SPECTECTOR: Principled Detection of Speculative Information Flows

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

***Abstract*—Since the advent of SPECTRE, a number of countermeasures have been proposed and deployed. Rigorously reasoning about their effectiveness, however, requires a well-defined notion of security against speculative execution attacks, which has been missing until now.**

In this paper (1) we put forward *speculative non-interference*, the first semantic notion of security against speculative execution attacks, and (2) we develop SPECTECTOR, an algorithm based on symbolic execution to automatically prove speculative non-interference, or to detect violations.

We implement SPECTECTOR in a tool, which we use to detect subtle leaks and optimizations opportunities in the way major compilers place SPECTRE countermeasures. A scalability analysis indicates that checking speculative non-interference does not exhibit fundamental bottlenecks beyond those inherited by symbolic execution.

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 is incorrect, 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]–[5] 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 [6], the use of branchless bounds checks [7], and speculative load hardening [8]. Several compilers support the automated insertion of these countermeasures during compilation [8]–[10], and the first static analyses to help identify vulnerable code patterns are emerging [11].

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 [12], [13] 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 be distinguished under the speculative semantics only if they can also be distinguished under the standard, non-speculative semantics.

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

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

Fig. 1. SPECTRE variant 1 - C code

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 non-interference.

Case studies: We implement a prototype of SPECTECTOR, with a front end for parsing (a subset of) x86 assembly and a back end for solving SMT formulas using the Z3 SMT solver. 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 [14]. 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 [14], (2) detect novel, subtle leaks that are out of scope of existing approaches that check for known vulnerable code patterns [11], 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 [2]. For an in-depth discussion of our approach's scope, see Section X.

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.

SPECTECTOR is available at `https://spectector.github.io`. An extended version of this paper containing proofs of the technical results is available at [15].

