Short paper: Principled Detection of Speculative Information Flows

Marco Guarnieri*, Boris Köpf†, José F. Morales*, Jan Reineke‡, and Andrés Sánchez*
*IMDEA Software Institute †Microsoft Research Cambridge ‡Saarland University

I. Introduction

Speculative execution avoids expensive pipeline stalls by predicting the outcome of branching (and other) decisions, and by speculatively executing the corresponding instructions. If a prediction turns out to be wrong, the processor aborts the speculative execution and rolls back the effect of the speculatively executed instructions on the architectural (ISA) state, which consists of registers, flags, and main memory.

However, the speculative execution's effect on the microarchitectural state, which comprises the content of the cache, is not (or only partially) rolled back. This side effect can leak information about the speculatively accessed data and thus violate confidentiality. The family of SPECTRE attacks [1] demonstrates that this vulnerability affects all modern general-purpose processors and poses a serious threat for platforms with multiple tenants.

Since the advent of SPECTRE, a number of countermeasures have been proposed and deployed. At the software-level, these include, for instance, the insertion of serializing instructions [2], the use of branchless bounds checks [3], and speculative load hardening [4]. Several compilers support the automated insertion of these countermeasures during compilation [4]–[6], and the first static analyses to help identify vulnerable code patterns are emerging [7].

However, we still lack a precise characterization of *security against speculative execution attacks*. Such a characterization is a prerequisite for reasoning about the effectiveness of countermeasures, and for making principled decisions about their placement. It would enable one, for example, to identify cases where countermeasures do not prevent all attacks, or where they are unnecessary.

Our Approach: We develop a novel, principled approach for detecting information flows introduced by speculative execution, and for reasoning about software defenses against SPECTRE-style attacks. Our approach is backed by a semantic notion of security against speculative execution attacks, and it comes with an algorithm, based on symbolic execution, for proving the absence of speculative leaks.

Defining Security: The foundation of our approach is *speculative non-interference*, a novel semantic notion of security against speculative execution attacks. Speculative non-interference is based on comparing a program with respect to two different semantics:

- The first is a standard, *non-speculative semantics*. We use this semantics as a proxy for the intended program behavior.
- The second is a novel, *speculative semantics* that can follow mispredicted branches for a bounded number of steps before backtracking. We use this semantics to capture the effect of speculatively executed instructions.

In a nutshell, speculative non-interference requires that *speculatively executed instructions do not leak more information into the microarchitectural state than what the intended behavior does*, i.e., than what is leaked by the standard, non-speculative semantics.

To capture "leakage into the microarchitectural state", we consider an observer of the program execution that sees the locations of memory accesses and jump targets. This observer model is commonly used for characterizing "side-channel free" or "constant-time" code [8] in the absence of detailed models of the microarchitecture.

Under this observer model, an adversary may distinguish two initial program states if they yield different traces of memory locations and jump targets. *Speculative non-interference* (SNI) requires that two initial program states can only be distinguished under the speculative semantics if they can also be distinguished under the standard, non-speculative semantics. Concretely, SNI is a variant of non-interference where the non-speculative semantics specifies what a program, executed under the speculative semantics, is allowed to leak.

The speculative semantics, and hence SNI, depends on the decisions taken by a branch predictor. We show that one can abstract from the specific predictor by considering a worst-case predictor that mispredicts every branching decision. SNI w.r.t. this worst-case predictor implies SNI w.r.t. a large class of real-world branch predictors, without introducing false alarms.

Checking Speculative Non-Interference: We propose SPECTECTOR, an algorithm to automatically prove that programs satisfy SNI. Given a program p, SPECTECTOR uses symbolic execution with respect to the speculative semantics and the worst-case branch predictor to derive a concise representation of the traces of memory accesses and jump targets during execution along all possible program paths.

Based on this representation, SPECTECTOR creates an SMT formula that captures that, whenever two initial program states produce the same memory access patterns in the standard semantics, they also produce the same access patterns in the speculative semantics. Validity of this formula for each program path implies speculative noninterference.

Case studies: We implement a prototype of SPECTECTOR, with a front end for parsing (a subset of) x86 assembly and the Z3 SMT solver as a back end for solving SMT formulas.

```
1  if (y < size)
2     temp &= B[A[y] * 512];
```

Fig. 1. SPECTRE variant 1 - C code

```
   mov    size, %rax
   mov    y, %rbx
   cmp    %rbx, %rax
   jbe    END
   mov    A(%rbx), %rax
   shl    $9, %rax
   mov    B(%rax), %rax
   and    %rax, temp
```

Fig. 2. SPECTRE variant 1 - Assembly code

We perform two case studies where we evaluate the precision and scalability of SPECTECTOR.

