SPECTECTOR: Principled Detection of Speculative Information Flows

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

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.**

We present a novel, principled approach for reasoning about software defenses against SPECTRE-style attacks. Our approach builds on *speculative non-interference*, the first semantic notion of security against speculative execution attacks. We develop SPECTECTOR, an algorithm based on symbolic execution for automatically proving speculative non-interference, or detecting violations.

We implement SPECTECTOR in a tool, and we use it to detect subtle leaks – and optimizations opportunities – in the way major compilers place SPECTRE countermeasures.

I. INTRODUCTION

Speculative execution avoids expensive pipeline stalls by predicting the outcome of branching (and other) decisions, and by speculatively executing the corresponding instructions. If a prediction turns out to be wrong, the CPU aborts the speculative execution and rolls back the effect of the speculatively executed instructions on the logical state, which consists of the CPU's registers and flags as well as the 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], [2], [3], [4], [5], [6] demonstrates that the vulnerability affects all modern general-purpose CPUs and poses a serious threat against platforms with multiple tenants.

Since the advent of SPECTRE, a number of countermeasures have been proposed and deployed. These include, for instance, the insertion of serializing instructions [7], the use of branchless bounds checks [8], and speculative load hardening [9]. Several compilers support the automated insertion of these countermeasures during compilation [10], [11], [12], and the first static analyses to help identify vulnerable code patterns are emerging [13].

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 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 semantics* without speculative execution. We use this semantics as a proxy for the intended program behavior.
- The second is a novel, *speculative semantics*, which can follow mispredicted branches for a bounded number of steps before backtracking. We use this semantics for capturing 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 semantics.

For capturing "leakage into the microarchitectural state", we consider an observer of the program execution that sees the locations of memory accesses and jump targets. This is a common way of characterizing "side-channel free" or "constant-time" code [14], [15] in the absence of detailed models of the microarchitecture. We characterize what this observer can learn in terms of an equivalence relation over initial program states, where a finer relation distinguishes between more states and corresponds to more leakage [16].

Formally, speculative non-interference requires that the relation characterizing the leak in the speculative semantics does not refine the relation characterizing the leak in the standard semantics.

Automation: We propose SPECTECTOR, a novel technique for automatically proving that programs are speculatively non-interferent. Given a program p, SPECTECTOR uses symbolic execution with respect to the speculative semantics to derive a concise representation of the trace of all mem-

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

Fig. 1. SPECTRE variant 1 - C code

ory accesses and jump targets during execution along each program path. Based on this representation, SPECTECTOR creates a SMT formula that describes that, whenever two initial states produce the same memory access patterns in the standard semantics, they also produce the same access patterns in the speculative semantics. Validity of this formula for each program path implies speculative noninterference.

Case studies: We implement a prototype of SPECTECTOR, with the Z3 SMT solver as a back-end for analysing formulae and a front-end for parsing (a subset of) x86 assembly.[1] We perform a case study where we run this prototype on a corpus of 210 small example programs. These programs are obtained by compiling the 15 variants of SPECTRE v1 by Kocher [17] 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 [17], (2) detect novel, subtle leaks that are out of scope of existing approaches that check for known vulnerable code patterns [13], and (3) identify cases where compilers unnecessarily inject countermeasures, i.e., opportunities for optimiziation without sacrificing security.

Scope: Our work focuses on detecting leaks introduced by speculatively executed instructions resulting from mispredicted branch outcomes. That is, we focus on SPECTRE v1 [2] and its variants. Generalizing speculative non-interference (and SPECTECTOR) to other forms of speculation, such as indirect jump predictors [2], return stack buffers [1], and memory disambiguation predictors [18] will require extending the speculative semantics (and the symbolic execution engine) accordingly. For a more in-depth discussion of the scope of our approach, see Section VIII.

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 three state-of-the-art compilers inject SPECTRE countermeasures.

II. ILLUSTRATIVE EXAMPLE

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

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],

[1]SPECTECTOR is available at https://spectector.github.io.

```
1     mov    size, %rax
2     mov    y, %rbx
3     cmp    %rbx, %rax
4     jbe    END
5     mov    A(%rbx), %rax
6     shl    $9, %rax
7     mov    B(%rax), %rax
8     and    %rax, temp
```

Fig. 2. SPECTRE variant 1 - Assembly code

amplifies it with a multiple of the cache line size (here: 512), and uses the result as an address for accessing the array B.

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

When size becomes available, the CPU checks whether the speculated branch is the correct one. If it is not, the CPU rolls back the architectural state's changes and executes the correct branch. However, the speculatively executed memory accesses leave a footprint in the cache, which enables an attacker 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. The distinguishing feature of SPECTECTOR is that it is backed by a sematic notion of security. Specifically, SPECTECTOR detects a leak whenever executing the program under a speculative semantics, which captures that the execution can go down a mispredicted path for a bounded number of steps, leaks more information into the microarchitecture than executing the program under a simple, non-speculative semantics.

For illustrating how SPECTECTOR operates, we consider the translation of the program in Figure 1 into x86 assembly[2], see Figure 2.

SPECTECTOR performs symbolic execution with respect to the speculative semantics to derive a concise representation of the trace 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.