II. ILLUSTRATIVE EXAMPLE

To illustrate our approach, we show how SPECTECTOR applies to the SPECTRE v1 example [2] shown in Figure 1.

```
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

Spectre v1: The program checks whether the index stored in the variable `y` is less than the size of the array `A`, stored in the 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 shown in Section V-C, 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]

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

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

[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.

where $\tau = \texttt{load}\ (\texttt{A}+\texttt{y}) \cdot \texttt{load}\ (\texttt{B}+\texttt{A[y]}\ \texttt{*}\ \texttt{512})$. 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.

Proving security with SPECTECTOR*:* The CLANG 7.0.0 C++ compiler implements a countermeasure, called speculative load hardening [8], 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

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

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

The analysis of Trace 1 is as before. For Trace 2, however, SPECTECTOR determines (via a query to Z3 [16]) that, for all $\texttt{y} \geq \texttt{size}$ there is exactly *one* observation that the adversary can make during the speculative execution, namely $\texttt{load}\ (\texttt{A}+\texttt{y}) \cdot \texttt{load}\ (\texttt{B}+\texttt{0xFF..FF})$, from which it concludes that no information leaks into the microarchitectural state, i.e., the countermeasure is effective in securing the program. See Section VIII for examples where SPECTECTOR detects that countermeasures are not applied effectively.

Basic Types

(Registers)	x	$\in$	*Regs*
(Values)	n, ℓ	$\in$	$\mathit{Vals} = \mathbb{N} \cup \{\bot\}$

Syntax

(Expressions)	e	$:=$	$n \mid x \mid \ominus e \mid e_1 \otimes e_2$
(Instructions)	i	$:=$	$\mathbf{skip} \mid x \leftarrow e \mid \mathbf{load}\ x, e \mid \mathbf{store}\ x, e \mid \mathbf{jmp}\ e \mid \mathbf{beqz}\ x, \ell \mid x \xleftarrow{e'?} e \mid \mathbf{spbarr}$
(Programs)	p	$:=$	$n : i \mid p_1; p_2$

Fig. 4. μASM syntax

III. LANGUAGE AND SEMANTICS

We now introduce μASM, a core assembly language which we use for defining speculative non-interference and describing SPECTECTOR.

A. Syntax

The syntax of μASM is defined in Figure 4. Expressions are built from a set of register identifiers *Regs*, which contains a designated element $\mathbf{pc}$ representing the program counter, and a set *Vals* of values, which consists of the natural numbers and $\bot$. μASM features eight kinds of instructions: a $\mathbf{skip}$ instruction, (conditional) assignments, load and store instructions, branching instructions, indirect jumps, and speculation barriers $\mathbf{spbarr}$. Both conditional assignments and speculation barriers are commonly used to implement SPECTRE countermeasures [6], [8].

A μASM program is a sequence of pairs $n : i$, where i is an instruction and $n \in \mathbb{N}$ is a value representing the instruction's label. We say that a program is *well-formed* if (1) it does not contain duplicate labels, (2) it contains an instruction labeled with 0, i.e., the initial instruction, and (3) it does not contain branch instructions of the form $n : \mathbf{beqz}\ x, n+1$. In the following we consider only well-formed programs.

We often treat programs p as partial functions from natural numbers to instructions. Namely, given a program p and a number $n \in \mathbb{N}$, we denote by $p(n)$ the instruction labelled with n in p if it exists, and $\bot$ otherwise.

Example 1. The SPECTRE v1 example from Figure 1 can be expressed in μASM as follows:

$$\begin{aligned}
&0 : x \leftarrow \texttt{y} < \texttt{size}\\
&1 : \mathbf{beqz}\ x, \bot\\
&2 : \mathbf{load}\ z, \texttt{A} + \texttt{y}\\
&3 : z \leftarrow z * 512\\
&4 : \mathbf{load}\ w, \texttt{B} + z\\
&5 : \texttt{temp} \leftarrow \texttt{temp}\ \&\ w
\end{aligned}$$

Here, registers `y`, `size`, and `temp` store the respective variables. Similarly, registers `A` and `B` store the memory addresses of the first elements of the arrays `A` and `B`. ■

B. Non-speculative Semantics

The standard, non-speculative semantics models the execution of μASM programs on a platform without speculation. This semantics is formalized as a ternary relation $\sigma \xrightarrow{o} \sigma'$ mapping a configuration σ to a configuration σ', while producing an observation o. Observations are used to capture what an adversary can see about a given execution trace. We describe the individual components of the semantics below.

Configurations: A *configuration* σ is a pair $\langle m, a\rangle$ of a *memory* $m \in \mathit{Mem}$ and a *register assignment* $a \in \mathit{Assgn}$, modeling the state of the computation. Memories m are functions mapping memory addresses, represented by natural numbers, to values in Vals. Register assignments a are functions mapping register identifiers to values. We require that $\bot$ can only be assigned to the program counter $\mathbf{pc}$, signaling termination. A configuration $\langle m, a\rangle$ is *initial* (respectively *final*) if $a(\mathbf{pc}) = 0$ (respectively $a(\mathbf{pc}) = \bot$). We denote the set $\mathit{Mem} \times \mathit{Assgn}$ of all configurations by Conf.

Adversary model and observations: We consider an adversary that observes the program counter and the locations of memory accesses during computation. This adversary model is commonly used to formalize timing side-channel free code [12], [13], without requiring microarchitectural models. In particular, it captures leakage through caches without requiring an explicit cache model.

We model this adversary in our semantics by annotating transactions with observations `load` n and `store` n, which expose read and write accesses to an address n, and observations `pc` n, which expose the value of the program counter. We denote the set of all observations by Obs.

Evaluation relation: We describe the execution of μASM programs using the evaluation relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Obs} \times \mathit{Conf}$. Most of the rules defining $\rightarrow$ are fairly standard, which is why Figure 5 presents only a selection. We refer the reader to Appendix A for the remaining rules.

The rules LOAD and STORE describe the behavior of instructions $\mathbf{load}\ x, e$ and $\mathbf{store}\ x, e$ respectively. The former assigns to the register x the memory content at the address n to which expression e evaluates; the latter stores the content of x at that address. Both rules expose the address n using observations and increment the program counter.

The rule CONDUPDATE-SAT describes the behavior of a conditional update $x \xleftarrow{e'?} e$ whose condition e' is satisfied. It first checks that the condition e' evaluates to 0. It then updates the register assignment a by storing in x the value of e, and by incrementing $\mathbf{pc}$.

The rule BEQZ-SAT describes the effect of the instruction $\mathbf{beqz}\ x, \ell$ when the branch is taken. Under the condition that x evaluates to 0, it sets the program counter to ℓ and exposes this change using the observation `pc` ℓ.

Finally, the rule JMP executes $\mathbf{jmp}\ e$ instructions. The rule stores the value of e in the program counter and records this change using the observation `pc` ℓ.

Runs and traces: The evaluation relation captures individual steps in the execution of a program. Runs capture full executions of the program. We formalize them as triples $\langle \sigma, \tau, \sigma'\rangle$ consisting of an initial configuration σ, a trace of observations τ, and a final configuration σ'. Given a program p, we denote by $(\!|p|\!)$ the set of all possible runs of the non-speculative semantics, i.e., it contains all triples $\langle \sigma, \tau, \sigma'\rangle$ corresponding to executions $\sigma \xrightarrow{\tau}{}^{*} \sigma'$. Finally, we denote by $(\!|p|\!)(\sigma)$ the trace τ such that there is a final configuration σ' for which $\langle \sigma, \tau, \sigma'\rangle \in (\!|p|\!)$. In this paper, we only consider terminating programs. Extending the definitions and algorithms to non-terminating programs is future work.

IV. SPECULATIVE SEMANTICS

This section introduces a model of speculation that captures the execution of μASM programs on speculative in-order microarchitectures. We first informally explain this model in Section IV-A before formalizing it in the rest of the section.

A. Modeling speculation

Non-branching instructions are executed as in the standard semantics. Upon reaching a branching instruction, the *prediction oracle*, which is a parameter of our model, is queried to obtain a branch prediction that is used to decide which of the two branches to execute speculatively.

To enable a subsequent rollback in case of a misprediction, a snapshot of the current program configuration is taken, before starting a *speculative transaction*. In this speculative transaction, the program is executed speculatively along the predicted branch for a bounded number of computation steps. Computing the precise length w of a speculative transactions would (among other aspects) require a detailed model of the memory hierarchy. To abstract from this complexity, in our model w is also provided by the prediction oracle.

At the end of a speculative transaction, the correctness of the prediction is evaluated:

- If the prediction was *correct*, the transaction is committed and the computation continues using the current configuration.
- If the prediction was *incorrect*, the transaction is aborted, the original configuration is restored, and the computation continues on the correct branch.

In the following we formalize the behavior intuitively described above in the *speculative semantics*. The main technical challenge lies in catering for nested branches and transactions.

B. Prediction oracles

In our model, prediction oracles serve two distinct purposes: (1) predicting branches, and (2) determining the speculative transactions' lengths. A *prediction oracle* $\mathcal{O}$ is a partial function that takes as input a program p, a branching history h, and a label ℓ such that $p(\ell)$ is a branching instruction, and that returns as output a pair $\langle \ell', w\rangle \in \mathit{Vals} \times \mathbb{N}$, where ℓ' represents the predicted branch (i.e., $\ell' \in \{\ell+1, \ell''\}$ where $p(\ell) = \mathbf{beqz}\ x, \ell''$) and w represents the speculative transaction's length.

Taking into account the *branching history* enables us to capture history-based branch predictors, a general class of branch predictors that base their decisions on the sequence of branches leading up to a branching instruction. Formally,

$$\textsc{Load}\quad \frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad n = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{load}\ n} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle}$$

$$\textsc{Store}\quad \frac{p(a(\mathbf{pc})) = \mathbf{store}\ x, e \qquad n = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{store}\ n} \langle m[n \mapsto a(x)], a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

$$\textsc{CondUpdate-Sat}\quad \frac{p(a(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](a) = 0 \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto [\![e]\!](a)]\rangle}$$

$$\textsc{Beqz-Sat}\quad \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad a(x) = 0}{\langle m, a\rangle \xrightarrow{\texttt{pc}\ \ell} \langle m, a[\mathbf{pc} \mapsto \ell]\rangle}$$

$$\textsc{Jmp}\quad \frac{p(a(\mathbf{pc})) = \mathbf{jmp}\ e \qquad \ell = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{pc}\ \ell} \langle m, a[\mathbf{pc} \mapsto \ell]\rangle}$$

Fig. 5. Standard semantics for μASM program p – selected rules

a branching history is a sequence of triples $\langle \ell, id, \ell'\rangle$, where $\ell \in \mathit{Vals}$ is the label of a branching instruction, $\ell' \in \mathit{Vals}$ is the label of the predicted branch, and $id \in \mathbb{N}$ is the identifier of the transaction in which the branch is executed.

A prediction oracle $\mathcal{O}$ has *speculative window at most* w if the length of the transactions generated by its predictions is at most w, i.e., for all programs p, branching histories h, and labels ℓ, $\mathcal{O}(p, h, \ell) = \langle \ell', w'\rangle$, for some ℓ' and with $w' \leq w$.

Example 2. The "backward taken forward not taken" (BTFNT) branch predictor, implemented in early CPUs [17], predicts the branch as taken if the target instruction address is lower than the program counter. It can be formalized as part of a prediction oracle $BTFNT$, for a fixed speculative window w, as follows: $BTFNT(p, h, \ell) = \langle min(\ell + 1, \ell'), w\rangle$, where $p(\ell) = \mathbf{beqz}\ x, \ell'$. ■

Dynamic branch predictors, such as simple 2-bit predictors and more complex correlating or tournament predictors [17], can also be formalized using prediction oracles.

C. Speculative transactions

To manage each ongoing speculative transaction[3], the speculative semantics needs to remember a snapshot σ of the configuration prior to the start of the transaction, the length w of the transaction (i.e., the number of instructions left to be executed in this transaction), the branch prediction ℓ used at the start of the transaction, and the transaction's identifier id. We call such a 4-tuple $\langle \sigma, id, w, \ell\rangle \in \mathit{Conf} \times \mathbb{N} \times \mathbb{N} \times \mathit{Vals}$, a *speculative state*, and we denote by SpecS the set of all speculative states.

Nested transactions are represented by sequences of speculative states. We use standard notation for sequences: S^* is the set of all finite sequences over the set S, ε is the empty sequence, and $s_1 \cdot s_2$ is the concatenation of sequences s_1 and s_2.

We use the following two helper functions to manipulate sequences of speculative states $s \in \mathit{SpecS}^*$:

- $decr : \mathit{SpecS}^* \to \mathit{SpecS}^*$ decrements by 1 the length of all transactions in the sequence.
- $zeroes : \mathit{SpecS}^* \to \mathit{SpecS}^*$ sets to 0 the length of all transactions in the sequence.
- The predicate $enabled(s)$ holds if and only if none of the transactions in s has remaining length 0.

In addition to branch and jump instructions, speculative transactions can also modify the program counter: rolling back a transaction results in resetting the program counter to the one in the correct branch. To expose such changes to the adversary, we extend the set Obs of observations with elements of the form `start` id, `commit` id, and `rollback` id, to denote start, commit, and rollback of a speculative transaction id. ExtObs denotes the set of extended observations.

D. Evaluation relation

The speculative semantics operates on *extended configurations*, which are 4-tuples $\langle ctr, \sigma, s, h\rangle \in \mathit{ExtConf}$ consisting of a global counter $ctr \in \mathbb{N}$ for generating transaction identifiers, a configuration $\sigma \in \mathit{Conf}$, a sequence s of speculative states representing the ongoing speculative transactions, and a branching history h. Along the lines of the standard semantics, we describe the speculative semantics of μASM programs under a prediction oracle $\mathcal{O}$ using the relation $\rightsquigarrow \subseteq \mathit{ExtConf} \times \mathit{ExtObs}^* \times \mathit{ExtConf}$. The rules are given in Figure 6 and are explained below:

SE-NoBranch captures the behavior of non-branching instructions as long as the length of all speculative states in s is greater than 0, that is, as long as $enabled(s)$ holds. In this case, $\rightsquigarrow$ mimics the behavior of the non-speculative semantics $\to$. If the instruction is not a speculation barrier, the lengths of all speculative transactions are decremented by 1 using $decr$. In contrast, if the instruction is a speculation barrier **spbarr**, the length of all transactions is set to 0 using $zeroes$. In this way, **spbarr** forces the termination (either with a commit or with a rollback) of all ongoing speculative transactions.