- For evaluating precision, we analyze the 15 variants of SPECTRE v1 by Kocher [9]. We create a corpus of 240 microbenchmarks by compiling the 15 programs with the CLANG, INTEL ICC, and Microsoft VISUAL C++ compilers, using different levels of optimization and protection against SPECTRE. Using SPECTECTOR, we successfully (1) detect all leaks pointed out in [9], (2) detect novel, subtle leaks that are out of scope of existing approaches that check for known vulnerable code patterns [7], and (3) identify cases where compilers unnecessarily inject countermeasures, i.e., opportunities for optimization without sacrificing security.
- For evaluating scalability, we apply SPECTECTOR to the codebase of the Xen Project Hypervisor. Our evaluation indicates that the cost of checking speculative non-interference is comparable to that of discovering symbolic paths, which shows that our approach does not exhibit bottlenecks beyond those inherited by symbolic execution.

Scope: We focus on leaks introduced by speculatively executed instructions resulting from mispredicted branch outcomes, such as those exploited in SPECTRE v1 [1]. We discuss our formal model's validity in [10].

Summary of contributions: Our contributions are both theoretical and practical. On the theoretical side, we present *speculative non-interference*, the first semantic notion of security against speculative execution attacks. On the practical side, we develop SPECTECTOR, an automated technique for detecting speculative leaks (or prove their absence), and we use it to detect subtle leaks – and optimization opportunities – in the way compilers inject SPECTRE countermeasures.

Additional material: This paper is a short version of [10]. SPECTECTOR and the full paper describing our approach are available at `https://spectector.github.io`.

II. ILLUSTRATIVE EXAMPLE

To illustrate our approach, we show how SPECTECTOR applies to the SPECTRE v1 example [1] shown in Figure 1.

Spectre v1: The program checks whether the index stored in variable `y` is less than the size of the array `A`, stored in variable `size`. If that is the case, the program retrieves `A[y]`, amplifies it with a multiple (here: `512`) of the cache line size, and uses the result as an address for accessing the array `B`.

If `size` is not cached, evaluating the branch condition requires traditional processors to wait until `size` is fetched from main memory. Modern processors instead speculate on the condition's outcome and continue the computation. Hence, the memory accesses in line 2 may be executed even if $\texttt{y} \geq \texttt{size}$.

When `size` becomes available, the processor checks whether the speculated branch is the correct one. If it is not, it rolls back the architectural (i.e. ISA) state's changes and executes the correct branch. However, the speculatively executed memory accesses leave a footprint in the microarchitectural state, in particular in the cache, which enables an adversary to retrieve `A[y]`, even for $\texttt{y} \geq \texttt{size}$, by probing the array `B`.

Detecting Leaks with SPECTECTOR: SPECTECTOR automatically detects leaks introduced by speculatively executed instructions, or proves their absence. Specifically, SPECTECTOR detects a leak whenever executing the program under the speculative semantics, which captures that the execution can go down a mispredicted path for a bounded number of steps, leaks more information into the microarchitectural state than executing the program under a non-speculative semantics.

To illustrate how SPECTECTOR operates, we consider the x86 assembly[1] translation of Figure 1's program (cf. Figure 2).

SPECTECTOR performs symbolic execution with respect to the speculative semantics to derive a concise representation of the concrete traces of memory accesses and program counter values along each path of the program. These symbolic traces capture the program's effect on the microarchitectural state.

During speculative execution, the speculatively executed parts are determined by the predictions of the branch predictor. As we prove in [10], leakage due to speculative execution is maximized under a branch predictor that mispredicts every branch. The code in Figure 2 yields two symbolic traces w.r.t. the speculative semantics that mispredicts every branch:[2]

$$\textbf{start} \cdot \textbf{rollback} \cdot \tau \quad \text{when} \quad \texttt{y} < \texttt{size} \tag{1}$$

$$\textbf{start} \cdot \tau \cdot \textbf{rollback} \quad \text{when} \quad \texttt{y} \geq \texttt{size} \tag{2}$$

where $\tau = \textbf{load}\ (\texttt{A} + \texttt{y}) \cdot \textbf{load}\ (\texttt{B} + \texttt{A[y]} \ * \ \texttt{512})$. Here, the argument of **load** is visible to the observer, while **start** and **rollback** denote the start and the end of a misspeculated execution. The traces of the *non-speculative* semantics are obtained from those of the speculative semantics by removing all observations in between **start** and **rollback**.