The code in Figure 2 yields two symbolic traces with respect to the *speculative* semantics:[3]

$$\mathbf{start} \cdot \mathbf{rollback} \cdot \tau \quad \text{when} \quad \texttt{y} < \texttt{size} \qquad (1)$$

$$\mathbf{start} \cdot \tau \cdot \mathbf{rollback} \quad \text{when} \quad \texttt{y} \geq \texttt{size} \qquad (2)$$

where $\tau = \mathbf{loadO}\ (\texttt{A} + \texttt{y}) \cdot \mathbf{loadO}\ (\texttt{B} + \texttt{A[y]} \ * \ \texttt{512})$. Here, the argument of **loadO** is visible to the observer,

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

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

while **start** and **rollback** denote the start and the end of a mis-speculated 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., `y` $<$ `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 `y` $\geq$ `size`, the speculative execution generates observations τ that depend on variables (here: `A[y]`) that are not visible in the standard 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 [9], 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

$$\tau = \textbf{loadO}\ (\texttt{A} + \texttt{y}) \cdot \textbf{loadO}\ (\texttt{B} + (\texttt{A[y]}\ \star\ \texttt{512}) \,|\, \mathit{mask}),$$

where $\mathit{mask} = \textbf{ite}(\texttt{y} < \texttt{size}, \texttt{0x0}, \texttt{0xFF..FF})$ corresponds to the conditional move in line 6 and $|$ is a bitwise-or operator. Here, $\textbf{ite}(\texttt{y} < \texttt{size}, \texttt{0x0}, \texttt{0xFF..FF})$ is a symbolic if-then-else expression evaluating to `0x0` if `y` $<$ `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 [19]) that, for all `y` $\geq$ `size` – which we assume is public – there is exactly *one* observation that the attacker can make in the speculation, namely $\textbf{loadO}\ (\texttt{A}{+}\texttt{y}) \cdot \textbf{loadO}\ (\texttt{B}{+}\texttt{0xFF..FF})$, from which we conclude that no information leaks into the microarchitectural state, i.e., the countermeasure is securely applied. See Section VII for examples where SPECTECTOR detects that this is not the case.

III. LANGUAGE AND SEMANTICS

Here, we introduce μASM, a simple language capturing the main aspects of assembler languages.

Basic Types			
(Registers)	x	$\in$	*Regs*
(Values)	n	$\in$	$\mathit{Vals} = \mathbb{N} \cup \{\bot\}$
Syntax			
(Expressions)	e	$:=$	$n \mid x \mid \ominus e \mid e_1 \otimes e_2$
(Instructions)	i	$:=$	$\textbf{skip} \mid x \leftarrow e \mid \textbf{load}\ x, e \mid$ $\textbf{store}\ x, e \mid \textbf{jmp}\ e \mid \textbf{beqz}\ x, n \mid$ $x \overset{e'}{\leftarrow} e \mid \textbf{spbarr}$
(Programs)	p	$:=$	$n : i \mid p_1; p_2$

Fig. 4. μASM syntax

A. Syntax

μASM, whose syntax is defined in Figure 4, supports eight kinds of instructions: **skip** instructions, assignments, load and store instructions, branching instructions, indirect jumps, conditional updates, and speculation barriers. Both conditional updates and speculation barriers have been used to implement SPECTRE countermeasures [9], [7].

Assignments are of the form $x \leftarrow e$, where x is a register and e an expression on registers. μASM features two memory access instructions. A $\textbf{load}\ x, e$ instruction accesses memory at the address specified by the expression e and saves the memory's content in the register x. In contrast, a $\textbf{store}\ x, e$ instruction stores the content of the register x in the memory address pointed by the expression e. Branching is supported through the $\textbf{beqz}\ x, n$ instruction, which sets the program counter to the value n in case the value stored in the register x is 0. In contrast, indirect jumps are supported using the $\textbf{jmp}\ e$ instruction, which sets the program counter to the value of the expression e. A conditional update $x \overset{e'}{\leftarrow} e$ assigns to x the value of e only if the value of the expression e' is 0. Finally, a speculation barrier **spbarr** stops speculative execution; otherwise it has no effect.

A μASM program is a sequence of pairs $n : i$, where $n \in \mathit{Vals}$ is a value representing the instruction's label and i is an instruction. A program is *well-formed* if (1) it contains no duplicate labels, (2) it contains an instruction labeled with 0, i.e., the initial instruction, (3) it does not contain instructions labelled with the designated label $\bot$, which we use to denote termination, and (4) it does not contain instructions $n : \textbf{beqz}\ x, n+1$. In the following we consider only well-formed programs. Given a program p and a value $n \in \mathit{Vals}$, we denote by $p(n)$ the instruction labelled with n in p if it exists and $\bot$ otherwise, i.e., we treat p as a partial function from values to instruction.

B. Non-speculative semantics

Here, we formalize the standard, non-speculative semantics for μASM programs. This semantics models the behavior of μASM programs executed on a very simple microarchitecture that ignores speculative execution.

Notation. By *Regs* we denote the set of register identifiers, which contains a designated identifier **pc** representing the program counter register. μASM programs compute on natural numbers only. Hence, the set *Vals* of all values consists of the natural numbers and the designated value $\bot$, which we use to denote termination.

We use standard notation for sequences. Given a set S, S^* is the set of all finite sequences over S. We also denote by ε the empty sequence and by $s_1 \cdot s_2$ the concatenation of s_1 and s_2.