SE-Branch models the behavior of branch instructions. The rule (1) queries the prediction oracle $\mathcal{O}$ to obtain a prediction $\langle \ell, w\rangle$ consisting of the predicted next instruction address ℓ and the length of the transaction w, (2) sets the program counter to ℓ, (3) decrements the length of the transactions in s, (4) increments the transaction counter ctr, (5) appends a new speculative state with configuration σ, identifier ctr, transaction's length w, and predicted instruction address ℓ, and (6) updates the branching history by appending an entry $\langle a(\mathbf{pc}), ctr, \ell\rangle$ modeling the prediction. The rule also records the start of the speculative execution and the change of the program counter through observations.

SE-Commit captures a speculative transaction's commit. It is executed whenever a speculative state's remaining length reaches 0. Application of the rule requires that the prediction made for the transaction is correct, which is checked by

[3]Due to nesting, multiple transactions may be happening simultaneously.

comparing the predicted address ℓ with the one obtained by executing one step of the non-speculative semantics starting from the configuration σ'. The rule records the transaction's commit through an observation, and it updates the branching history according to the branch decision that has been taken.

SE-ROLLBACK captures a speculative transaction's rollback. The rule checks that the prediction is incorrect (again by comparing the predicted address ℓ with the one obtained from the non-speculative semantics), and it restores the configuration stored in s. Rolling back a transaction also terminates the speculative execution of all the nested transactions. This is modeled by dropping the portion s' of the speculative state associated with the nested transactions. The rule also produces observations recording the transaction's rollback and the change of the program counter, and it updates the branching history by recording the branch instruction's correct outcome.

Runs and traces: Runs and traces are defined analogously to the non-speculative case: Given a program p and an oracle $\mathcal{O}$, we denote by $[\![p]\!]_{\mathcal{O}}$ the set of all possible runs of the speculative semantics. By $[\![p]\!]_{\mathcal{O}}(\sigma)$ we denote the trace τ such that there is a final configuration σ' for which $\langle \sigma, \tau, \sigma' \rangle \in [\![p]\!]_{\mathcal{O}}$.

Example 3. For illustrating the speculative semantics, we execute the program from Ex. 1 with the oracle from Ex. 2 and a configuration $\langle 0, \langle m, a \rangle, \varepsilon, \varepsilon \rangle$ where $a(\mathtt{y}) \geq a(\mathtt{size})$.

First, the rule SE-NOJUMP is applied to execute the assignment $x \leftarrow \mathtt{y} < \mathtt{size}$. Then, the branch instruction $\mathbf{beqz}\ x, \bot$ is reached and so rule SE-JUMP applies. This produces the observations $\mathtt{start}\ 0$, modeling the beginning of a speculative transaction with id 0, and $\mathtt{pc}\ 2$, representing the program counter's change. Next, rule SE-NOJUMP applies three times to execute the instructions 2–5, thereby producing the observations $\mathtt{load}\ v_1$ and $\mathtt{load}\ v_2$ that record the memory accesses. Finally, rule SE-ROLLBACK applies, which terminates the speculative transaction and rolls back its effects. This rule produces the observations $\mathtt{rollback}\ 0$ and $\mathtt{pc}\ \bot$. Thus, executing the program produces the trace:

$$\tau := \mathtt{start}\ 0 \cdot \mathtt{pc}\ 2 \cdot \mathtt{load}\ v_1 \cdot \mathtt{load}\ v_2 \cdot \mathtt{rollback}\ 0 \cdot \mathtt{pc}\ \bot\ ,$$

where $v_1 = a(\mathtt{A}) + a(\mathtt{y})$ and $v_2 = a(\mathtt{B}) + m(v_1) * 512$. ■

E. Speculative and Non-speculative Semantics

We conclude this section by connecting the speculative and non-speculative semantics. For this, we introduce two projections of speculative traces τ:

- the *non-speculative projection* $\tau\restriction_{nse}$ is the trace obtained by removing from τ (1) all substrings that correspond to rolled-back transactions, i.e. all substrings $\mathtt{start}\ id \cdot \tau' \cdot \mathtt{rollback}\ id$, and (2) all extended observations.
- the *speculative projection* $\tau\restriction_{se}$ is the trace produced by rolled-back transactions, i.e. the complement of $\tau\restriction_{nse}$.

We lift projections $\restriction_{se}$ and $\restriction_{nse}$ to sets of runs in the natural way. Then, a program's non-speculative behavior can be obtained from its speculative behavior by dropping all speculative observations, i.e., by applying $\tau\restriction_{nse}$ to all of its runs τ:

Proposition 1. *Let p be a program and $\mathcal{O}$ be a prediction oracle. Then, $(\!|p|\!) = [\![p]\!]_{\mathcal{O}}\restriction_{nse}$.*

V. SPECULATIVE NON-INTERFERENCE

This section introduces *speculative non-interference* (SNI), a semantic notion of security characterizing those information leaks that are introduced by speculative execution.

A. Security policies

Speculative non-interference is parametric in a policy that specifies which parts of the configuration are known or controlled by an adversary, i.e., "public" or "low" data.

Formally, a security policy P is a finite subset of $\mathit{Regs} \cup \mathbb{N}$ specifying the low register identifiers and memory addresses. Two configurations $\sigma, \sigma' \in \mathit{Conf}$ are *indistinguishable* with respect to a policy P, written $\sigma \sim_P \sigma'$, iff they agree on all registers and memory locations in P.

Example 4. A policy P for the program from Example 1 may state that the content of the registers $\mathtt{y}$, $\mathtt{size}$, $\mathtt{A}$, and $\mathtt{B}$ is non-sensitive, i.e., $P = \{\mathtt{y}, \mathtt{size}, \mathtt{A}, \mathtt{B}\}$. ■

Policies need not be manually specified but can in principle be inferred from the context in which a piece of code executes, e.g., whether a variable is reachable from public input or not.

B. Speculative non-interference

Speculative non-interference requires that executing a program under the speculative semantics does not leak more information than executing the same program under the non-speculative semantics. Formally, whenever two indistinguishable configurations produce the same non-speculative traces, then they must also produce the same speculative traces.

Definition 1. A program p satisfies *speculative non-interference* for a prediction oracle $\mathcal{O}$ and a security policy P iff for all initial configurations $\sigma, \sigma' \in \mathit{InitConf}$, if $\sigma \sim_P \sigma'$ and $(\!|p|\!)(\sigma) = (\!|p|\!)(\sigma')$, then $[\![p]\!]_{\mathcal{O}}(\sigma) = [\![p]\!]_{\mathcal{O}}(\sigma')$.

Speculative non-interference is a variant of non-interference. While non-interference compares what is leaked by a program with a policy specifying the allowed leaks, speculative non-interference compares the program leakage under two semantics, the non-speculative and the speculative one. The security policy and the non-speculative semantics, together, specify what the program may leak under the speculative semantics.[4]

Example 5. The program p from Example 1 does not satisfy speculative non-interference for the BTFNT oracle from Example 2 and the policy P from Example 4. Consider two initial configurations $\sigma := \langle m, a \rangle, \sigma' := \langle m', a' \rangle$ that agree on the values of $\mathtt{y}$, $\mathtt{size}$, $\mathtt{A}$, and $\mathtt{B}$ but disagree on the value of $\mathtt{B}[\mathtt{A}[\mathtt{y}] * 512]$. Say, for instance, that $m(a(\mathtt{A}) + a(\mathtt{y})) = 0$ and $m'(a'(\mathtt{A}) + a'(\mathtt{y})) = 1$. Additionally, assume that $\mathtt{y} \geq \mathtt{size}$.

Executing the program under the non-speculative semantics produces the trace $\mathtt{pc}\ \bot$ when starting from σ and σ'.

[4]Conceptually, the non-speculative semantics can be seen as a declassification assertion for the speculative semantics [18].

$$\textsc{Se-NoBranch}\quad \frac{p(\sigma(\mathbf{pc})) \neq \mathbf{beqz}\ x,\ell \qquad \sigma \xrightarrow{\tau} \sigma' \qquad \mathit{enabled}(s) \qquad s' = \begin{cases} \mathit{decr}(s) & \text{if } p(\sigma(\mathbf{pc})) \neq \mathbf{spbarr} \\ \mathit{zeroes}(s) & \text{otherwise} \end{cases}}{\langle \mathit{ctr}, \sigma, s, h\rangle \overset{\tau}{\rightsquigarrow} \langle \mathit{ctr}, \sigma', s', h\rangle}$$

$$\textsc{Se-Branch}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x,\ell' \qquad \mathcal{O}(p,h,\sigma(\mathbf{pc})) = \langle \ell, w\rangle \qquad \sigma = \langle m, a\rangle \qquad \mathit{enabled}(s) \qquad s' = \mathit{decr}(s)\cdot\langle \sigma, \mathit{ctr}, w, \ell\rangle \qquad \mathit{id} = \mathit{ctr}}{\langle \mathit{ctr}, \sigma, s, h\rangle \overset{\texttt{start}\ \mathit{id}\cdot\texttt{pc}\ \ell}{\rightsquigarrow} \langle \mathit{ctr}+1, \langle m, a[\mathbf{pc}\mapsto \ell]\rangle, s', h\cdot\langle a(\mathbf{pc}), \mathit{id}, \ell\rangle\rangle}$$

$$\textsc{Se-Commit}\quad \frac{\sigma' \xrightarrow{\tau} \langle m, a\rangle \qquad \ell = a(\mathbf{pc}) \qquad \mathit{enabled}(s') \qquad h' = h\cdot\langle \sigma'(\mathbf{pc}), \mathit{id}, a(\mathbf{pc})\rangle}{\langle \mathit{ctr}, \sigma, s\cdot\langle \sigma', \mathit{id}, 0, \ell\rangle\cdot s', h\rangle \overset{\texttt{commit}\ \mathit{id}}{\rightsquigarrow} \langle \mathit{ctr}, \sigma, s\cdot s', h'\rangle}$$

$$\textsc{Se-Rollback}\quad \frac{\sigma' \xrightarrow{\tau} \langle m, a\rangle \qquad \ell \neq a(\mathbf{pc}) \qquad \mathit{enabled}(s') \qquad h' = h\cdot\langle \sigma'(\mathbf{pc}), \mathit{id}, a(\mathbf{pc})\rangle}{\langle \mathit{ctr}, \sigma, s\cdot\langle \sigma', \mathit{id}, 0, \ell\rangle\cdot s', h\rangle \overset{\texttt{rollback}\ \mathit{id}\cdot\texttt{pc}\ a(\mathbf{pc})}{\rightsquigarrow} \langle \mathit{ctr}, \langle m, a\rangle, s, h'\rangle}$$

Fig. 6. Speculative execution for μASM for a program p and a prediction oracle $\mathcal{O}$

Moreover, the two initial configurations are indistinguishable with respect to the policy P. However, executing p under the speculative semantics produces two distinct traces $\tau = \texttt{start}\ 0\cdot\texttt{pc}\ 3\cdot\texttt{load}\ v_1\cdot\texttt{load}\ (a'(\texttt{B})+0)\cdot\texttt{rollback}\ 0\cdot\texttt{pc}\ \bot$ and $\tau' = \texttt{start}\ 0\cdot\texttt{pc}\ 3\cdot\texttt{load}\ v_1\cdot\texttt{load}\ (a'(\texttt{B})+1)\cdot\texttt{rollback}\ 0\cdot\texttt{pc}\ \bot$, where $v_1 = a(\texttt{A}) + a(\texttt{y}) = a'(\texttt{A}) + a'(\texttt{y})$. Therefore, p does not satisfy speculative non-interference. ∎

C. Always-mispredict speculative semantics

The speculative semantics and SNI are parametric in the prediction oracle $\mathcal{O}$. Often, it is desirable obtaining guarantees w.r.t. *any* prediction oracle, since branch prediction models in modern CPUs are unavailable and as different CPUs employ different predictors. To this end, we introduce a variant of the speculative semantics that facilitates such an analysis.

Intuitively, leakage due to speculative execution is maximized under a branch predictor that mispredicts every branch. This intuition holds true unless speculative transactions are nested, where a correct prediction of a nested branch sometimes yields more leakage than a misprediction.

Example 6. Consider the following variation of the SPECTRE v1 example [2] from Figure 1, and assume that the function `benign()` runs for longer than the speculative window and does not leak any information.

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

Then, under a branch predictor that mispredicts every branch, the speculative transaction corresponding to the outer branch will be rolled back before reaching line 4. On the other hand, given a correct prediction of the inner branch, line 4 would be reached and a speculative leak would be present. ∎

A simple but inefficient approach to deal with this challenge would be to consider both cases, correct and incorrect prediction, upon every branch. This, however, would result in an exponential explosion of the number of paths to consider.

To avoid this, we introduce the *always-mispredict semantics* that differs from the speculative semantics in three key ways:

(1) It mispredicts every branch, hence its name. In particular, it is not parametric in the prediction oracle.

(2) It initializes the length of every *non-nested* transaction to w, and the length of every *nested* transaction to the remaining length of its enclosing transaction decremented by 1.

(3) Upon executing instructions, only the remaining length of the innermost transaction is decremented.

The consequence of these modifications is that nested transactions do not reduce the number of steps that the semantics may explore the correct path for, after the nested transactions have been rolled back. In Example 6, after rolling back the nested speculative transaction, the outer transaction continues as if the nested branch had been correctly predicted in the first place, and thus the speculative leak in line 4 is reached.

Modifications (1)-(3) are formally captured in the three rules AM-NOBRANCH, AM-BRANCH, and AM-ROLLBACK given in Appendix C. Similarly to $[\![p]\!]_{\mathcal{O}}(\sigma)$, we denote by $\{\!|p|\!\}_w(\sigma)$ the trace of observations obtained by executing the program p, starting from initial configuration σ according to the always-mispredict evaluation relation with speculative window w.

Theorem 1 states that checking SNI w.r.t. the always-mispredict semantics is sufficient to obtain security guarantees w.r.t. all prediction oracles.

Theorem 1. *A program p satisfies SNI for a security policy P and all prediction oracles $\mathcal{O}$ with speculative window at most w iff for all initial configurations $\sigma, \sigma' \in \mathit{InitConf}$, if $\sigma \sim_P \sigma'$ and $(\!|p|\!)(\sigma) = (\!|p|\!)(\sigma')$, then $\{\!|p|\!\}_w(\sigma) = \{\!|p|\!\}_w(\sigma')$.*

In our case studies in Sections VIII and IX, we use $w = 200$. This is motivated by typical sizes of the reorder buffer [19], which limits the lengths of speculative transactions in modern microarchitectures.

VI. DETECTING SPECULATIVE INFORMATION FLOWS

We now present SPECTECTOR, an approach to detect speculative leaks, or to prove their absence. SPECTECTOR symbolically executes the program p under analysis to derive a concise representation of p's behavior as a set of symbolic traces. It analyzes each symbolic trace using an SMT solver to detect possible speculative leaks through memory accesses or control-flow instructions. If neither memory nor control leaks are detected, SPECTECTOR reports the program as secure.

A. Symbolically executing μASM programs

We symbolically execute programs w.r.t. the *always mispredict* semantics, which enables us to derive security guarantees that hold for arbitrary prediction oracles, see Theorem 1. Our symbolic execution engine relies on the following components:

- A *symbolic expression* se is a concrete value $n \in \mathit{Vals}$, a symbolic value $s \in \mathit{SymbVals}$, an if-then-else expression $\mathbf{ite}(se, se', se'')$, or the application of unary or binary operators to symbolic expressions.
- A *symbolic memory* is a term in the standard theory of arrays [20]. A memory update $\mathbf{write}(sm, se, se')$ updates the symbolic memory sm by assigning the symbolic value se' to the symbolic address se. We extend symbolic expressions with memory reads $\mathbf{read}(sm, se)$, which retrieve the value of the symbolic address se from the symbolic memory sm.
