Exorcising Spectres with Secure Compilers

Marco Guarnieri
IMDEA Software Institute

Marco Patrignani
Stanford University
CISPA Helmholz Center for Information Security

1 Introduction & Outline

Speculative execution is an optimization technique that speeds up computation by predicting the outcome of branching (and other instructions) and continuing the execution based on such predictions. Whenever the processor realises that the prediction is incorrect, it rolls back the effects of the speculatively executed instructions on the CPU architectural state. Side-effects on the so-called microarchitectural state, e.g., cache content, are, however, not rolled back.

As demonstrated by the Spectre attacks [10, 12, 13, 15, 22], an attacker can exploit microarchitectural side-effects of speculatively executed instructions to violate data confidentiality in all modern general-purpose CPUs. Recently, many software-level countermeasures have been proposed to prevent (or at least mitigate) the impact of speculative execution attacks. For instance, the insertion of speculation barriers [8], the use of branchless bound checks [21], the injection of data dependencies [18], and speculative load hardening [2] *could* all be used to mitigate speculative execution attacks on branch instructions (the so-called Spectre V1 attacks). Some of these countermeasures have been deployed in major compilers: the Microsoft Visual C++ and the Intel C++ compilers can automatically insert speculation barriers [9, 19], whereas Clang supports speculative load harderning [2].

These compiler-level countermeasures, however, are not proven to be secure, and some of them provide no precise security guarantees. For instance, the countermeasure implemented in the Microsoft Visual C++ compiler fails in blocking some speculative leaks [7, 11].

In this paper we report on our ongoing effort towards formally reasoning about the effectiveness of these countermeasures. For this, we require: (1) a property characterizing security against speculative execution attacks, and (2) a criterion stating that a compiler can preserve such a security property. Concerning point (1), we rely on *speculative non-interference* [7], which we present in Section 1.1. Regarding point (2), we rely on the *robust compilation* criteria by Abate et al. [1], which we discuss in Section 1.2. We outline our methodology to formally reason about the effectiveness of Spectre countermeasures in Section 1.3. We conclude by presenting a concrete plan for tackling this goal in Section 1.4.

1.1 Speculative Non-Interference

Several characterizations of security against speculative execution attacks have been recently proposed [3, 4, 7]. Here we focus on *speculative non-interference* (SNI) [7], which compares the leakage of a program under two different semantics: a standard, *non-speculative semantics* and a *speculative semantics.* The former is used to capture the intended program behaviour, whereas the latter captures the effects of speculatively executed instructions. Informally, SNI requires that speculatively executed instructions do not leak more information into the microarchitectural state than what is leaked by the standard, non-speculative semantics. To capture this microarchitectural leakage, Guarnieri et al. [7] consider an observer of the program execution that sees *traces* describing the locations of memory accesses and jump targets.

The SNI framework is parametric in two trace semantics, $\mathtt{Sem}\,(P,s)$ and $\mathtt{Spec\text{-}Sem}\,(P,s)$, which yield traces of program counters and memory accesses that result by executing a program P from a state s under the non-speculative and speculative semantics respectively. Formally, a program P satisfies SNI ($P \vdash \mathit{SNI}$) if whenever two states produce the same traces under the non-speculative semantics $\mathtt{Sem}\,(P,\cdot)$, then they also produce the same traces under the speculative semantics $\mathtt{Spec\text{-}Sem}\,(P,\cdot)$, as stated in Definition 1.

Definition 1 (SNI). $P \vdash \mathit{SNI} \overset{\text{def}}{=} \forall\, s_1, s_2.$

$$\begin{aligned}&\mathtt{Sem}\,(P,s_1) = \mathtt{Sem}\,(P,s_2) \Rightarrow\\ &\quad \mathtt{Spec\text{-}Sem}\,(P,s_1) = \mathtt{Spec\text{-}Sem}\,(P,s_2)\end{aligned}$$

Definition 1 is formalized in terms of the semantics $\mathtt{Sem}\,(P,s)$ and $\mathtt{Spec\text{-}Sem}\,(P,s)$. For the semantics given in [7], SNI can be equivalently formulated using only the speculative semantics $\mathtt{Spec\text{-}Sem}\,(P,s)$ and a trace projection $\cdot\!\upharpoonright_{nse}$ that returns the non-speculative projection of a speculative trace [7, Sec. 4.E]. That is, SNI can be equivalently formulated as:

$$\begin{aligned}\forall s_1, s_2.\ &\mathtt{Spec\text{-}Sem}\,(P,s_1)\!\upharpoonright_{nse} = \mathtt{Spec\text{-}Sem}\,(P,s_2)\!\upharpoonright_{nse}\\ &\Rightarrow \mathtt{Spec\text{-}Sem}\,(P,s_1) = \mathtt{Spec\text{-}Sem}\,(P,s_2)\end{aligned}$$

Before illustrating how (a variation of) SNI can be used as a security foundation for compiler-based Spectre countermeasures (in Section 1.3), we discuss robust compilation.