Configurations. A *memory* $m \in \mathit{Mem}$ is a function mapping memory addresses, which we represent by natural numbers, to values. In contrast, a *register assignment* $a \in \mathit{Assgn}$ is a function mapping register identifiers to values. We assume that the designated value $\bot$ can be assigned only to the program counter register $\mathbf{pc}$.

A *configuration* σ is a pair $\langle m, a\rangle \in \mathit{Mem} \times \mathit{Assgn}$, where m is the current memory and a records the values stored in the registers. Observe that $a(\mathbf{pc})$ stores the current value of the program counter, which points to the next instruction to be executed. The set *Conf* of all configurations is *Mem* $\times$ *Assgn*. A configuration $\langle m, a\rangle$ is *initial* (respectively *final*) if the program counter in a is 0 (respectively $\bot$).

Observations. We consider an attacker that observes the current program counter as well as the location of memory accesses. We capture this by annotating the transactions in our semantics with observations $o \in \mathit{Obs}$. Observations $\mathbf{loadO}\ n$ and $\mathbf{storeO}\ n$ record memory accesses to an address n and are produced by $\mathbf{load}$ and $\mathbf{store}$ instructions. Observations $\mathbf{pcO}\ n$, produced by $\mathbf{jmp}$ and $\mathbf{beqz}$ instructions, record changes to the program counter.

Evaluation relation. We capture the μASM semantics with the evaluation relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Obs} \times \mathit{Conf}$, which is fully formalized in Appendix A. Most of the rules defining $\rightarrow$ are rather standard. Figure 5 depicts some selected rules.

The rule CONDITIONALUPDATE-1 models the behavior of a conditional update whose condition is satisfied. It checks that the condition e' evaluates to 0 under the current register assignment, written $[\![e']\!](a) = 0$. The rule updates the assignment a by storing the new value associated with the register x and increments the program counter.

The rules LOAD and STORE handle memory accesses. The former assigns to the register x the memory content at the address pointed by e. The latter stores x's content at the memory location pointed by e. The rules increment the program counter and record memory accesses using observations.

The rule BEQZ-1 handles branching instructions $\mathbf{beqz}\ x$, n. It checks that x's values is 0 and sets the program counter to n. It also records the changes to the program counter by producing the observation $\mathbf{pcO}\ n$. A corresponding rule BEQZ-2 found in Appendix A handles the case in which the branch is not taken. Finally, the rule JMP executes $\mathbf{jmp}\ e$ instructions. The rule stores e's value in the program counter and records the change using an observation.

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

$$\begin{aligned}
&0 : x \leftarrow \mathtt{y} \geq \mathtt{size}\\
&1 : \mathbf{beqz}\ x, 3\\
&2 : \mathbf{jmp}\ \bot\\
&3 : \mathbf{load}\ z, \mathtt{A} + \mathtt{y}\\
&4 : z \leftarrow z * 512\\
&5 : \mathbf{load}\ w, \mathtt{B} + z\\
&6 : \mathtt{temp} \leftarrow \mathtt{temp}\ \&\ w
\end{aligned}$$

In the above program, the variables y, size, and temp are assumed to be kept in registers y, size, and temp. Similarly, the two registers A and B store the memory addresses of the first elements of the arrays A and B. ■

IV. SPECULATIVE SEMANTICS

Here, we formalize the behavior of μASM programs executed by a microarchitecture supporting a simple form of speculative execution.

Speculative states. A speculative transaction consists of all the instructions that are executed between the time when the CPU predicts a branching decision and when the CPU realizes whether its prediction was correct. We associate each speculative transaction with a unique identifier $id \in \mathbb{N}$.

Whenever a speculative transaction starts, the semantics records the prediction's outcome and the current configuration that may need to be restored. A *speculative snapshot* is a 4-tuple $\langle id, w, n, \sigma\rangle \in \mathbb{N} \times \mathbb{N} \times \mathit{Vals} \times \mathit{Conf}$ where id is the speculative transaction's identifier, w indicates the remaining speculative window, n indicates the predicted next instruction's address, and σ is the configuration that should be restored in case the speculative execution is rolled back.

A *speculative state* $s \in \mathit{SpecS}$ is a sequence of snapshots.

Executing commands inside a speculative transaction reduces the remaining speculative window. We model this with the function $\mathit{decr} : \mathit{SpecS} \rightarrow \mathit{SpecS}$ which takes a speculative state s and decrements all of its speculative windows by 1.

The speculative semantics performs a computation step only in case all the remaining speculative windows are greater than 0. When a speculative window reaches 0, the speculative transaction is either commited or rolled back. The predicate $\mathit{enabled}(s)$ denotes that all speculative windows in the state s are greater than 0.

Extended configurations. The speculative semantics operates on *extended configurations* which are triples $\langle \mathit{ctr}, \sigma, s\rangle \in \mathit{ExtConf}$ consisting of a global counter $\mathit{ctr} \in \mathbb{N}$ to generate speculative transactions' identifiers, a configuration $\sigma \in \mathit{Conf}$, and a speculative state $s \in \mathit{SpecS}$.

Extended observations. To model the speculative execution's effects, we introduce new observations. We extend the set *Obs* with observations of the form $\mathbf{start}\ id$, $\mathbf{commit}\ id$, and $\mathbf{rollback}\ id$, where $id \in \mathbb{N}$ is a speculative transaction's identifier, that denote a transaction's start, commit, and rollback. *ExtObs* denotes the set of extended observations.