- A *symbolic trace* τ is a sequence of symbolic observations of the form $\texttt{load}\ se$ or $\texttt{store}\ se$, symbolic branching conditions of the form $\texttt{symPc}(se)$, and transaction-related observations of the form $\texttt{start}\ n$ and $\texttt{rollback}\ n$, for natural numbers n and symbolic expressions se.
- The path condition $\mathit{pthCnd}(\tau)=\bigwedge_{\texttt{symPc}(se)\in\tau} se$ of trace τ is the conjunction of all symbolic branching conditions in τ.
- The symbolic execution derives symbolic runs $\langle\sigma, \tau, \sigma'\rangle$, consisting of symbolic configurations σ, σ' and a symbolic trace τ. The set of all symbolic runs forms the symbolic semantics, which we denote by $\{\!|p|\!\}_w^{symb}$. The derivation rules are fairly standard and are given in Appendix D.
- The value of an expression se depends on a *valuation* $\mu : \mathit{SymbVals} \to \mathit{Vals}$ mapping symbolic values to concrete ones. The evaluation $\mu(se)$ of se under μ is standard and formalized in Appendix D.
- A symbolic expression se is *satisfiable*, written $\mu \models se$, if there is a valuation μ such that $\mu(se) \neq 0$. Every valuation that satisfies a symbolic run's path condition maps the run to a concrete run. We denote by $\gamma(\langle\sigma, \tau, \sigma'\rangle)$ the set $\{\langle\mu(\sigma), \mu(\tau), \mu(\sigma')\rangle \mid \mu \models \mathit{pthCnd}(\tau)\}$ of $\langle\sigma, \tau, \sigma'\rangle$'s concretizations, and we lift it to $\{\!|p|\!\}_w^{symb}$. The concretization of the symbolic runs yields the set of all concrete runs:

Proposition 2. *Let p be a program and $w \in \mathbb{N}$ be a speculative window. Then, $\{\!|p|\!\}_w = \gamma(\{\!|p|\!\}_w^{symb})$.*

The proof of Proposition 2 is given in [15].

Example 7. Executing the program from Example 1 under the symbolic speculative semantics with speculative window 2 yields the following two symbolic traces: $\tau_1 := \texttt{symPc}(\texttt{y} < \texttt{size}) \cdot \texttt{start}\ 0 \cdot \texttt{pc}\ 2 \cdot \texttt{pc}\ 10 \cdot \texttt{rollback}\ 0 \cdot \texttt{pc}\ 3 \cdot \texttt{load}\ \texttt{A} + \texttt{y} \cdot \texttt{load}\ \texttt{B} + \mathbf{read}(sm, (\texttt{A} + \texttt{y})) * 512$, and $\tau_2 := \texttt{symPc}(\texttt{y} \geq \texttt{size}) \cdot \texttt{start}\ 0 \cdot \texttt{pc}\ 3 \cdot \texttt{load}\ \texttt{A} + \texttt{y} \cdot \texttt{load}\ \texttt{B} + \mathbf{read}(sm, (\texttt{A} + \texttt{y})) * 512 \cdot \texttt{rollback}\ 0 \cdot \texttt{pc}\ 2 \cdot \texttt{pc}\ 10$. ■

B. Checking speculative non-interference

SPECTECTOR is given in Algorithm 1. It relies on two procedures: MEMLEAK and CTRLLEAK, to detect leaks resulting

Algorithm 1 SPECTECTOR

Input: A program p, a security policy P, a speculative window $w \in \mathbb{N}$.

Output: SECURE if p satisfies speculative non-interference with respect to the policy P; INSECURE otherwise

1: **procedure** SPECTECTOR(p, P, w)
2:     **for** each symbolic run $\langle\sigma, \tau, \sigma'\rangle \in \{\!|p|\!\}_w^{symb}$ **do**
3:         **if** MEMLEAK(τ, P) $\vee$ CTRLLEAK(τ, P) **then**
4:             **return** INSECURE
5:     **return** SECURE

6: **procedure** MEMLEAK(τ, P)
7:     $\psi \leftarrow \mathit{pthCnd}(\tau)_{1\wedge 2} \wedge \mathit{polEqv}(P) \wedge \mathit{obsEqv}(\tau{\upharpoonright_{nse}}) \wedge \neg \mathit{obsEqv}(\tau{\upharpoonright_{se}})$
8:     **return** SATISFIABLE(ψ)

9: **procedure** CTRLLEAK(τ, P)
10:     **for** each prefix $\nu \cdot \texttt{symPc}(se)$ of $\tau{\upharpoonright_{se}}$ **do**
11:         $\psi \leftarrow \mathit{pthCnd}(\tau{\upharpoonright_{nse}} \cdot \nu)_{1\wedge 2} \wedge \mathit{polEqv}(P) \wedge \mathit{obsEqv}(\tau{\upharpoonright_{nse}}) \wedge \neg \mathit{sameSymbPc}(se)$
12:         **if** SATISFIABLE(ψ) **then**
13:             **return** $\top$
14:     **return** $\bot$

from memory and control-flow instructions, respectively. We start by discussing the SPECTECTOR algorithm and next explain the MEMLEAK and CTRLLEAK procedures.

SPECTECTOR takes as input a program p, a policy P specifying the non-sensitive information, and a speculative window w. The algorithm iterates over all symbolic runs produced by the symbolic always-mispredict speculative semantics (lines 2-4). For each run $\langle\sigma, \tau, \sigma'\rangle$, the algorithm checks whether τ speculatively leaks information through memory accesses or control-flow instructions. If this is the case, then SPECTECTOR has found a witness of a speculative leak and it reports p as INSECURE. If none of the traces contain speculative leaks, the algorithms terminates returning SECURE (line 5).

Detecting leaks caused by memory accesses: The procedure MEMLEAK takes as input a trace τ and a policy P, and it determines whether τ leaks information through symbolic $\texttt{load}$ and $\texttt{store}$ observations. The check is expressed as a satisfiability check of a constraint ψ. The construction of ψ is inspired by self-composition [21], which reduces reasoning about *pairs* of program runs to reasoning about single runs by replacing each symbolic variable x with two copies x_1 and x_2. We lift the subscript notation to symbolic expressions.

The constraint ψ is the conjunction of four formulas:

- $\mathit{pthCnd}(\tau)_{1\wedge 2}$ stands for $\mathit{pthCnd}(\tau)_1 \wedge \mathit{pthCnd}(\tau)_2$, which ensures that both runs follow the path associated with τ.
- $\mathit{polEqv}(P)$ introduces constraints $x_1 = x_2$ for each register $x \in P$ and $\mathbf{read}(sm_1, n) = \mathbf{read}(sm_2, n)$ for each memory location $n \in P$, which ensure that both runs agree on all non-sensitive inputs.
- $\mathit{obsEqv}(\tau{\upharpoonright_{nse}})$ introduces a constraint $se_1 = se_2$ for

each load se or store se in $\tau\lceil_{nse}$, which ensures that the non-speculative observations associated with memory accesses are the same in both runs.

- $\neg obsEqv(\tau\lceil_{se})$ ensures that speculative observations associated with memory accesses differ among the two runs.

If ψ is satisfiable, there are two P-indistinguishable configurations that produce the same non-speculative traces (since $pthCnd(\tau)_{1\wedge 2} \wedge polEqv(P) \wedge obsEqv(\tau\lceil_{nse})$ is satisfied) and whose speculative traces differ in a memory access observation (since $\neg obsEqv(\tau\lceil_{se})$ is satisfied), i.e. a violation of SNI.

Detecting leaks caused by control-flow instructions: To detect leaks caused by control-flow instructions, CTRLLEAK checks whether there are two traces in τ's concretization that agree on the outcomes of all non-speculative branch and jump instructions, while differing in the outcome of at least one speculatively executed branch or jump instruction.

In addition to $pthCnd(\tau)$, $obsEqv(\tau)$, and $polEqv(P)$, the procedure relies on the function $sameSymbPc(se)$ that introduces the constraint $se_1 \leftrightarrow se_2$ ensuring that se is satisfied in one concretization iff it is satisfied in the other.

CTRLLEAK checks, for each prefix $\nu \cdot$ symPc(se) in τ's speculative projection $\tau\lceil_{se}$, the satisfiability of the conjunction of $pthCnd(\tau\lceil_{nse} \cdot \nu)_{1\wedge 2}$, $polEqv(P)$, $obsEqv(\tau\lceil_{nse})$, and $\neg sameSymbPc(se)$. Whenever the formula is satisfiable, there are two P-indistinguishable configurations that produce the same non-speculative traces, but whose speculative traces differ on program counter observations, i.e. a violation of SNI.

Example 8. Consider the trace $\tau_2 :=$ symPc($\mathtt{y} \geq \mathtt{size}$) $\cdot$ start $0 \cdot$ pc $3 \cdot$ load $\mathtt{A} + \mathtt{y} \cdot$ load $\mathtt{B} + \mathbf{read}(sm, (\mathtt{A}+\mathtt{y})) * 512 \cdot$ rollback $0 \cdot$ pc $2 \cdot$ pc 10 from Example 7. MEMLEAK detects a leak caused by the observation load $\mathtt{B} + \mathbf{read}(sm, (\mathtt{A} + \mathtt{y})) * 512$. Specifically, it detects that there are distinct symbolic valuations that agree on the non-speculative observations but disagree on the value of load $\mathtt{B} + \mathbf{read}(sm, (\mathtt{A} + \mathtt{y})) * 512$. That is, the observation depends on sensitive information that is not disclosed by τ_2's non-speculative projection. ■

Soundness and completeness: Theorem 2 states that SPECTECTOR deems secure only speculatively non-interferent programs, and all detected leaks are actual violations of SNI.

Theorem 2. *If* SPECTECTOR(p, P, w) *terminates, then* SPECTECTOR$(p, P, w) =$ SECURE *iff the program p satisfies speculative non-interference w.r.t. the policy P and all prediction oracles $\mathcal{O}$ with speculative window at most w.*

The theorem follows from the soundness and completeness of the always-mispredict semantics w.r.t. prediction oracles (Theorem 1) and of the symbolic semantics w.r.t. to the always-mispredict semantics (Proposition 2).

VII. TOOL IMPLEMENTATION

We implement our approach in our tool SPECTECTOR, which is available at `https://spectector.github.io`. The tool, which is implemented on top of the CIAO logic programming system [22], consists of three components: a front end that translates x86 assembly programs into μASM, a core engine implementing Algorithm 1, and a back end handling SMT queries.

x86 front end: The front end translates AT&T/GAS and Intel-style assembly files into μASM. It currently supports over 120 instructions: data movement instructions (**mov**, etc.), logical, arithmetic, and comparison instructions (**xor**, **add**, **cmp**, etc.), branching and jumping instructions (**jae**, **jmp**, etc.), conditional moves (**cmovae**, etc.), stack manipulation (**push**, **pop**, etc.), and function calls[5] (**call**, **ret**).

It currently does not support privileged x86 instructions, e.g., for handling model specific registers and virtual memory. Further it does not support sub-registers (like `eax`, `ah`, and `al`) and unaligned memory accesses, i.e., we assume that only 64-bit words are read/written at each address without overlaps. Finally, the translation currently maps symbolic address names to μASM instruction addresses, limiting arithmetic on code addresses.

Core engine: The core engine implements Algorithm 1. It relies on a concolic approach to implement symbolic execution that performs a depth-first exploration of the symbolic runs. Starting from a concrete initial configuration, the engine executes the program under the always-mispredict speculative semantics while keeping track of the symbolic configuration and path condition. It discovers new runs by iteratively negating the last (not previously negated) conjunct in the path condition until it finds a new initial configuration, which is then used to re-execute the program concolically. In our current implementation, indirect jumps are *not* included in the path conditions, and thus new symbolic runs and corresponding inputs are only discovered based on negated branch conditions.[6] This process is interleaved with the MEMLEAK and CTRLLEAK checks and iterates until a leak is found or all paths have been explored.

SMT back end: The Z3 SMT solver [16] acts as a back end for checking satisfiability and finding models of symbolic expressions using the BITVECTOR and ARRAY theories, which are used to model registers and memory. The implementation currently does not rely on incremental solving, since it was less efficient than one-shot solving for the selected theories.

VIII. CASE STUDY: COMPILER COUNTERMEASURES

This section reports on a case study in which we apply SPECTECTOR to analyze the security of compiler-level countermeasures against SPECTRE. We analyze a corpus of 240 assembly programs derived from the variants of the SPECTRE v1 vulnerability by Kocher [14] using different compilers and compiler options. This case study's goals are: (1) to determine whether speculative non-interference realistically captures speculative leaks, and (2) to assess SPECTECTOR's precision.

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.

[5] We model the so-called "near calls", where the callee is in the same code segment as the caller.

[6] We plan to remove this limitation in a future release of our tool.

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. 7. Analysis of Kocher's examples [14], compiled with different 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.

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.[7] (c) SLH: we compile using speculative load hardening.[8]

Compiling each of the 15 examples from [14] 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.[9] 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 7 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. See Appendix E-A for the corresponding CLANG assembly.

- 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 [14] (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 [14]), SPECTECTOR establishes the security

[7] 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.

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

[9] The resulting assembly files are available at https://spectector.github.io.

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`.

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 non-interference.

While this example nicely illustrates the scope of Spectector, it is likely not a problem in practice: First, 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 leak effectively terminates speculation, the reported leak is spurious. Second, the leak can be mitigated at the OS-level by ensuring that `0xFF...FF` is not mapped in the page tables, or that the value of `A[0xFF...FF]` does not contain any secret [23]. Such contextual information can be expressed with policies (see V-A) to improve the precision of the analysis.

C. Performance

We run all experiments on a Linux machine (kernel 4.9.0-8-amd64) with Debian 9.0, a Xeon Gold 6154 CPU, and 64 GB of RAM. We use Ciao version 1.18 and the Z3 version 4.8.4.

Spectector terminates within less than 30 seconds on all examples, with several examples being analyzed in about 0.1 seconds, except for Example #5 in mode Slh `-O2`. In this exceptional case, Spectector needs 2 minutes for proving security. This is due to Example #5's complex control-flow, which leads to loops involving several branch instructions.

IX. Case study: Xen Project Hypervisor

This section reports on a case study in which we apply Spectector on the Xen Project hypervisor [24]. This case study's goal is to understand the challenges in scaling the tool to a significant real-world code base. It forms a snapshot of our ongoing effort towards the comprehensive side-channel analysis of the Xen hypervisor.

A. Challenges for scaling-up

There are three main challenges for scaling Spectector to a large code base such as the Xen hypervisor:

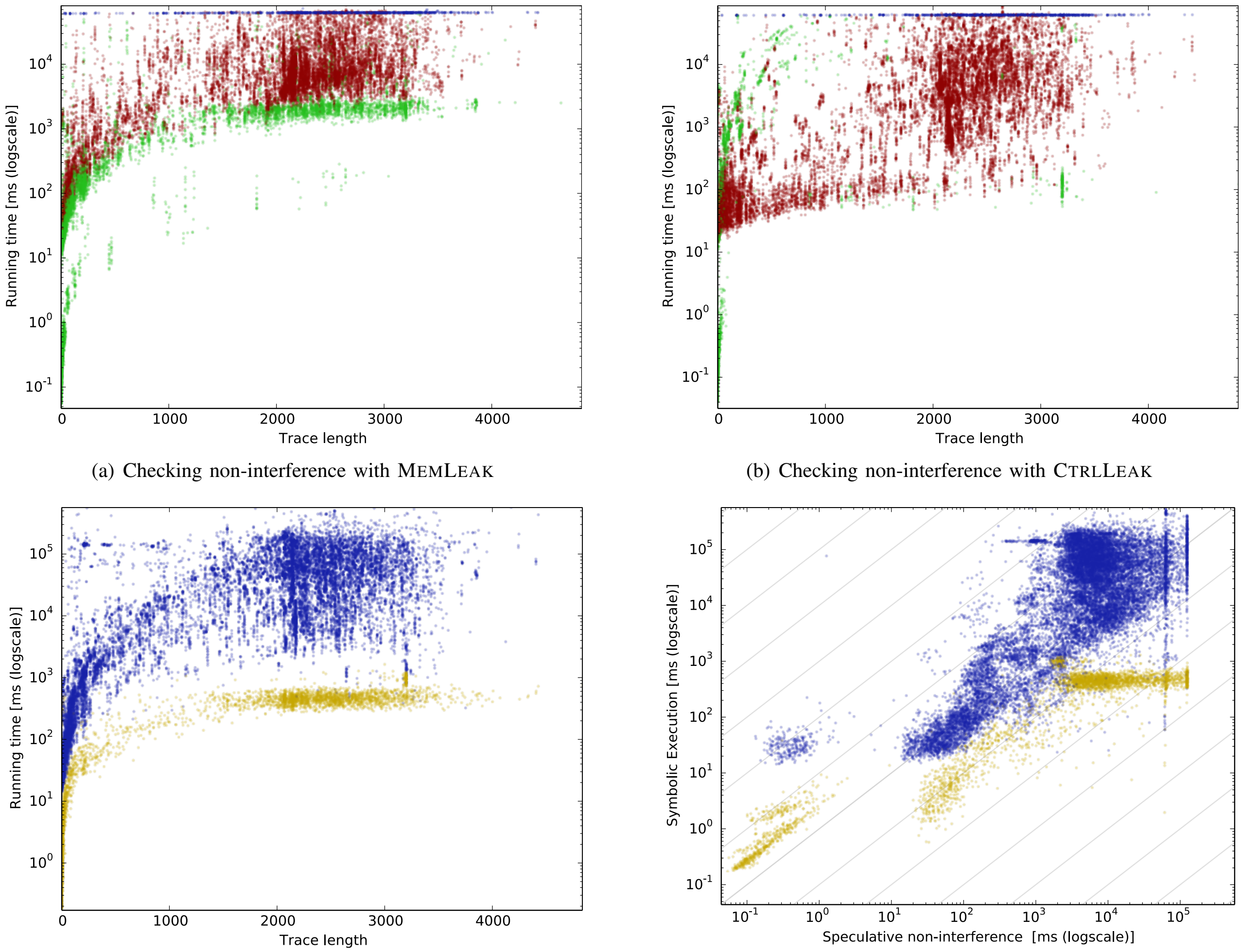
ISA support: Our front end currently supports only a fraction of the x64 ISA (cf. Section VII). Supporting the full x64 ISA is conceptually straightforward but out of the scope of this paper. For this case study, we treat unsupported instructions as **skip**, sacrificing the analysis's correctness.

Policies: Spectector uses policies specifying the public and secret parts of configurations. The manual specification of precise policies (as in Section VIII) is infeasible for large code bases, and their automatic inference from the calling context is not yet supported by Spectector. For this case study, we use a policy that treats registers as "low" and memory locations as "high", which may introduce false alarms. For instance, the policy treats as "high" all function parameters that are retrieved from memory (e.g., popped from the stack), which is why Spectector flags their speculative uses in memory or branching instructions as leaks.

