Lifting Compiler Security Properties to Stronger Attackers: the Speculation Case

Xaver Fabian
Cispa Helmholtz Center for Information Security
Saarbrücken, Germany

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain

Michael Backes
Cispa Helmholtz Center for Information Security
Saarbrücken, Germany

1 Introduction

Speculative execution avoids pipeline stalls by predicting intermediate results and by speculatively executing instructions based on such predictions. When a prediction is incorrect, the processor squashes the speculative instructions, thereby rolling back their effect on the architectural state. Speculative instructions, however, leave footprints in microarchitectural components (e.g., caches) that persist even after speculative execution terminates. Modern processors have different speculation mechanisms (branch predictors, memory disambiguators, etc.) that are used to speculate over different kinds of instructions: conditional branching [10], indirect jumps [10, 13], store and load operations [8], and return instructions [11]. As shown by Spectre [10], attackers can exploit the side effect of these instructions to leak information about speculatively accessed data.

Countermeasures against speculative leaks are typically developed as secure compiler passes. These passes have one concrete attacker model in mind, that is, an attacker that observes certain specific events during the execution of the program e.g., memory effects via store and load observations. Thus, a secure compiler is secure w.r.t a concrete attacker. For example, the SLH countermeasure [3] is used against Spectre v1; there, the attacker is able to observe speculative execution of branch instructions.

However, three concerns arise in this setting. First, when the developer of the secure compiler pass does not consider other kinds of speculation (or, different attackers observing different kinds of speculation) new speculative leaks arise, as in the case of the work of Daniel et al. [4]. Second, new sources of speculation are still discovered to this day [2, 13], so we need to make sure that countermeasures have no speculative leaks even with respect to new attacks. Third, some secure compiler passes turn out to be secure even with regards to a stronger attacker, namely one that is observing more speculative leaks.

In this paper, we want to identify when compilers can be secure even for stronger attackers and we devise a formal framework for establishing exactly the kind of well-formedness conditions that lead to these kind of security guarantees.

The rest of the paper is structured as follows. First, we devise a way to combine different attacker models into stronger ones (Section 1.1). Second, we define the necessary conditions to lift our secure compiler pass from being secure against the weaker attacker to being secure against the stronger attacker (Section 1.2) and show that in certain cases (relevant to Spectre countermeasures) we can derive the well-formedness conditions for free using a syntactic argument (Section 1.3). Lastly, we discuss whether existing Spectre countermeasures satisfy these conditions (Section 1.4).

We focus on countermeasures against Spectre attacks, though we think our definitions scale to a more general setting too, but we leave this investigation to future work.

1.1 General Definitions and Combined Attacker

We use μAsm, an assembly-like language [7] for the definition of our semantics and define the attacker using the contract framework of Guarnieri et al. [6]. Contracts are defined as the combination of an observer and an execution mode. The observer governs what information a contract exposes and is defined via labels on the semantics. The execution mode characterizes which paths of the program need to be explored. We fix the observer to be the constant-time attacker [1, 12] and define the execution mode using our semantics, capturing different sources of speculation. We write $\textcircled{?}_x$ to indicate the attacker with execution mode for Spectre variant x and the fixed constant-time observer.

Now, a stronger attacker can observe more speculative leaks than a weaker one. We use the framework proposed by Fabian et al. [5], which combines speculative semantics $\textcircled{?}_x$ and $\textcircled{?}_y$ into a new combined speculative semantics $\textcircled{?}_{xy}$ allowing for speculation of both source semantics at the same time, as new execution modes for our contracts. Thus, the attacker $\textcircled{?}_{xy}$ is stronger than the attacker $\textcircled{?}_x$ and $\textcircled{?}_y$.

Compilers $[\![\cdot]\!]$ compile from μAsm to μAsm. We write $[\![\cdot]\!]_x$ to mean that the compiler inserts a countermeasure against semantics $\textcircled{?}_x$. We call a program speculative safe for variant x, written $\vdash^{\textcircled{?}_x} p : \text{SS}$, iff there are no speculative leaks under the $\textcircled{?}_x$ attacker and a compiler is speculative safe written $\vdash^{\textcircled{?}_x} [\![\cdot]\!]_x : \text{SS}$, iff it does not leak under variant x, i.e., iff $\forall p, \vdash^{\textcircled{?}_x} [\![p]\!] : \text{SS}$.

1.2 Well-Formed Translator

We say that a compiler for variant y is a well-formed translator for variant x if the compiler does not add leakage under either x or y. Intuitively, a compiler attains this property,

when it is already preventing leaks under y (dubbed Security in Source) and when it is independent of leaks introduced by variant x (dubbed Independence).

Security in Source tells us that the countermeasure of the compiler is effective under attacker Attacker_y and it ensures that the compiler countermeasure works.

Independence means that a compiler $[\![\cdot]\!]_y$ against attacker Attacker_y does not introduce new vulnerabilities under the attacker of Attacker_x. Phrased differently, there are no bad interactions happening by the introduction of the compiler countermeasure into program p under the attacker of Attacker_x. For example, the retpoline countermeasure [9] protects against Spectre v2 by inserting **ret** instructions into the program. Spectre v5 abuses speculation on **ret** instructions. Thus, the question naturally arises if the newly introduced **ret** instructions introduced by the countermeasure for Spectre v2 can be abused to create a speculative leakage caused by Spectre v5. If the retpoline countermeasure fulfils independence, then we know that the introduced **ret** instructions do not yield new speculative leaks under the Spectre v5 attacker.