CONDITIONALUPDATE-1
$$\frac{p(a(\mathbf{pc})) = x \overset{e'}{\leftarrow} e \qquad \llbracket e' \rrbracket(a) = 0 \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto \llbracket e \rrbracket(a)]\rangle}$$

LOAD
$$\frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad n = \llbracket e \rrbracket(a)}{\langle m, a\rangle \xrightarrow{\mathbf{loadO}\ n} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle}$$

STORE
$$\frac{p(a(\mathbf{pc})) = \mathbf{store}\ x, e \qquad n = \llbracket e \rrbracket(a)}{\langle m, a\rangle \xrightarrow{\mathbf{storeO}\ n} \langle m[n \mapsto a(x)], a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

BEQZ-1
$$\frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad a(x) = 0}{\langle m, a\rangle \xrightarrow{\mathbf{pcO}\ n} \langle m, a[\mathbf{pc} \mapsto n]\rangle}$$

JMP
$$\frac{p(a(\mathbf{pc})) = \mathbf{jmp}\ e \qquad n = \llbracket e \rrbracket(a)}{\langle m, a\rangle \xrightarrow{\mathbf{pcO}\ n} \langle m, a[\mathbf{pc} \mapsto n]\rangle}$$

Fig. 5. μASM semantics for a program p – Selected rules

Prediction oracle. Our goal is to model the security-critical features of speculative execution. To this end, we abstract away from internal details of branch predictors implemented in CPUs, and we model them as prediction oracles.

A *prediction oracle* $\mathcal{O}$ is a partial function that takes as input a configuration σ and a program p and returns as output a pair $\langle n, w\rangle \in \mathit{Vals} \times \mathbb{N}$, where n is the predicted next instruction's label and w is the speculation window indicating the number of steps that should be executed speculatively. We consider only oracles $\mathcal{O}$ that are defined only if the current instruction pointed by $\mathbf{pc}$ is a branch instruction. That is, our semantics only speculates over branch outcomes.

Example 2. The "backward taken forward not taken" (BTFNT) branch predictor, which predicts the branch as taken if the target instruction address is lower than the program counter, implemented in early CPUs [20] can be formalized as a simple oracle BTFNT, for a fixed speculative window w: $\mathit{BTFNT}(\langle m, a\rangle, p)$ is $\langle a(\mathbf{pc}) + 1, w\rangle$ whenever $p(a(\mathbf{pc})) = \mathbf{beqz}\ x, n$ and $n > a(\mathbf{pc})$, and $\langle n, w\rangle$ otherwise. ■

Example 3. Another security-relevant oracle is the "always mispredict" oracle, which mispredicts the outcome of every branch instruction. Using this oracle, we always speculatively execute the instructions in the wrong branch before backtracking and executing the instructions in the correct branch. The "always mispredict" oracle has already been implicitly used in previous SPECTRE detection techniques [13]. SPECTECTOR also relies on an "always mispredict" oracle, as it is often the worst predictor in terms of leaked information. ■

Evaluation relation. The speculative semantics of a program p under a prediction oracle $\mathcal{O}$ is modeled by the relation $\rightsquigarrow \subseteq \mathit{ExtConf} \times \mathit{ExtObs}^* \times \mathit{ExtConf}$ defined in Figure 6.

The rule SE-NOBRANCH executes non-branching instructions, i.e., $\mathcal{O}(\sigma, p) = \perp$, as long as no speculative window has reached 0, that is, if $\mathit{enabled}(s)$. In this case, $\rightsquigarrow$ simply mimics the behavior of the non-speculative semantics $\to$. If the instruction is not a speculation barrier, the rule also decrements the speculation windows in the speculative state. In contrast, speculation barriers $\mathbf{spbarr}$ stop the speculative execution. We formalize this by setting to 0 the speculation window of all snapshots in the current speculative state using the function $\mathit{zeroes} : \mathit{SpecS} \to \mathit{SpecS}$. That is, $\mathbf{spbarr}$ forces the termination (either with a commit or with a rollback) of all on-going speculative transactions.

The rule SE-BRANCH models the behavior of branch instructions, i.e., those for which $\mathcal{O}(\sigma, p) \neq \perp$. The rule (1) queries the prediction oracle $\mathcal{O}$ to obtain a prediction $\langle n, w\rangle$ consisting of the predicted next instruction address n and the speculation window n, (2) sets the program counter to n, (3) decrements the speculation windows in s, (4) increments the transaction counter ctr, and (5) appends a new snapshot with identifier ctr recording the speculative window w, the predicted instruction address n, and the current configuration σ. The rule also records the speculative execution's start and the program counter's change through observations.

The rule SE-COMMIT commits a speculative transaction. The rule is executed only when there is a speculative snapshot whose remaining speculative window has reached 0. The rule ensures that the prediction made for the transaction is correct by comparing the predicted address n 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 observations.

In contrast, the rule SE-ROLLBACK rolls back a speculative transaction. The rule checks that the prediction is incorrect (again by comparing the predicted address n with the one obtained from the non-speculative semantics), and it restores the configuration stored in s. Rolling back a transaction in the middle of the speculative state 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 speculative execution's rollback and the program counter's change.