Path explosion and nontermination: Spectector is based on symbolic execution, which suffers from path explosion and nontermination when run on programs with loops and indirect jumps. In the future, we plan to address this challenge by employing approximate but sound static analysis techniques, such as abstract interpretation. Such techniques can be employed both to efficiently infer loop invariants, and jump targets, but also to directly address the question whether a given program satisfies SNI or not. A systematic study of techniques to soundly approximate SNI is out of scope of this paper. For this case study, as discussed in the following section, we bound the number and the lengths of symbolic paths that are explored, thereby sacrificing the soundness of our analysis.

B. Evaluating scalability

Approach: To perform a meaningful evaluation of Spectector's scalability despite the incomplete path coverage, we compare the time spent on discovering new symbolic paths with the time spent on checking SNI. Analyzing paths of different lengths enables us to evaluate the scalability of checking SNI *relative to that* of symbolic execution, which factors out the path explosion problem from the analysis.



(a) Checking non-interference with MEMLEAK

(b) Checking non-interference with CTRLLEAK

(c) Discovering symbolic paths

(d) Symbolic execution versus SNI check

Fig. 8. Scalability analysis for the Xen Project hypervisor. In (a) and (b), green denotes secure traces, red denotes insecure traces, and blue denotes traces producing timeouts. In (c) and (d), yellow denotes the first trace discovered for each function, while blue denotes all discovered further traces. The vertical lines in (d) represent traces where either MEMLEAK times out and CTRLLEAK succeed or both time out.

We stress that we sacrifice soundness and completeness of the analysis for running SPECTECTOR on the full Xen codebase (see Section IX-A). This is why in this section we do *not* attempt to make statements about the security of the hypervisor.

Setup: We analyze the Xen Project hypervisor version 4.10, which we compile using CLANG v7.0.0. We identify 3 959 functions in the generated assembly. For each function, we explore at most 25 symbolic paths of at most 10 000 instructions each, with a global timeout of 10 minutes.[10]

We record execution times as a function of the trace length, i.e., the number of `load` se, `store` se, and `symPc`(se) observations, rather than path length, since the former is more relevant for the size of the resulting SMT formulas. We execute our experiments on the machine described in Section VIII-C.

[10]The sources and scripts needed for reproducing our results are available at `https://spectector.github.io`.

C. Experimental results

Cost of symbolic execution: We measure the time taken for discovering symbolic paths (cf. Section VII). In total, SPECTECTOR discovers 24 701 symbolic paths. Figure 8(c) depicts the time for discovering paths. We highlight the following findings:

- As we apply concolic execution, discovering the first symbolic path does not require any SMT queries and is hence cheap. These cases are depicted by yellow dots in Fig. 8(c).
- Discovering further paths requires SMT queries. This increases execution time by approximately two orders of magnitude. These cases correspond to the blue dots in Fig. 8(c).
- For 48.3% of the functions we do not reach the limit of 25 paths, for 35.4% we do not reach the limit of 10 000 instructions per path, and for 18.7% we do not encounter unsupported instructions. 13 functions satisfy all three conditions.

Cost of checking SNI: We apply MEMLEAK and CTRLLEAK to the 24 701 traces (derived from the discovered paths), with a timeout of 1 minute each. Figure 8(a) and 8(b) depict the respective analysis runtimes; Figure 8(d) relates the time required for discovering a new trace with the time for checking SNI, i.e., for executing lines 3–4 in Algorithm 1.

We highlight the following findings:

- MEMLEAK and CTRLLEAK can analyze 93.8% and 94.7%, respectively, of the 24 701 traces in less than 1 minute. The remaining traces result in timeouts.
- For 41.9% of the traces, checking SNI is at most 10x faster than discovering the trace, and for 20.2% of the traces it is between 10x and 100x faster. On the other hand, for 26.9% of the traces, discovering the trace is at most 10x faster than checking SNI, and for 7.9% of the traces, discovering the trace is between 10x and 100x faster than checking SNI.

Summary: Overall, our data indicates that the cost of checking SNI is comparable to that of discovering symbolic paths. This may be surprising since SNI is a relational property, which requires *comparing* executions and is know to scale poorly. However, note that SPECTECTOR only compares executions that follow the *same symbolic path*. This is sufficient because the program counter is observable, i.e., speculative non-interference never requires to consider two executions that disagree on path conditions. We hence conclude that our approach does not exhibit fundamental bottlenecks beyond those it inherits from symbolic execution.

X. DISCUSSION

A. Exploitability

Exploiting speculative execution attacks requires an adversary to (1) prepare the microarchitectural state, (2) run victim code—partially speculatively—to encode information into the microarchitectural state, and (3) extract the leaked information from the microarchitectural state. SPECTECTOR analyzes the victim code to determine whether it may speculatively leak information into the microarchitectural state in any possible attack context. Following the terminology of [25], [26], speculative non-interference is a semantic characterization of *disclosure gadgets* enabled by speculative execution.

B. Scope of the model

The results obtained by SPECTECTOR are only valid to the extent that the speculative semantics and the observer model accurately capture the target system.

In particular, SPECTECTOR may incorrectly classify a program as secure if the speculative semantics does not implicitly[11] capture all additional observations an adversary may make due to speculative execution on an actual microarchitecture. For example, microarchitectures could potentially speculate on the condition of a conditional update, which our speculative semantics currently does not permit.

Similarly, SPECTECTOR may incorrectly classify a program as insecure if the speculative semantics admits speculative executions that are not actually possible on an actual microarchitecture. This might be the case for speculative load hardening on Kocher's Example #10, as discussed in Section VIII.

[11] *Implicitly*, because we take the memory accesses performed by the program and the flow of control as a proxy for the observations an adversary might make, e.g., through the cache.

The speculative semantics, however, can always be adapted to more accurately reflect reality, once better documentation of processor behavior becomes available. In particular, it would be relatively straightforward to extend the speculative semantics with models of indirect jump predictors [2], return stack buffers [1], and memory disambiguation predictors [27]. The notion of SNI itself is robust to such changes, as it is defined relative to the speculative semantics.

We capture "leakage into the microarchitectural state" using the relatively powerful observer of the program execution that sees the location of memory accesses and the jump targets. This observer could be replaced by a weaker one, which accounts for more detailed models of a CPU's memory hierarchy, and SPECTECTOR could be adapted accordingly, e.g. by adopting the cache models from CacheAudit [28]. We believe, however, that highly detailed models are not actually desirable for several reasons: (a) they encourage brittle designs that break under small changes to the model, (b) they have to be adapted frequently, and (c) they are hard to understand and reason about for compiler developers and hardware engineers. The "constant-time" observer model adopted in this paper has proven to offer a good tradeoff between precision and robustness [12], [13].

XI. RELATED WORK

Speculative execution attacks: These attacks exploit speculatively executed instructions to leak information. After SPECTRE [2], [5], many speculative execution attacks have been discovered that differ in the exploited speculation sources [1], [4], [27], the covert channels [3], [29], [30] used, or the target platforms [31]. We refer the reader to [26] for a survey of speculative execution attacks and their countermeasures.

Here, we overview only SPECTRE v1 software-level countermeasures. AMD and Intel suggest inserting **lfence** instructions after branches [6], [32]. These instructions effectively act as speculation barriers, and prevent speculative leaks. The Intel C++ compiler [9], the Microsoft Visual C++ compiler [10], and the CLANG [8] compiler can automatically inject this countermeasure at compile time. Taram et al. [33] propose *context-sensitive fencing*, a defense mechanism that dynamically injects fences at the microoperation level where necessary, as determined by a dynamic information-flow tracker. An alternative technique to injecting fences is to introduce artificial data dependencies [8], [34]. Speculative Load Hardening (SLH) [8], implemented in the CLANG compiler, employs carefully injected data dependencies and masking operations to prevent the leak of sensitive information into the microarchitectural state. A third software-level countermeasure consists in replacing branching instructions by other computations, like bit masking, that do not trigger speculative execution [7].

Detecting speculative leaks: oo7 [11] is a binary analysis tool for detecting speculative leaks. The tool looks for specific syntactic code patterns and it can analyze large code bases. However, it misses some speculative leaks, like Example #4 from Section VIII. oo7 would also incorrectly classify all the programs patched by SLH in our case studies as insecure, since they still match oo7's vulnerable patterns. In contrast, SPECTECTOR builds on a semantic notion of security and is thus not limited to particular syntactic code patterns.

Disselkoen et al. [35] and Mcilroy et al. [36] develop models for capturing speculative execution, which they use to illustrate several known Spectre variants. Neither approach provides a security notion or a detection technique. Compared with our speculative semantics, the model of [36] more closely resembles microarchitectural implementations by explicitly modeling the reorder buffer, caches, and branch predictors, which we intentionally abstract away.

In work concurrent to ours, Cheang et al. [37] introduce the notion of trace property-dependent observational determinism (TPOD), which they instantiate to formally capture the new leaks introduced by the interaction of microarchitectural side channels with speculative execution. As TPOD is a 4-safety property it can be checked using 4-way self composition. In contrast, SNI can be checked by 2-way self composition (cf. Proposition 1), which is likely to be more efficient.

Formal architecture models: Armstrong et al. [38] present formal models for the ARMv8-A, RISC-V, MIPS, and CHERI-MIPS instruction-set architectures. Degenbaev [39] and Goel et al. [40] develop formal models for parts of the x86 architecture. Such models enable, for instance, the formal verification of compilers, operating systems, and hypervisors. However, ISA models naturally abstract from microarchitectural aspects such as speculative execution or caches, which are required to reason about side-channel vulnerabilities.

Zhang et al. [41] present Coppelia, a tool to automatically generate software exploits for hardware designs. However, the processor designs they consider, OR1200, PULPino, and Mor1kx, do not feature speculative execution.

Static detection of side-channel vulnerabilities: Several approaches have been proposed for statically detecting side-channel vulnerabilities in programs [13], [28], [42], [43]. These differ from our work in that (1) they do not consider speculative execution, and (2) we exclusively target speculation leaks, i.e., we ignore leaks from the standard semantics. However, we note that our tool could easily be adapted to also detect leaks from the standard semantics.

XII. CONCLUSIONS

We introduce speculative non-interference, the first semantic notion of security against speculative execution attacks. Based on this notion we develop SPECTECTOR, a tool for automatically detecting speculative leaks or proving their absence, and we show how it can be used to detect subtle leaks—and optimization opportunities—in the way state-of-the-art compilers apply SPECTRE mitigations.

Acknowledgments: We thank Roberto Giacobazzi, Catalin Marinas, Matt Miller, Matthew Parkinson, Niki Vazou, and the anonymous reviewers for helpful discussions and comments. This work was supported by a grant from Intel Corporation, Ramón y Cajal grant RYC-2014-16766, Atracción de Talento Investigador grant 2018-T2/TIC-11732A, Spanish projects TIN2015-70713-R DEDETIS, TIN2015-67522-C3-1-R TRACES, and RTI2018-102043-B-I00 SCUM, and Madrid regional projects S2013/ICE-2731 N-GREENS and S2018/TCS-4339 BLOQUES.

REFERENCES

[1] G. Maisuradze and C. Rossow, "Ret2Spec: Speculative Execution Using Return Stack Buffers," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018.

[2] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.

[3] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," *CoRR*, vol. abs/1807.10535, 2018.

[4] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies*, ser. WOOT '18. USENIX Association, 2018.

[5] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *CoRR*, vol. abs/1807.03757, 2018.

[6] 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.

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

[8] C. Carruth, "Speculative load hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 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] A. Pardoe, "Spectre mitigations in MSVC," https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[11] 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.

[12] 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 *Proceedings of the 8th International Conference on Information Security and Cryptology*, ser. ICISC '05. Springer, 2005.

[13] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 25th USENIX Security Symposium*, ser. USENIX Security '16. USENIX Association, 2016.

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

[15] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," *CoRR*, vol. abs/1812.08639, 2018.

[16] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '08. Springer, 2008.

[17] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[18] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, 2009.

[19] I. Anati, D. Blythe, J. Doweck, H. Jiang, W. Kao, J. Mandelblat, L. Rappoport, E. Rotem, and A. Yasin, "Inside 6th gen intel core: New microarchitecture code named Skylake," in *Proceedings of the 28th IEEE Hot Chips Symposium*, ser. HCS '16, 2016.

[20] A. R. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*. Springer, 2007.

[21] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, ser. CSFW '04. IEEE, 2004.

[22] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *Theory and Practice of Logic Programming*, vol. 12, no. 1–2, 2012.

[23] C. Marinas, Personal communication.

[24] "Xen Project," https://xenproject.org.

[25] M. 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.

[26] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *Proceedings of the 28th USENIX Security Symposium*, ser. USENIX Security '19. USENIX Association, 2019.

[27] J. Horn, "CVE-2018-3639 - speculative store bypass," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639, 2018.

[28] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, 2015.

[29] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, vol. abs/1802.03802, 2018.

[30] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *CoRR*, vol. abs/1806.07480, 2018.

[31] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Stealing intel secrets from SGX enclaves via speculative execution," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy*, ser. EuroS&P '19. IEEE, 2019.