Trace 1 shows that whenever `y` is in bounds (i.e., $\texttt{y} < \texttt{size}$) the observations of the speculative semantics and the non-speculative semantics coincide (i.e. they are both τ). In contrast, Trace 2 shows that whenever $\texttt{y} \geq \texttt{size}$, the speculative execution generates observations τ that depend on `A[y]` whose value is not visible in the non-speculative execution. This is flagged as a leak by SPECTECTOR.

[1]We use a simplified AT&T syntax without operand sizes

[2]For simplicity of presentation, the example traces capture only loads but not the program counter.

Proving Security with SPECTECTOR*:* The CLANG 7.0.0 C++ compiler implements a countermeasure, called speculative load hardening [4], that applies conditional masks to addresses to prevent leaks into the microarchitectural state. Figure 3 depicts the protected output of CLANG on the program from Figure 1.

```
1   mov     size, %rax
2   mov     y, %rbx
3   mov     $0, %rdx
4   cmp     %rbx, %rax
5   jbe     END
6   cmovbe  $-1, %rdx
7   mov     A(%rbx), %rax
8   shl     $9, %rax
9   or      %rdx, %rax
10  mov     B(%rax), %rax
11  or      %rdx, %rax
12  and     %rax, temp
```

Fig. 3. SPECTRE variant 1 - Assembly code with speculative load hardening. CLANG inserted instructions 3, 6, 9, and 11.

The symbolic execution of the speculative semantics produces, as before, Trace 1 and Trace 2, but with

$$\tau = \mathbf{load}\ (\mathtt{A} + \mathtt{y}) \cdot \mathbf{load}\ (\mathtt{B} + (\mathtt{A[y]\ \ *\ \ 512}) \,|\, \mathit{mask}),$$

where $\mathit{mask} = \mathbf{ite}(\mathtt{y} < \mathtt{size}, \mathtt{0x0}, \mathtt{0xFF..FF})$ corresponds to the conditional move in line 6 and | is a bitwise-or operator. Here, $\mathbf{ite}(\mathtt{y} < \mathtt{size}, \mathtt{0x0}, \mathtt{0xFF..FF})$ is a symbolic if-then-else expression evaluating to `0x0` if $\mathtt{y} < \mathtt{size}$ and to `0xFF..FF` otherwise.

The analysis of Trace 1 is as before. For Trace 2, however, SPECTECTOR determines (via a query to Z3 [11]) that, for all $\mathtt{y} \geq \mathtt{size}$ there is exactly *one* observation that the adversary can make during the speculative execution, namely $\mathbf{load}\ (\mathtt{A} + \mathtt{y}) \cdot \mathbf{load}\ (\mathtt{B} + \mathtt{0xFF..FF})$, from which it concludes that no information leaks into the microarchitectural state, i.e., the countermeasure is effective in securing the program.

III. CASE STUDY: COMPILER COUNTERMEASURES

Here, we report the results of using SPECTECTOR to analyze the security of compiler-level countermeasures on 15 variants of SPECTRE v1 [9]. We refer the reader to [10] for our case study evaluating SPECTECTOR's scalability.

A. Experimental Setup

For our analysis, we rely on three state-of-the-art compilers: Microsoft VISUAL C++ versions v19.15.26732.1 and v19.20.27317.96, Intel ICC v19.0.0.117, and CLANG v7.0.0.

We compile the programs using two different *optimization levels* (`-O0` and `-O2`) and three *mitigation levels*: (a) UNP: we compile without any SPECTRE mitigations. (b) FEN: we compile with automated injection of speculation barriers.[3] (c) SLH: we compile using speculative load hardening.[4]

[3]Fences are supported by CLANG with the flag `-x86-speculative-load-hardening-lfence`, by ICC with `-mconditional-branch=all-fix`, and by VISUAL C++ with `/Qspectre`.

[4]Speculative load hardening is supported by CLANG with the flag `-x86-speculative-load-hardening`.

Compiling each of the 15 examples from [9] with each of the 3 compilers, each of the 2 optimization levels, and each of the 2-3 mitigation levels, yields a corpus of 240 x64 assembly programs.[5] For each program, we specify a security policy that flags as "low" all registers and memory locations that can either be controlled by the adversary or can be assumed to be public. This includes variables `y` and `size`, and the base addresses of the arrays `A` and `B` as well as the stack pointer.

B. Experimental Results

Figure 4 depicts the results of applying SPECTECTOR to the 240 examples. We highlight the following findings:

- SPECTECTOR detects the speculative leaks in almost all unprotected programs, for all compilers (see the UNP columns). The exception is Example #8, which uses a conditional expression instead of the if statement of Figure 1:

```
temp &= B[A[y<size?(y+1):0]*512];
```

At optimization level `-O0`, this is translated to a (vulnerable) branch instruction by all compilers, and at level `-O2` to a (safe) conditional move, thus closing the leak.