1.2 Robust Hypersafety-Preserving Compilation

Recent work on secure compilation has devised a whole new spectrum of criteria, which we refer to as Robust Compilation (RC) [1], that preserve classes of hyperproperties (i.e., arbitrary program behaviours [5]). These criteria are robust, so they talk about compiled code interacting with arbitrary target-level attackers. RC criteria exist for compilers that wish to preserve safety properties, arbitrary properties, 2-hypersafety properties, arbitrary safety properties, arbitrary hyperproperties and many more. Since SNI can be reduced to a non-interference property, and since non-interference is a 2-hypersafety property, we choose the criterion *Robust 2-Hypersafety Compilation* (R2HSP) for our development (Definition 2 below).

Intuitively, a compiler ($[\![\cdot]\!]$) preserves any 2-hypersafety property robustly ($[\![\cdot]\!] \vdash \mathit{R2HSP}$), if any two *finite traces* (or prefixes) m_1 and m_2 generated by the compiled program ($[\![\mathrm{P}]\!]$) can also be generated by its source counterpart (P). A program generates a prefix if the prefix is in the set of behaviours of the program. Behaviours are calculated only with respect to whole programs, so compiled and source programs are closed respectively with target and source program contexts (respectively C and C). Behaviours are generated according to the semantics of each language (thus their colouring) but they are expressed in a language common to both source and target (thus the black colouring of prefixes). This is a rather common abstraction on program behaviours which does not hamper our reasoning [14].

Definition 2 (R2HSP)**.** $[\![\cdot]\!] \vdash \mathit{R2HSP} : \forall \mathrm{P}, \mathrm{C}, \{m_1, m_2\}.$ $\{m_1, m_2\} \leq \mathrm{Beh}(\mathrm{C}\,[[\![\mathrm{P}]\!]]) \Rightarrow \exists \mathrm{C}.\{m_1, m_2\} \leq \mathrm{Beh}(\mathrm{C}\,[\mathrm{P}])$

1.3 Are Spectre Countermeasures Secure?

To lift SNI to the compilation setting, we will adapt the $\mathit{Beh}(\cdot)$ notions from Definition 2 to their counterparts in Definition 1 in a new criterion (SNIP, Definition 3). Concretely, $\mathrm{Beh}(\cdot)$ will capture speculative execution ($\mathtt{Spec\text{-}Sem}\,(\cdot,\cdot)$) according to the language semantics, so compiled programs will be subject to speculation. On the other side, $\mathrm{Beh}(\cdot)$ will capture the non-speculative execution ($\mathtt{Sem}\,(\cdot,\cdot)$), so any source program trace will not contain speculation. The two traces m_1 and m_2 will then capture the traces starting from the initial states s_1 and s_2 respectively. In the source, they are generated by applying $\cdot\!\upharpoonright_{nse}$ to the generated traces, in the target, they are obtained by the semantics. Thus, SNIP effectively lifts SNI to the compilation setting, evaluating the semantics of the program patched with the countermeasure against the non-speculative semantic.

Definition 3 (SNIP, Informally)**.** $[\![\cdot]\!] \vdash \mathit{SNIP} : \forall \mathrm{P}, \mathrm{C}, \{m_1, m_2\}.$

$$\{m_1, m_2\} \leq \mathrm{Spec\text{-}Sem}\,(\mathrm{C}\,[[\![\mathrm{P}]\!]])$$
$$\Rightarrow \exists \mathrm{C}.\{m_1\!\upharpoonright_{nse}, m_2\!\upharpoonright_{nse}\} \leq \mathrm{Sem}\,(\mathrm{C}\,[\mathrm{P}])$$

Proving a compiler to be SNIP (i.e., our form of R2HSP) means that compiled code behaves as though the speculatively executed instructions do not leak sensitive information *no matter what they interact with* ($\forall \mathrm{C}$). That is, any speculatively executed instruction only reveals information already disclosed under the non-speculative semantics, and as such it does not leak anything via speculative execution attacks.