Example 4. Let us now execute the program from Example 1 with the "always mispredict" oracle from Example 3. Consider an initial configuration $\langle 0, \langle m, a\rangle, \varepsilon\rangle$ in which $a(\mathtt{y})$ is greater than $a(\mathtt{size})$. Executing the program produces the trace $\tau := \mathbf{start}\ 0 \cdot \mathbf{pcO}\ 3 \cdot \mathbf{loadO}\ v_1 \cdot \mathbf{loadO}\ v_2 \cdot \mathbf{rollback}\ 0 \cdot \mathbf{pcO}\ \perp$, where $v_1 = a(\mathtt{A}) + a(\mathtt{y})$ and $v_2 = a(\mathtt{B}) + v_1 * 512$. First, the rule SE-NOJUMP is applied to execute the assignment $x \leftarrow \mathtt{y} \geq \mathtt{size}$. Then, the branch instruction $\mathbf{beqz}\ x, 3$ is reached and so rule SE-JUMP applies. The rule queries the "always mispredict" oracle, which predicts that the branch is not taken. This produces the observations $\mathbf{start}\ 0$, modeling the beginning of a speculative transaction, and $\mathbf{pcO}\ 3$, representing the program counter's change. Next, we apply the rule SE-NOJUMP three times to execute the instructions 3–6, thereby producing the observations $\mathbf{loadO}\ v_1$ and $\mathbf{loadO}\ v_2$ that record the memory accesses. Finally, the rule SE-ROLLBACK applies, which terminates the speculative transaction and rolls back its effects. This rule produces the observations $\mathbf{rollback}\ 0$ and $\mathbf{pcO}\ \bot$. ■

Runs. Runs are 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 and an oracle $\mathcal{O}$, we denote by $(\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$ the set of all possible runs of the speculative semantics, i.e., it contains all triples $\langle \sigma, \tau, \sigma' \rangle$ corresponding to runs $\langle 0, \sigma, \varepsilon \rangle \overset{\tau}{\rightsquigarrow}^* \langle ctr, \sigma', \varepsilon \rangle$. Similarly, $(\!|p|\!)_{\rightarrow}$ denotes the set of all possible runs of the non-speculative semantics, i.e., it contains all triples $\langle \sigma, \tau, \sigma' \rangle$ corresponding to runs $\sigma \xrightarrow{\tau}^* \sigma'$.

Projections. Given a trace τ produced by the speculative semantics, its non-speculative projection contains only the observations that are produced by committed transactions or that are produced non-speculatively; in other words, rolled-back transactions are removed in the projection. Formally, the non-speculative projection $\tau{\upharpoonright_{nse}}$ is the trace obtained by removing from τ (1) all substrings $\mathbf{start}\ id \cdot \tau' \cdot \mathbf{rollback}\ id$, and (2) all the remaining extended observations.

In contrast, τ's speculative projection $\tau{\upharpoonright_{se}}$ contains all the observations produced by rolled-back transactions. That is, $\tau{\upharpoonright_{se}}$ is obtained by concatenating all the substrings τ' discarded by $\tau{\upharpoonright_{nse}}$, i.e., those for which there is a maximal substring $\mathbf{start}\ id \cdot \tau' \cdot \mathbf{rollback}\ id$ in τ.

We formalize both $\tau{\upharpoonright_{nse}}$ and $\tau{\upharpoonright_{se}}$ in Appendix B.

Speculative and non-speculative semantics. The speculative and non-speculative semantics are tightly connected. As stated in Proposition 1, every speculative run $\langle \sigma, \tau, \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$ corresponds to a non-speculative run $\langle \sigma, \tau', \sigma' \rangle \in (\!|p|\!)_{\rightarrow}$ such that τ' agrees with τ's non-speculative projection, and vice versa. That is, from a speculative run $\langle \sigma, \tau, \sigma' \rangle$ one can recover the program's non-speculative behavior by ignoring the speculative observations, i.e., by computing $\tau{\upharpoonright_{nse}}$.

Proposition 1. *Let p be a program and $\mathcal{O}$ be an oracle.*
(a) If $\langle \sigma, \tau, \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$, then $\langle \sigma, \tau{\upharpoonright_{nse}}, \sigma' \rangle \in (\!|p|\!)_{\rightarrow}$.
(b) If $\langle \sigma, \tau, \sigma' \rangle \in (\!|p|\!)_{\rightarrow}$, then there is a τ' such that $\langle \sigma, \tau', \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$ and $\tau = \tau'{\upharpoonright_{nse}}$.

V. SPECULATIVE NON-INTERFERENCE

Here, we define a semantic notion of security characterizing leaks caused by speculative execution.

Security Policy. Our notion is parameterized by a *security policy* P specifying which parts of the state are known or controlled by an adversary, i.e., "public" or "low" data. Formally, P is a finite subset of *Regs* $\cup\, \mathbb{N}$ specifying the non-sensitive register identifiers and memory addresses.

Two configurations $\sigma, \sigma' \in$ *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 5. 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.

Attacker Knowledge. We characterize what attackers can infer from an execution in terms of their *knowledge*, which is commonly defined as the set of all possible initial configurations that are coherent with their observations [21]. Definition 1 formalizes knowledge with respect to the speculative and non-speculative semantics.