- The CLANG and Intel ICC compilers defensively insert fences after each branch instruction, and SPECTECTOR can prove security for all cases (see the FEN columns for CLANG and ICC). In Example #8 with options `-O2` and FEN, ICC inserts an **lfence** instruction, even though the baseline relies on a conditional move, see line 10 below. This **lfence** is unnecessary according to our semantics, but may close leaks on processors that speculate over conditional moves.

```
1   mov     y, %rdi
2   lea     1(%rdi), %rdx
3   mov     size, %rax
4   xor     %rcx, %rcx
5   cmp     %rax, %rdi
6   cmovb   %rdx, %rcx
7   mov     temp, %r8b
8   mov     A(%rcx), %rsi
9   shl     $9, %rsi
10  lfence
11  and     B(%rsi), %r8b
12  mov     %r8b, temp
```

- For the VISUAL C++ compiler, SPECTECTOR automatically detects all leaks pointed out in [9] (see the FEN 19.15 `-O2` column for VCC). Our analysis differs from Kocher's only on Example #8, where the compiler v19.15.26732.1 introduces a safe conditional move, as explained above. Moreover, without compiler optimizations (which is not considered in [9]), SPECTECTOR establishes the security of Examples 3 and 5 (see the FEN 19.15 `-O0` column). The latest VCC compiler additionally mitigates the leaks in Examples #4, #12, and #14 (see the FEN 19.20 column).
- SPECTECTOR can prove the security of speculative load hardening in Clang (see the SLH column for CLANG), except for Example #10 with `-O2` and Example #15 with `-O0`.

[5]The programs are available at `https://spectector.github.io`.

Ex.	VCC						ICC				CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH	
	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
15	○	○	○	○	○	○	○	○	●	●	○	○	●	●	○	●

Fig. 4. Analysis of Kocher's examples [9] compiled with compilers and options. For each of the 15 examples, we analyzed the unpatched version (denoted by UNP), the version patched with speculation barriers (denoted by FEN), and the version patched using speculative load hardening (denoted by SLH). Programs have been compiled without optimizations (-O0) or with compiler optimizations (-O2) using the compilers VISUAL C++ (two versions), ICC, and CLANG. ○ denotes that SPECTECTOR detects a speculative leak, whereas ● indicates that SPECTECTOR proves the program secure.

Example 10 with Speculative Load Hardening: Example #10 differs from Figure 1 in that it leaks sensitive information into the microarchitectural state by conditionally reading the content of `B[0]`, depending on the value of `A[y]`.

```
if (y < size)
 if (A[y] == k)
  temp &= B[0];
```

SPECTECTOR proves the security of the program produced with CLANG -O0, and speculative load hardening.

However, at optimization level -O2, CLANG outputs the following code that SPECTECTOR reports as insecure.

```
1  mov     size, %rdx
2  mov     y, %rbx
3  mov     $0, %rax
4  cmp     %rbx, %rdx
5  jbe     END
6  cmovbe  $-1, %rax
7  or      %rax, %rbx
8  mov     k, %rcx
9  cmp     %rcx, A(%rbx)
10 jne     END
11 cmovne  $-1, %rax
12 mov     B, %rcx
13 and     %rcx, temp
14 jmp     END
```

The reason for this is that CLANG masks only the register `%rbx` that contains the index of the memory access `A[y]`, cf. lines 6–7. However, it does *not* mask the value that is read from `A[y]`. As a result, the comparison at line 9 speculatively leaks (via the jump target) whether the content of `A[0xFF...FF]` is `k`. SPECTECTOR detects this subtle leak and flags a violation of speculative noninterference.

While this example nicely illustrates the scope of SPECTECTOR, it is likely not a problem in practice. First, the adversary can only determine one bit of information about the content of a fixed memory location. Second, the leak may be mitigated by how data dependencies are handled in modern out-of-order CPUs. Specifically, the conditional move in line 6 relies on the comparison in line 4. If executing the conditional move effectively terminates speculation, the leak is spurious.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P 2019*.
[2] Intel, "Intel analysis of speculative execution side channels," 2018.
[3] "What spectre and meltdown mean for webkit," https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.
[4] C. Carruth, "Speculative load hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.
[5] 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.
[6] A. Pardoe, "Spectre mitigations in msvc," https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.
[7] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," *CoRR*, vol. abs/1807.05843, 2018.
[8] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2005, pp. 156–168.
[9] P. Kocher, "Spectre mitigations in Microsoft's C/C++ compiler," https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.
[10] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *S&P 2020*.
[11] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS 2008*.