \mathbf{v}, \mathbf{Alloc}(\ell; \mathbf{n})^{\sigma} \in \bar{\mathbf{a}} \\ & \text{s.t. } \mathbf{Get}(\ell; \mathbf{idx}; \mathbf{v})^{\sigma'} \in \bar{\mathbf{a}} \\ & \text{and } \exists m, \mathbf{n} = 2^m \text{ and } \mathbf{idx} \leq 2^m \\ & \text{and } \ell \approx \ell \text{ and } \mathsf{v} \approx \mathbf{v}\end{aligned}$$

  IMD changes memory allocation to be powers-of-2 and masks all indices with the bounds of the array.

- Insertion of `lfences` (LFENCE) [11] (v1, v4)

$$\begin{aligned}\bar{\mathsf{a}} \sim^{v1,v4}_{\texttt{LFENCE}} \bar{\mathbf{a}} \quad \equiv \quad & (\text{if } \forall i, \bar{\mathbf{a}}[i] = \mathbf{If}(\_)^{\sigma} \text{ then } \bar{\mathbf{a}}[i+1] = \times^{\sigma'}) \\ \text{and} \quad & (\text{if } \forall i > 0, \bar{\mathbf{a}}[i] = \mathbf{Get}(\_)^{\sigma''} \\ & \text{then } \bar{\mathbf{a}}[i-1] = \times^{\sigma'''})\end{aligned}$$

  LFENCE simply puts a barrier after any branch or before any load instruction. While the literature provides (partial) solutions that do not insert the barrier *everywhere*, due to the significant performance penalty, the considered pass is simple and puts the barrier „everywhere", i.e., in front of loads and after branches. Since the S-level trace is completely irrelevant, this is an example for an enforcement.

- Return Trampoline (Retpoline) [12] (v2)

$$\frac{\mathsf{v} \approx \mathbf{v} \qquad \bar{\mathbf{a}} = \mathbf{Set}(\ell; \mathbf{sp}; \mathbf{v})^{\sigma} \cdot \mathbf{Ret}^{\sigma'} \cdot \overline{\mathbf{Jmp}}^{\overline{\sigma''}} \cdot \mathbf{Rlb} \cdot \bar{\mathbf{a}}'}{\mathsf{Ibranch}(\mathsf{v}) \cdot \bar{\mathsf{a}} \sim^{v2}_{\texttt{Retpoline}} \bar{\mathbf{a}}}\ \text{(retpol-ibranch)}$$

  The Retpoline applies for every indirect branch on the source-level trace. Each indirect branch at source-level must have an associated retpoline at target-level, as sketched with the rule retpol-ibranch. That is, the address of the indirect call must be pushed onto the stack to be used in the return instruction and speculation busy waits until the rollback happens.

- Set Model Specific Register Flags (MSR) [5] (`v2, v4, v5`)

$$\bar{\mathsf{a}} \sim^{v2,v4,v5}_{\texttt{MSR}} \bar{\mathbf{a}} \equiv \forall \mathbf{a}^{\sigma}, \sigma \notin \{\mathbf{v2}, \mathbf{v4}, \mathbf{v5}\}$$

  Modern processors have flags to turn off speculation features, resulting in complete absence of speculation (for these variants). This is another example of an enforcement.

It remains to show Definition 1.1 (Compatibility). Without a proof of Definition 1.1 (Compatibility), the composition of mitigations may not provide the security guarantees of interest, since one could intuitively „undo" what another one did. This extended abstract does not provide formal proof for all possible compositions of above mitigations, but sketches the anticipated proofs of compatibility theorems in Figure 1.

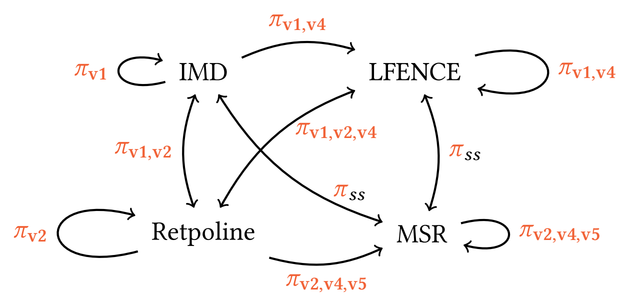

Figure 1. Compatibility of source-code instrumentations to prevent attacks of individual or multiple SPECTRE-variants. Nodes are mitigations that perform the respective source-code instrumentation. Edges are directed and represent compatibility of the composition. The origin of an edge is the compiler that should be run first, the target of an edge is the compiler that should be run afterwards. Edge labels indicate the SS variants.

References

[1] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An extended account of trace-relating compiler correctness and secure compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4 (2021), 1–48.

[2] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 1–28.

[3] 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 *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 256–25615. https://doi.org/10.1109/CSF.2019.00025

[4] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”. In *CSF 2018 - 31st IEEE Computer Security Foundations Symposium*. Oxford, United Kingdom. https://hal.archives-ouvertes.fr/hal-01959560

[5] Red Hat. [n. d.]. Controlling the Performance Impact of Microcode and Security Patches for CVE-2017-5754 CVE-2017-5715 and CVE-2017-5753 using Red Hat Enterprise Linux Tunables. https://access.redhat.com/articles/3311301.

[6] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[7] Matthis Kruse, Michael Backes, and Marco Patrignani. 2023. Secure Composition of Robust and Optimising Compilers. arXiv:2307.08681 [cs.CR]

[8] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 1:1–1:41. https://doi.org/10.1145/3436809

[9] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with secure compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 445–461.

[10] Filip Pizlo. 2018. What Spectre and Meltdown Mean For WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[11] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2022. Mitigating speculative execution attacks via context-sensitive fencing. *IEEE Design & Test* (2022).

[12] Paul Turner. [n. d.]. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.