[32] Advanced Micro Devices, Inc., "Software techniques for managing speculation on amd processors," https://developer.amd.com/wp-content/resources/90343-B_SotwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[33] M. Taram, A. Venkat, and D. M. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019.

[34] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You shall not bypass: Employing data dependencies to prevent bounds check bypass," *CoRR*, vol. abs/1805.08506, 2018.

[35] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely, "Code that never ran: modeling attacks on speculative evaluation," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.

[36] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019.

[37] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *Proceedings of the 32nd IEEE Computer Security Foundations Symposium*, ser. CSF '19, 2019.

[38] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019.

[39] U. Degenbaev, "Formal specification of the x86 instruction set architecture," Ph.D. dissertation, Universität des Saarlandes, 2012.

[40] S. Goel, W. A. Hunt, and M. Kaufmann, *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. Springer, 2017.

[41] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '18. IEEE/ACM, 2018.

[42] G. Doychev and B. Köpf, "Rigorous Analysis of Software Countermeasures against Cache Attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '17. ACM, 2017.

[43] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.

APPENDIX A: NON-SPECULATIVE SEMANTICS

Given a program p, we formalize its non-speculative semantics using the relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Obs} \times \mathit{Conf}$ in Figure 9.

APPENDIX B: TRACE PROJECTIONS

Here, we formalize the speculative projection $\tau\restriction_{se}$ and the non-speculative projection $\tau\restriction_{nse}$.

Non-speculative projection: Given a trace τ, its non-speculative projection contains only the observations that are produced by committed transactions; in other words, rolled-back transactions are removed in the projection. Formally, $\tau\restriction_{nse}$ is defined as follows: $\varepsilon\restriction_{nse} = \varepsilon$, $(o \cdot \tau)\restriction_{nse} = o \cdot \tau\restriction_{nse}$ if o is $\texttt{load}\ se$, $\texttt{store}\ se$, $\texttt{pc}\ n$, or $\texttt{symPc}(se)$, $(\texttt{start}\ i \cdot \tau)\restriction_{nse} = \tau\restriction_{nse}$ if $\texttt{rollback}\ i$ is not in τ, $(\texttt{commit}\ i \cdot \tau)\restriction_{nse} = \tau\restriction_{nse}$, $(\texttt{start}\ i \cdot \tau \cdot \texttt{rollback}\ i \cdot \tau')\restriction_{nse} = \tau'\restriction_{nse}$, and $\tau\restriction_{nse} = \varepsilon$ otherwise.

Speculative projection: Given a speculative trace τ, its speculative projection contains only the observations produced by rolled-back transactions. Formally, $\tau\restriction_{se}$ is defined as: $\varepsilon\restriction_{se} = \varepsilon$, $(o \cdot \tau)\restriction_{se} = \tau\restriction_{se}$ if o is $\texttt{load}\ se$, $\texttt{store}\ se$, $\texttt{pc}\ n$, or $\texttt{symPc}(se)$, $(\texttt{start}\ i \cdot \tau)\restriction_{se} = \tau\restriction_{se}$ if $\texttt{rollback}\ i$ is not in τ, $(\texttt{commit}\ i \cdot \tau)\restriction_{se} = \tau\restriction_{se}$, $(\texttt{start}\ i \cdot \tau \cdot \texttt{rollback}\ i \cdot \tau')\restriction_{se} = \mathit{filter}(\tau) \cdot \tau'\restriction_{nse}$, and $\tau\restriction_{nse} = \varepsilon$ otherwise, where $\mathit{filter}(\tau)$ denotes the trace obtained by dropping all extended observations $\texttt{start}\ id$, $\texttt{commit}\ id$, and $\texttt{rollback}\ id$ from τ.

APPENDIX C: ALWAYS-MISPREDICT SEMANTICS

We describe the execution of μASM programs under the always-mispredict oracle with speculative window w as a ternary evaluation relation $\langle ctr, \sigma, s\rangle \overset{\tau}{\Longrightarrow} \langle ctr', \sigma', s'\rangle$ mapping a configuration $\langle ctr, \sigma, s\rangle$ to a configuration $\langle ctr', \sigma', s'\rangle$ while producing the observations τ. Differently from the speculative semantics, the always-mispredict semantics does not require a branching history h, since its prediction only depends on the branch outcome. The rules formalizing the always-mispredict semantics are given in Figure 10.

AM-NOBRANCH captures the behavior of non-branching instructions. Similar to its counterpart SE-NOBRANCH, the rule acts as a wrapper for the standard semantics. The difference lies in the the auxiliary predicate $\mathit{enabled}'(s)$ and the auxiliary functions $\mathit{decr}'(s)$, and $\mathit{zeroes}'(s)$, which apply their non-primed counterpart only to the *last* transaction in the speculative state. E.g., $\mathit{enabled}'(s \cdot \langle id, w, \ell, \sigma\rangle) = \mathit{enabled}(\langle id, w, \ell, \sigma\rangle)$. This ensures that upon rolling back a nested transaction, its enclosing transaction can explore the other alternative branch to the full depth of the speculative window (corresponding to the case of a correct prediction).

AM-BRANCH models the behavior of branching instructions $\mathbf{beqz}\ x, \ell'$. The rule mispredicts the outcome of the branch instruction by setting the program counter to ℓ' only when the condition is *not* satisfied. The length of the new transaction is set to the minimum of the oracle's speculative window w and $\mathit{wndw}(s) - 1$, where $\mathit{wndw}(s)$ is the remaining length of the last speculative transaction in s. This ensures that nested transactions are not explored for longer than permitted by

Expression evaluation

$$[\![n]\!](a) = n \qquad [\![x]\!](a) = a(x) \qquad [\![\ominus e]\!](a) = \ominus [\![e]\!](a) \qquad [\![e_1 \otimes e_2]\!](a) = [\![e_1]\!](a) \otimes [\![e_2]\!](a)$$

Instruction evaluation

$$\textsc{Skip}\ \frac{p(a(\mathbf{pc})) = \mathbf{skip}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle} \qquad \textsc{Barrier}\ \frac{p(a(\mathbf{pc})) = \mathbf{spbarr}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

$$\textsc{Assign}\ \frac{p(a(\mathbf{pc})) = x \leftarrow e \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto [\![e]\!](a)]\rangle}$$

$$\textsc{ConditionalUpdate-Sat}\ \frac{p(a(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](a) = 0 \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto [\![e]\!](a)]\rangle}$$

$$\textsc{ConditionalUpdate-Unsat}\ \frac{p(a(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](a) \neq 0 \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle} \qquad \textsc{Terminate}\ \frac{p(a(\mathbf{pc})) = \bot}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto \bot]\rangle}$$

$$\textsc{Load}\ \frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad n = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{load}\ n} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle}$$

$$\textsc{Store}\ \frac{p(a(\mathbf{pc})) = \mathbf{store}\ x, e \qquad n = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{store}\ n} \langle m[n \mapsto a(x)], a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

$$\textsc{Beqz-Sat}\ \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad a(x) = 0}{\langle m, a\rangle \xrightarrow{\texttt{pc}\ \ell} \langle m, a[\mathbf{pc} \mapsto \ell]\rangle} \qquad \textsc{Beqz-Unsat}\ \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad a(x) \neq 0}{\langle m, a\rangle \xrightarrow{\texttt{pc}\ a(\mathbf{pc})+1} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

$$\textsc{Jmp}\ \frac{p(a(\mathbf{pc})) = \mathbf{jmp}\ e \qquad \ell = [\![e]\!](a)}{\langle m, a\rangle \xrightarrow{\texttt{pc}\ \ell} \langle m, a[\mathbf{pc} \mapsto \ell]\rangle}$$

Fig. 9. μASM semantics for a program p

their enclosing transactions, whose remaining lengths are not decremented during the execution of the nested transaction.

AM-ROLLBACK models the rollback of speculative transactions. Different from SE-ROLLBACK, and by design of AM-NOBRANCH, the rule applies only to the last transaction in s. Since the semantics always-mispredicts the outcome of branch instructions, SE-ROLLBACK is always applied, i.e there is no need for a rule that handles committed transactions.

Similarly to Proposition 1, a program's non-speculative behavior can be recovered from the always-mispredict semantics.

Proposition 3. *Let p be a program and w be a speculative window. Then, $(\!|p|\!) = \{\!|p|\!\}_w\lceil_{nse}$.*

Proposition 4 states that the always-mispredict semantics yields the worst-case leakage.

Proposition 4. *Let p be a program, $w \in \mathbb{N}$ be a speculative window, and $\sigma, \sigma' \in$ InitConf be initial configurations. $\{\!|p|\!\}_w(\sigma) = \{\!|p|\!\}_w(\sigma')$ iff $[\![p]\!]_{\mathcal{O}}(\sigma) = [\![p]\!]_{\mathcal{O}}(\sigma')$ for all prediction oracles $\mathcal{O}$ with speculative window at most w.*

APPENDIX D: SYMBOLIC SEMANTICS

Here, we formalize the symbolic semantics.

Symbolic expressions: Symbolic expressions represent computations over symbolic values. A *symbolic expression* se is a concrete value $n \in \mathit{Vals}$, a symbolic value $s \in \mathit{SymbVals}$, an if-then-else expression $\mathbf{ite}(se, se', se'')$, or the application of a unary $\ominus$ or a binary operator $\otimes$.

$$se := n \mid s \mid \mathbf{ite}(se, se', se'') \mid \ominus se \mid se \otimes se'$$

Symbolic memories: We model symbolic memories as symbolic arrays using the standard theory of arrays [20]. That is, we model memory updates as triples of the form $\mathbf{write}(sm, se, se')$, which updates the symbolic memory sm by assigning the symbolic value se' to the symbolic location se, and memory reads as $\mathbf{read}(sm, se)$, which denote retrieving the value assigned to the symbolic expression se.

A *symbolic memory* sm is either a function $\mathit{mem} : \mathbb{N} \to \mathit{SymbVals}$ mapping memory addresses to symbolic values or a term $\mathbf{write}(sm, se, se')$, where sm is a symbolic memory and se, se' are symbolic expressions. To account for symbolic memories, we extend symbolic expressions with terms of the form $\mathbf{read}(sm, se)$, where sm is a symbolic memory and se is a symbolic expression, representing memory reads.

$$\begin{aligned} sm &:= \mathit{mem} \mid \mathbf{write}(sm, se, se') \\ se &:= \ldots \mid \mathbf{read}(sm, se) \end{aligned}$$

Evaluating symbolic expressions: The value of a symbolic expression se depends on a *valuation* $\mu : \mathit{SymbVals} \to \mathit{Vals}$ mapping symbolic values to concrete ones:

$$\begin{aligned} \mu(n) &= n \text{ if } n \in \mathit{Vals} \\ \mu(s) &= \mu(s) \text{ if } s \in \mathit{SymbVals} \\ \mu(\mathbf{ite}(se, se', se'')) &= \mu(se') \text{ if } \mu(se) \neq 0 \\ \mu(\mathbf{ite}(se, se', se'')) &= \mu(se'') \text{ if } \mu(se) = 0 \\ \mu(\ominus se) &= \ominus \mu(se) \\ \mu(se \otimes se') &= \mu(se) \otimes \mu(se') \\ \mu(\mathit{mem}) &= \mu \circ \mathit{mem} \\ \mu(\mathbf{write}(sm, se, se')) &= \mu(sm)[\mu(se) \mapsto \mu(se')] \\ \mu(\mathbf{read}(sm, se)) &= \mu(sm)(\mu(se)) \end{aligned}$$

An expression se is *satisfiable* if there is a valuation μ satisfying it, i.e., $\mu(se) \neq 0$.

Symbolic assignments: A *symbolic assignment* sa is a function mapping registers to symbolic expressions $sa : \mathit{Regs} \to \mathit{SymbExprs}$. Given a symbolic assignment sa and a valuation μ, $\mu(sa)$ denotes the assignment $\mu \circ sa$. We assume the program counter $\mathbf{pc}$ to always be concrete, i.e., $sa(\mathbf{pc}) \in \mathit{Vals}$.

AM-NoBranch

$$\frac{p(\sigma(\mathbf{pc})) \neq \mathbf{beqz}\ x,\ell \qquad \sigma \xrightarrow{\tau} \sigma' \qquad enabled'(s) \qquad s' = \begin{cases} decr'(s) & \text{if } p(\sigma(\mathbf{pc})) \neq \mathbf{spbarr} \\ zeroes'(s) & \text{otherwise} \end{cases}}{\langle ctr, \sigma, s\rangle \xRightarrow{\tau} \langle ctr, \sigma', s'\rangle}$$

AM-Branch

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x,\ell' \qquad enabled'(s) \qquad \ell = \begin{cases} \sigma(\mathbf{pc})+1 & \text{if } \sigma(x) = 0 \\ \ell' & \text{if } \sigma(x) \neq 0 \end{cases} \qquad id = ctr \qquad s' = decr'(s) \cdot \langle \sigma, ctr, min(w, wndw(s)-1), \ell\rangle}{\langle ctr, \sigma, s\rangle \xRightarrow{\mathtt{start}\ id \cdot pc\ \ell} \langle ctr+1, \sigma[\mathbf{pc} \mapsto \ell], s'\rangle}$$

AM-Rollback

$$\frac{\sigma' \xrightarrow{\tau} \sigma''}{\langle ctr, \sigma, s \cdot \langle \sigma', id, 0, \ell\rangle\rangle \xRightarrow{\mathtt{rollback}\ id \cdot pc\ \sigma''(\mathbf{pc})} \langle ctr, \sigma'', s\rangle}$$

Fig. 10. Always-mispredict speculative semantics for a program p and speculative window w

Symbolic configurations: A *symbolic configuration* is a pair $\langle sm, sa\rangle$ consisting of a symbolic memory sm and a symbolic assignment sa. We lift speculative states to symbolic configurations. A *symbolic extended configuration* is a triple $\langle ctr, \sigma, s\rangle$ where $ctr \in \mathbb{N}$ is a counter, $\sigma \in Conf$ is a symbolic configuration, and s is a symbolic speculative state.

Symbolic observations: When symbolically executing a program, we may produce observations whose value is symbolic. To account for this, we introduce symbolic observations of the form `load` se and `store` se, which are produced when symbolically executing **load** and **store** commands, and `symPc`(se), produced when symbolically evaluating branching instructions, where se is a symbolic expression. In our symbolic semantics, we use the observations `symPc`(se) to represent the symbolic path condition indicating when a path is feasible. Given a sequence of symbolic observations τ and a valuation μ, $\mu(\tau)$ denotes the trace obtained by (1) dropping all observations `symPc`(se), and (2) evaluating all symbolic observations different from `symPc`(se) under μ.

Symbolic semantics: The *non-speculative* semantics is captured by the relation $\rightarrow_s$ in Fig. 11, while the *speculative* semantics is captured by the relation $\Longrightarrow_s$ in Fig. 12.

Computing symbolic runs and traces: We now fix the symbolic values. The set $SymbVals$ consists of a symbolic value x_s for each register identifier x and of a symbolic value mem_s^n for each memory address n. We also fix the initial symbolic memory $sm_0 = \lambda n \in \mathbb{N}.\ m_s^n$ and the symbolic assignment sa_0 such that $sa_0(\mathbf{pc}) = 0$ and $sa_0(x) = x_s$.

The set $\{\!|p|\!\}_w^{symb}$ contains all runs that can be derived using the symbolic semantics (with speculative window w) starting from the initial configuration $\langle sm_0, sa_0\rangle$. That is, $\{\!|p|\!\}_w^{symb}$ contains all triples $\langle\langle sm_0, sa_0\rangle, \tau, \sigma'\rangle$, where τ is a symbolic trace and σ' is a final symbolic configuration, corresponding to symbolic computations $\langle 0, \langle sm_0, sa_0\rangle, \varepsilon\rangle \xRightarrow{\tau}_s^* \langle ctr, \sigma', \varepsilon\rangle$ where the path condition $\bigwedge_{\mathtt{symPc}(se) \in \tau} se$ is satisfiable.

We compute $\{\!|p|\!\}_w^{symb}$ in the standard way. We keep track of a path constraint PC and we update it whenever the semantics produces an observation `symPc`(se). We start the computation from $\langle 0, \langle sm_0, sa_0\rangle, \varepsilon\rangle$ and $PC = \top$. When executing branch and jump instructions, we explore all branches consistent with the current PC, and, for each of them, we update PC.

Appendix E: Code from Case Studies

A. Example #8

In Example #8, the bounds check of Figure 1 is implemented using a conditional operator:

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

When compiling the example without countermeasures or optimizations, the conditional operator is translated to a branch instruction (cf. line 4), which is a source of speculation. Hence, the resulting program contains a speculative leak, which Spectector correctly detects.