Definition 1. Let p be a program and $\mathcal{O}$ be an oracle.
- The *speculative knowledge* $K_{\rightsquigarrow}(\tau, \sigma, P)$, given a sequence of observations $\tau \in$ *ExtObs**, an initial configuration σ, and a policy P, is the set of configurations $\{\sigma_0 \mid \exists \sigma'.\ \langle \sigma_0, \tau, \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow} \wedge \sigma \sim_P \sigma_0\}$.
- The *non-speculative knowledge* $K_{\rightarrow}(\tau, \sigma, P)$, given a sequence of observations $\tau \in$ *Obs**, an initial configuration σ, and a policy P, is the set of configurations $\{\sigma_0 \mid \exists \sigma'.\ \langle \sigma_0, \tau, \sigma' \rangle \in (\!|p|\!)_{\rightarrow} \wedge \sigma \sim_P \sigma_0\}$.

Defining speculative non-interference. Informally, speculative non-interference requires that executing a program under the speculative semantics leaks at most as much information as executing the program under the non-speculative semantics. Formally, we require that the non-speculative knowledge is always contained in the speculative knowledge.

Definition 2. A program p satisfies *speculative non-interference* for an oracle $\mathcal{O}$ and a policy P iff for all $\langle \sigma, \tau, \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$, we have $K_{\rightarrow}(\tau{\upharpoonright_{nse}}, \sigma, P) \subseteq K_{\rightsquigarrow}(\tau, \sigma, P)$.

Example 6. The program p from Example 1 does not satisfy speculative non-interference for the "always mispredict" oracle from Example 3 and the policy P from Example 5. Consider a run $\langle \sigma, \tau, \sigma' \rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$, where $\sigma := \langle m, a \rangle$ and $\mathtt{y}$ is greater than $\mathtt{size}$. Executing the program under the speculative semantics produces the trace $\tau = \mathbf{start}\ 0 \cdot \mathbf{pcO}\ 3 \cdot \mathbf{loadO}\ v_1 \cdot \mathbf{loadO}\ v_2 \cdot \mathbf{rollback}\ 0 \cdot \mathbf{pcO}\ \bot$, where $v_1 = a(\mathtt{A}) + a(\mathtt{y})$ and $v_2 = a(\mathtt{B}) + m(v_1) * 512$ (see Example 4). Thus, the speculative knowledge $K_{\rightsquigarrow}(\tau, \sigma, P)$ is $\{\langle m_0, a_0 \rangle \in \mathit{InitConf} \mid a(\mathtt{y}) = a_0(\mathtt{y}) \wedge a(\mathtt{size}) = a_0(\mathtt{size}) \wedge a(\mathtt{A}) = a_0(\mathtt{A}) \wedge a(\mathtt{B}) = a_0(\mathtt{B}) \wedge a(\mathtt{y}) \geq a(\mathtt{size}) \wedge m_0(a(\mathtt{A}) + a(\mathtt{y})) = m(a(\mathtt{A}) + a(\mathtt{y})) \wedge m_0(a(\mathtt{B}) + (a(\mathtt{A}) + a(\mathtt{y})) * 512) = m(a(\mathtt{B}) + (a(\mathtt{A}) + a(\mathtt{y})) * 512)\}$.

In contrast, the non-speculative projection of τ is $\tau{\upharpoonright_{nse}} = \varepsilon$ and the non-speculative knowledge $K_{\rightarrow}(\tau{\upharpoonright_{nse}}, \sigma, P)$ is $\{\langle m_0, a_0 \rangle \in \mathit{InitConf} \mid a(\mathtt{y}) = a_0(\mathtt{y}) \wedge a(\mathtt{size}) = a_0(\mathtt{size}) \wedge a(\mathtt{A}) = a_0(\mathtt{A}) \wedge a(\mathtt{B}) = a_0(\mathtt{B}) \wedge a(\mathtt{y}) \geq a(\mathtt{size})\}$. Therefore, p does not satisfy speculative non-interference since $K_{\rightarrow}(\tau{\upharpoonright_{nse}}, \sigma, P) \not\subseteq K_{\rightsquigarrow}(\tau, \sigma, P)$. ■

VI. DETECTING SPECULATIVE INFORMATION FLOWS

We now present SPECTECTOR, an approach to automatically check speculative non-interference, i.e., to detect speculative leaks or to prove their absence. SPECTECTOR relies on symbolic execution of the program p under analysis to derive a concise representation of p's behavior as a set of symbolic

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

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

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

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

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

traces. It analyses each symbolic trace to detect possible speculative leaks through memory accesses or control-flow instructions. If neither of such leaks are detected, SPECTECTOR deems the program as secure.

As prediction oracle, SPECTECTOR relies on the "always mispredict" oracle from Example 3 for a given speculative window w, which we fix for the rest of the section.

A. Symbolically executing μASM programs

Here, we define a symbolic semantics for μASM programs. The goal of our symbolic semantics is to concisely represent all possible concrete traces that can be produced by a program.

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 operator $\ominus se$ or a binary operator $se \otimes se'$.

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

The value of an expression se depends on a *model* $\mu : \mathit{SymbVals} \rightarrow \mathit{Vals}$ mapping symbolic values to concrete ones. The evaluation $\mu(se)$ of se under μ is standard, and we formalize it in Appendix C. An expression se is *satisfiable* if there is a model μ such that $\mu(se) \neq 0$.