Definition 1 (Well-formed Translator (WFT))**.**
We call a compiler $[\![\cdot]\!]_y$ a well-formed translator for Attacker_x written $\vdash [\![\cdot]\!]_y WFT_{\text{Attacker}_x}$ with respect to two source languages Attacker_x and Attacker_y and their well-formed composition Attacker_{xy} iff:

Security in Source $\vdash [\![\cdot]\!]_y : SS$

Independence $\vdash^{\text{Attacker}_x} p : SS \implies \vdash^{\text{Attacker}_x} [\![p]\!]_y : SS$

Equipped with the definition of a WFT, we can define the main corollary of our work. Namely, we can embed a secure compiler pass for variant Attacker_y into a stronger attacker model Attacker_{xy}, provided that the compiler $[\![\cdot]\!]_y$ is a WFT for Attacker_x and the program being speculative safe under the Attacker_x attacker.

Corollary 1 (Lifted Compiler Preservation)**.**
$\vdash ([\![\cdot]\!]_y : WFT_{\text{Attacker}_x} \wedge \vdash^{\text{Attacker}_x} p : SS \implies \vdash^{\text{Attacker}_{xy}} [\![p]\!]_y : SS$

1.3 Easier Independence Proofs

Using our running example of Spectre v2 and Spectre v5, we can see that the newly introduced **ret** instructions by the retpoline compiler could yield new speculative leaks caused by Spectre v5 in the combination. That is why we have the **Independence** condition in the definition of well-formedness that we need to prove. However, consider a secure compiler pass $[\![\cdot]\!]^f_5$ only inserting fences into the program to protect against Spectre v5. These fences cannot interact badly with other speculation mechanisms but we would still need to prove independence! Thus, we derive another condition, **Syntactic Independence**, that captures these cases when there are trivially no bad interactions and show how we can derive Independence for free.

As the name suggests, Syntactic Independence can be checked by *syntactic* inspection of the compiler $[\![\cdot]\!]$ and the target attacker Attacker_y. For example, for Spectre v5 and $[\![\cdot]\!]^f_5$ the instructions related to speculation are {**ret**} and the instructions added by the compiler are {**fence**}.

We are now ready to define Syntactic Independence:

Definition 2 (Syntactic Independence)**.**
A compiler $[\![\cdot]\!]_x$ is called syntactically independent of a source language Attacker_y with speculation written $\vdash [\![\cdot]\!]_x : SI_{\text{Attacker}_y}$ iff the compiler does not insert any instructions during compilation that are related to speculation in language Attacker_y.

Next, we connect syntactic independence to independence via the following corollary:

Corollary 2.
$\vdash [\![\cdot]\!] : SI_{\text{Attacker}_y} \implies (\vdash^{\text{Attacker}_y} p : SS \implies \vdash^{\text{Attacker}_y} [\![p]\!] : SS\ )$

Since syntactic independence is a syntactic property of the compiler and the attacker, it is simple to check and together with Corollary 2 we are able to derive independence for free.

1.4 Application to Existing Countermeasures

A preliminary investigation seems to suggest that independence or syntactic independence applies to several existing countermeasures against Spectre attacks (Table 1). These results still need to be formalized and we plan to investigate when *SI* is not enough (the *I* entries).

Compiler	Attacker_{1+2}	Attacker_{1+4}	Attacker_{1+5}	Attacker_{2+4}	Attacker_{2+5}	Attacker_{4+5}	Attacker_{1+4+5}	Attacker_{1+2+4}	Attacker_{2+4+5}	Attacker_{1+2+5}	$\text{Attacker}_{1+2+4+5}$
$[\![\cdot]\!]^f_4$		SI		SI		SI	SI	SI	SI		SI
$[\![\cdot]\!]^{\dot{-}}_2$	SI			SI	I			SI	I	I	I
$[\![\cdot]\!]^{SLH}_1$	SI	SI	SI				SI	SI		SI	SI
$[\![\cdot]\!]^f_5$			SI		SI	SI	SI		SI	SI	SI
$[\![\cdot]\!]^{idx}_1$	SI	✗	SI				✗	✗		SI	✗

Table 1. Lists of Secure compilers and how they attain Independence. SI means by Syntactic Independence, I means by proving Independence and ✗ means that Independence is not possible. $[\![\cdot]\!]^{\dot{-}}_2$ is the retpoline compiler [9], $[\![\cdot]\!]^{idx}_1$ is the countermeasure described by [4], $[\![\cdot]\!]^f_x$ inserts fences into the program and $[\![\cdot]\!]^{SLH}_1$ implements the SLH countermeasure [3]. Empty cases are orthogonal attackers, where the definition of well-formed translator is not applicable.

References

[1] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[2] arm. 2020. Straight-line Speculation. https://developer.arm.com/documentation/102825/0100/?lang=en. Accessed: 2022-09-05.

[3] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html

[4] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter — Efficient relational symbolic execution for Spectre with Haunted RelSE. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society.

[5] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 965–978. https://doi.org/10.1145/3548606.3560555

[6] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1868–1883.

[7] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP40000.2020.00011

[8] J. Horn. 2018. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: 2021-04-11.

[9] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[10] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*.

[11] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT'18)*. USENIX Association.

[12] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005*, Dong Ho Won and Seungjoo Kim (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–168.

[13] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association.