```
1      mov    size, %rcx
2      mov    y, %rax
3      cmp    %rcx, %rax
4      jae    .L1
5      add    $1, %rax
6      jmp    .L2
7  .L1:
8      xor    %rax, %rax
9      jmp    .L2
10 .L2:
11     mov    A(%rax), %rax
12     shl    $9, %rax
13     mov    B(%rax), %rax
14     mov    temp, %rcx
15     and    %rax, %rcx
16     mov    %rcx, temp
```

In the Unp -O2 mode, the conditional operator is translated as a conditional move (cf. line 6), for which Spectector can prove security.

```
1 mov    size, %rax
2 mov    y, %rdx
3 xor    %rcx, %rcx
4 cmp    %rdx, %rax
5 lea    1(%rdx), %rax
6 cmova  %rax, %rcx
7 mov    A(%rcx), %rax
8 shl    $9, %rax
9 mov    B(%rax), %rax
```

Expression evaluation

$$
\begin{array}{ll}
[\![n]\!](a) = n & \text{if } n \in \mathit{Vals} \\
[\![se]\!](a) = se & \text{if } sa \in \mathit{SymbExprs} \setminus \mathit{Vals} \\
[\![x]\!](a) = a(x) & \text{if } x \in \mathit{Regs} \\
[\![\ominus e]\!](a) = \mathit{apply}(\ominus, [\![e]\!](a)) & \text{if } [\![e]\!](a) \in \mathit{Vals} \\
[\![\ominus e]\!](a) = \ominus [\![e]\!](a) & \text{if } [\![e]\!](a) \in \mathit{SymbExprs} \setminus \mathit{Vals} \\
[\![e_1 \otimes e_2]\!](a) = \mathit{apply}(\otimes, [\![e_1]\!](a), [\![e_2]\!](a)) & \text{if } [\![e_1]\!](a), [\![e_2]\!](a) \in \mathit{Vals} \\
[\![e_1 \otimes e_2]\!](a) = [\![e_1]\!](a) \otimes [\![e_2]\!](a) & \text{if } [\![e_1]\!](a) \in \mathit{SymbExprs} \setminus \mathit{Vals} \\
[\![e_1 \otimes e_2]\!](a) = [\![e_1]\!](a) \otimes [\![e_2]\!](a) & \text{if } [\![e_2]\!](a) \in \mathit{SymbExprs} \setminus \mathit{Vals}
\end{array}
$$

Instruction evaluation

SKIP
$$\frac{p(sa(\mathbf{pc})) = \mathbf{skip}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

BARRIER
$$\frac{p(sa(\mathbf{pc})) = \mathbf{spbarr}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

ASSIGN
$$\frac{p(sa(\mathbf{pc})) = x \leftarrow e \qquad x \neq \mathbf{pc}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1, x \mapsto [\![e]\!](sa)]\rangle}$$

CONDITIONALUPDATE-CONCR-SAT
$$\frac{p(sa(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](sa) = 0 \qquad x \neq \mathbf{pc}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1, x \mapsto [\![e]\!](sa)]\rangle}$$

CONDITIONALUPDATE-CONCR-UNSAT
$$\frac{\begin{array}{c} p(sa(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](sa) = n \qquad n \in \mathit{Vals} \\ n \neq 0 \qquad x \neq \mathbf{pc} \end{array}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

CONDITIONALUPDATE-SYMB
$$\frac{p(sa(\mathbf{pc})) = x \xleftarrow{e'?} e \qquad [\![e']\!](sa) = se \qquad se \notin \mathit{Vals} \qquad x \neq \mathbf{pc}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1, x \mapsto \mathbf{ite}(se = 0, [\![e]\!](sa), sa(x))]\rangle}$$

LOAD-SYMB
$$\frac{\begin{array}{c} p(sa(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad se = [\![e]\!](sa) \\ se' = \mathbf{read}(sm, se) \end{array}}{\langle sm, sa\rangle \xrightarrow{\texttt{load}\ se}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1, x \mapsto se']\rangle}$$

STORE-SYMB
$$\frac{\begin{array}{c} p(sa(\mathbf{pc})) = \mathbf{store}\ x, e \qquad se = [\![e]\!](sa) \\ sm' = \mathbf{write}(sm, se, sa(x)) \end{array}}{\langle sm, sa\rangle \xrightarrow{\texttt{store}\ se}_s \langle sm', sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

BEQZ-CONCR-SAT
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) = 0 \qquad sa(x) \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}(\top)\cdot\texttt{pc}\ \ell}_s \langle sm, sa[\mathbf{pc} \mapsto \ell]\rangle}$$

BEQZ-SYMB-SAT
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \notin \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}(sa(x)=0)\cdot\texttt{pc}\ \ell}_s \langle sm, sa[\mathbf{pc} \mapsto \ell]\rangle}$$

BEQZ-CONCR-UNSAT
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \neq 0 \qquad sa(x) \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}(\top)\cdot\texttt{pc}\ sa(\mathbf{pc})+1}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

BEQZ-SYMB-UNSAT
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \notin \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}(sa(x)\neq 0)\cdot\texttt{pc}\ sa(\mathbf{pc})+1}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

JMP-CONCR
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad \ell = [\![e]\!](sa) \qquad \ell \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}(\top)\cdot\texttt{pc}\ \ell}_s \langle sm, sa[\mathbf{pc} \mapsto \ell]\rangle}$$

JMP-SYMB
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad [\![e]\!](sa) \notin \mathit{Vals} \qquad \ell \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\texttt{symPc}([\![e]\!](sa)=\ell)\cdot\texttt{pc}\ \ell}_s \langle sm, sa[\mathbf{pc} \mapsto \ell]\rangle}$$

TERMINATE
$$\frac{p(sa(\mathbf{pc})) = \bot}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto \bot]\rangle}$$

Fig. 11. μASM symbolic non-speculative semantics for a program p

$$\textsc{Am-NoBranch}\quad \dfrac{p(\sigma(\mathbf{pc})) \neq \mathbf{beqz}\ x, \ell \qquad \sigma \xrightarrow{\tau}_s \sigma' \qquad \mathit{enabled}'(s) \qquad s' = \begin{cases} \mathit{decr}'(s) & \text{if } p(\sigma(\mathbf{pc})) \neq \mathbf{spbarr} \\ \mathit{zeroes}'(s) & \text{otherwise} \end{cases}}{\langle \mathit{ctr}, \sigma, s\rangle \xRightarrow{\tau}_s \langle \mathit{ctr}, \sigma', s'\rangle}$$

$$\textsc{Am-Branch-Symb}\quad \dfrac{p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, \ell'' \qquad \mathit{enabled}'(s) \qquad \sigma \xrightarrow{\mathtt{symPc}(se)\cdot\mathtt{pc}\ \ell'}_s \sigma' \qquad \ell = \begin{cases} \sigma(\mathbf{pc})+1 & \text{if } \ell' \neq \sigma(\mathbf{pc})+1 \\ \ell'' & \text{if } \ell' = \sigma(\mathbf{pc})+1 \end{cases} \qquad s' = \mathit{decr}'(s) \cdot \langle \sigma, \mathit{ctr}, \mathit{min}(w, \mathit{wndw}(s)-1), \ell\rangle \qquad \mathit{id} = \mathit{ctr}}{\langle \mathit{ctr}, \sigma, s\rangle \xRightarrow{\mathtt{symPc}(se)\cdot\mathtt{start}\ \mathit{id}\cdot\mathtt{pc}\ \ell}_s \langle \mathit{ctr}+1, \sigma[\mathbf{pc} \mapsto \ell], s'\rangle}$$

$$\textsc{Am-Rollback}\quad \dfrac{\sigma' \xrightarrow{\tau}_s \sigma'' \qquad \sigma''(\mathbf{pc}) \neq \ell}{\langle \mathit{ctr}, \sigma, s \cdot \langle \sigma', \mathit{id}, 0, \ell\rangle\rangle \xRightarrow{\mathtt{rollback}\ \mathit{id}\cdot\mathtt{pc}\ \sigma''(\mathbf{pc})}_s \langle \mathit{ctr}, \sigma'', s\rangle}$$

Fig. 12. Symbolic always-mispredict speculative semantics for a program p and speculative window w