Symbolic memories. We model symbolic memories as symbolic arrays using the standard theory of arrays [22]. That is, we model memory updates as triples of the from $\mathbf{write}(sm, se, se')$, 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 $mem : \mathbb{N} \rightarrow \mathit{SymbVals}$ 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.

$$sm := mem \mid \mathbf{write}(sm, se, se')$$
$$se := \ldots \mid \mathbf{read}(sm, se)$$

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

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 snapshots and speculative states to symbolic configurations. A *symbolic extended configuration* is a triple $\langle \mathit{ctr}, \sigma, s\rangle$ where $\mathit{ctr} \in \mathbb{N}$ is a counter, $\sigma \in \mathit{Conf}$ 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 $\mathbf{loadO}\ se$ and $\mathbf{storeO}\ se$, which are produced when symbolically executing $\mathbf{load}$ and $\mathbf{store}$ commands, and $\mathbf{symPc}(se)$, produced when symbolically evaluating branching instructions, where se is a symbolic expression. In our symbolic semantics, we use the observations $\mathbf{symPc}(se)$ to represent the symbolic path condition indicating when a path is feasible. Given a sequence of symbolic observations τ and a model μ, $\mu(\tau)$ denotes the concrete trace obtained by evaluating all symbolic observations different from $\mathbf{symPc}(se)$ under μ.

Symbolic non-speculative semantics. We model the symbolic non-speculative semantics of a program p using the relation $\rightarrow_s$, which we fully formalize in Appendix C. Figure 7 depicts some selected rules.

The rule LOAD-SYMB executes $\mathbf{load}\ x, e$ instructions where the memory address in e evaluates to a symbolic expression se. The rule produces the symbolic observation $\mathbf{loadO}\ se$ and sets value of the register x to $\mathbf{read}(sm, se)$.

In contrast, the rule STORE-SYMB executes $\mathbf{store}\ x, e$ instructions. The rule records the memory access with the observation $\mathbf{storeO}\ se$. It also updates the symbolic memory sm by recording the write operation $\mathbf{write}(sm, se, sa(x))$.

Finally, the rule BEQZ-SYMB-1 executes branch instructions $\mathbf{beqz}\ x, n$ whose condition is satisfied, i.e., the symbolic value associated with x evaluates to 0. The rule sets

LOAD-SYMB

$$\frac{p(sa(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \quad se = [\![e]\!](sa) \quad se \notin \mathit{Vals} \quad se' = \mathbf{read}(sm, se)}{\langle sm, sa\rangle \xrightarrow{\mathbf{loadO}\ se}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1, x \mapsto se']\rangle}$$

STORE-SYMB

$$\frac{p(sa(\mathbf{pc})) = \mathbf{store}\ x, e \qquad se = [\![e]\!](sa) \quad se \notin \mathit{Vals} \quad sm' = \mathbf{write}(sm, se, sa(x))}{\langle sm, sa\rangle \xrightarrow{\mathbf{storeO}\ se}_s \langle sm', sa[\mathbf{pc} \mapsto sa(\mathbf{pc})+1]\rangle}$$

BEQZ-SYMB-1

$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad sa(x) \notin \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\mathbf{symPc}(sa(x)=0)\cdot \mathbf{pcO}\ n}_s \langle sm, sa[\mathbf{pc} \mapsto n]\rangle}$$

Fig. 7. μASM symbolic semantics - selected rules

the program counter to n and produces the observations $\mathbf{symPc}(sa(x) = 0)$ and $\mathbf{pcO}\ n$.

Symbolic speculative semantics. The symbolic speculative semantics is captured by the relation $\rightsquigarrow_s$ in Figure 8.

The rule SE-NOBRANCH symbolically executes all non-branching instructions. The rule is similar to its non-symbolic counterpart, and it delegates the computation to the symbolic non-speculative semantics.

The rule SE-BRANCH-SYMB symbolically executes branching instructions. The rule uses the non-speculative semantics to execute the instruction. Afterwards, it mimics the behavior of the "always mispredict" oracle by setting the program counter to the branch not taken by the non-speculative semantics. The rule uses the auxiliary function $\mathit{mispred}(p, \sigma, n)$, where p is a program, σ is a configuration, and n is an instruction label, to mis-predict the program counter. That is, for a branch instruction $\mathbf{beqz}\ x, n$, when the branch target is n, $\mathit{mispred}$ returns $\sigma(\mathbf{pc}) + 1$ (i.e., predicts the branch to be not taken), and when the branch target is $\sigma(\mathbf{pc}) + 1$, $\mathit{mispred}$ predicts the branch to be taken and returns n. Formally, $\mathit{mispred}(p, \sigma, n) = \sigma(\mathbf{pc}) + 1$ and $\mathit{mispred}(p, \sigma, \sigma(\mathbf{pc}) + 1) = n$ whenever $p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, n$.

The rule SE-ROLLBACK handles the roll-back of speculative transactions. Since we target the "always mispredict" oracle, speculative transactions are always rolled back. Hence, there is no need to check whether the prediction is correct. Observe also that there is no rule for committing transactions.

Computing symbolic runs and traces. We now fix the symbolic values. The set $\mathit{SymbVals}$ 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 $(\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}$ contains all runs that can be derived using the symbolic speculative semantics starting from the initial configuration $\langle sm_0, sa_0\rangle$. That is, $(\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}$ 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 $\langle 0, \langle sm_0, sa_0\rangle, \varepsilon\rangle \overset{\tau}{\rightsquigarrow}_s^* \langle ctr, \sigma', \varepsilon\rangle$ where the path condition $\bigwedge_{\mathbf{symPc}(se)\in\tau} se$ is satisfiable.