On the other hand, a compiler *violating* SNIP will generates code whose speculation behaviour *in the target language* is not possible in the source ($\nexists \mathrm{C}$). That behaviour (i.e., that pair of traces) speculatively leaks information that is not leaked under the non-speculative semantics. Thus, a compiler that is not SNIP produces programs that might be vulnerable to Spectre-like attacks.

A final caveat is that we must prevent the actions generated by the context to pop up in the traces. Otherwise, a context could speculate and be subject to Spectre and trivially invalidate our theorem. However, our results only care about compiled code (which contains the Spectre countermeasure), so we are not concerned with violations from the context, thus we will eliminate them.

Proof-wise, to prove a compiler is R2HSP (and also SNIP) we need to create (read, backtranslate, in secure compilation jargon) a source context C starting from either the target context C or the two traces m_1 and m_2. Depending on the structure and the information carried by the traces, they may be enough to create C, which means we can use a simplified form of so-called trace-based backtranslation [1, 20]. Otherwise, we can use a context-based backtranslation, relying solely on C to build C [1, 6, 17].

We believe we can show that the aforementioned Microsoft Visual C++ compiler [19] does not attain SNIP, while compilers that (correctly) implement the speculation barriers [2] and speculative load hardening [9] do attain SNIP. These results will confirm that our approach to reason about compiler-level Spectre countermeasures is correct and it captures the right security intuition.

1.4 Implementing This Strategy

We outline our concrete plan for the work we just presented on reasoning about the security of Spectre countermeasures.

1. As explained in Section 1.3, we will start our effort by analyzing the effectiveness of the countermeasures against Spectre V1 attacks, i.e., insertion of speculation barriers [9, 16] and speculative load hardening [2]. This requires formalizing such countermeasures as compilers and proving their correctness as discussed in Section 1.3. We plan to carry out all our proofs in the context of the speculative and non-speculative semantics of Guarnieri et al. [7]. However, we will lift those results from assembly to while languages for simplicity.
2. Then we plan to focus on compiler-level countermeasures for other Spectre variants (e.g., *retpoline* for Spectre V2). This will require extending the semantics of Guarnieri et al. [7] with additional features, such as speculation over indirect jumps, and showing that SNIP still captures these attacks. We will also have to prove the correctness of the new countermeasures against these variants.
3. Finally, we want to consider more complex speculative semantics, such as the one of Cauligi et al. [3] and see whether our approach scales to those semantics too.

Acknowledgements: This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by the Community of Madrid under the project S2018/TCS-4339 BLOQUES and the Atracción de Talento Investigador grant 2018-T2/TIC-11732A, by the Spanish Ministry of Science, Innovation, and University under the projects RTI2018-102043-B-I00 SCUM, and by a gift from Intel Corporation.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32th Computer Security Foundations Symposium*, CSF 2019, June 2019.

[2] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[3] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards constant-time foundations for the new spectre era. *arXiv preprint arXiv:1910.01755*, 2019.

[4] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, CSF '19, 2019.

[5] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://www.cs.cornell.edu/~clarkson/papers/clarkson_hyperproperties_journal.pdf.

[6] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[7] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, S&P '20. IEEE, 2020.

[8] Intel. Intel Analysis of Speculative Execution Side Channels. https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf, 2018.

[9] Intel. Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues, 2018.

[10] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *CoRR*, abs/1807.03757, 2018.

[11] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[12] 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. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, S&P '19. IEEE, 2019.

[13] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies*, WOOT '18. USENIX Association, 2018.

[14] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. URL http://dx.doi.org/10.1007/s10817-009-9155-4.

[15] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, CCS '18. ACM, 2018.

[16] Matt Miller. Mitigating speculative execution side channel hardware vulnerabilities. https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018.

[17] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *21st ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 103–116, 2016. doi: 10.1145/2951913.2951941. URL https://www.williamjbowman.com/resources/fabcc-paper.pdf.

[18] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *CoRR*, abs/1805.08506, 2018.

[19] Andrew Pardoe. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[20] Marco Patrignani and Deepak Garg. Robustly Safe Compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*, ESOP'19, 2019.

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

[22] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Proceedings of the 24th European Symposium on Research in Computer Security*, ESORICS '19, 2019.