```
and     %rax, temp
```

B. *Example #15 in* SLH *mode*

Here, the adversary provides the input via the pointer `*y`:

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

In the `-O0` SLH mode, CLANG hardens the address used for performing the memory access `A[*y]` in lines 8–12, but not the resulting value, which is stored in the register `%cx`. However, the value stored in `%cx` is used to perform a second memory access at line 14. An adversary can exploit the second memory access to speculatively leak the content of `A[0xFF...FF]`. In our experiments, SPECTECTOR correctly detected such leak.

```
1  mov     $0, %rax
2  mov     y, %rdx
3  mov     (%rdx), %rsi
4  mov     size, %rdx
5  cmp     %rdx, %rsi
6  jae      END
7  cmovae $-1, %rax
8  mov     y, %rcx
9  mov     (%rcx), %rcx
10 mov     %rax, %rdx
11 or      %rcx, %rdx
12 mov     A(%rdx), %rcx
13 shl     $9, %rcx
14 mov     B(%rcx), %rcx
15 mov     temp, %rdx
16 and     %rcx, %rdx
17 mov     %rdx, temp
```

In contrast, when Example #15 is compiled with the `-O2` flag, CLANG correctly hardens `A[*y]`'s result (cf. line 10). This prevents information from flowing into the microarchitectural state during speculative execution. Indeed, SPECTECTOR proves that the program satisfies speculative non-interference.

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