We compute $(\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}$ in the standard way. We keep track of a symbolic path constraint PC and we update it whenever the semantics produces an observation $\mathbf{symPc}(se)$. We start the computation from $\langle 0, \langle sm_0, sa_0\rangle, \varepsilon\rangle$ and $PC = \top$. Whenever we execute the rule SE-BRANCH-SYMB, we explore all branches that are consistent with the current PC, and, for each of them, we update PC.

The set $\mathit{traces}(p)$ contains all symbolic traces produced by the program p, and it is $\{\tau \mid \exists \sigma, \sigma'.\ \langle \sigma, \tau, \sigma'\rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}\}$.

Concrete and symbolic semantics. Proposition 2 links the concrete and the symbolic semantics. It states that (1) all the concretizations of a symbolic run correspond to concrete runs, and (2) each concrete run is captured by some symbolic run.

Proposition 2. *Let p be a program and $\mathcal{O}$ be an "always mispredict" oracle.*

(a) Whenever $\langle\langle sm_0, sa_0\rangle, \tau, \langle sm, sa\rangle\rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}$, then for all models μ satisfying $\bigwedge_{\mathbf{symPc}(se)\in\tau} se$, we have $\langle\langle \mu(sm_0), \mu(sa_0)\rangle, \mu(\tau), \langle \mu(sm), \mu(sa)\rangle\rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$.

(b) Whenever $\langle\langle m, a\rangle, \tau, \langle m', a'\rangle\rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow}$, then there is a run $\langle\langle sm_0, sa_0\rangle, \tau', \langle sm, sa\rangle\rangle \in (\!|p, \mathcal{O}|\!)_{\rightsquigarrow_s}$ and a model μ satisfying $\bigwedge_{\mathbf{symPc}(se)\in\tau'} se$ such that $\langle m, a\rangle = \langle \mu(sm_0), \mu(sa_0)\rangle$, $\tau = \mu(\tau')$, and $\langle m', a'\rangle = \langle \mu(sm), \mu(sa)\rangle$.

Example 7. Executing the program from Example 1 under the symbolic speculative semantics with the "always mispredict" oracle with speculative window 2 leads to the following symbolic traces: $t_1^s := \mathbf{symPc}(\mathtt{y}_s < \mathtt{size}_s) \cdot \mathbf{start}\ 0 \cdot \mathbf{pcO}\ 2 \cdot \mathbf{pcO}\ 10 \cdot \mathbf{rollback}\ 0 \cdot \mathbf{pcO}\ 3 \cdot \mathbf{loadO}\ \mathtt{A}_s + \mathtt{y}_s \cdot \mathbf{loadO}\ \mathbf{read}(sm_0, \mathtt{B}_s{+}(\mathtt{A}_s{+}\mathtt{y}_s){*}512)$ and $t_2^s := \mathbf{symPc}(\mathtt{y}_s \geq \mathtt{size}_s) \cdot \mathbf{start}\ 0 \cdot \mathbf{pcO}\ 3 \cdot \mathbf{loadO}\ \mathtt{A}_s + \mathtt{y}_s \cdot \mathbf{loadO}\ \mathbf{read}(sm_0, \mathtt{B}_s + (\mathtt{A}_s + \mathtt{y}_s) * 512) \cdot \mathbf{rollback}\ 0 \cdot \mathbf{pcO}\ 2 \cdot \mathbf{pcO}\ 10$. The trace t_1^s is produced whenever the bound check is satisfied, that is, under the condition $in_s < bound_s$. The trace t_2^s, instead, captures the program's behavior when the bound check is not satisfied, i.e., $in_s \geq bound_s$. ■

B. Checking speculative non-interference

SPECTECTOR is shown in Algorithm 1. It relies on the two procedures CHECK_MEM_LEAK and CHECK_CTRL_LEAK to detect leaks resulting from memory accesses and from control-flow instructions, respectively. We start by discussing the high-level SPECTECTOR algorithm and next explain the CHECK_MEM_LEAK and CHECK_CTRL_LEAK procedures.

SPECTECTOR. SPECTECTOR takes as input a program p and a policy P specifying the non-sensitive information. The algorithms iterates over all symbolic traces produced by the symbolic speculative semantics (line 3). For each trace τ, the algorithm checks whether τ speculatively leaks information through memory accesses (lines 3–4) or control-flow instructions (lines 5–6). If this is the case, then SPECTECTOR has found a counterexample to p's security and it flags p as INSECURE. If SPECTECTOR analyses all traces without finding leaks, it terminates by returning SECURE.

Se-NoBranch

$$\frac{\mathcal{O}(\sigma, p) = \bot \qquad \sigma \xrightarrow{\tau}_s \sigma' \qquad \mathit{enabled}(s) \qquad s' = \begin{cases} \mathit{decr}(s) & \text{if } p(\sigma(\mathbf{pc})) \neq \mathbf{spbarr} \\ \mathit{zeroes}(s) & \text{otherwise} \end{cases}}{\langle \mathit{ctr}, \sigma, s\rangle \overset{\tau}{\rightsquigarrow}_s \langle \mathit{ctr}, \sigma', s'\rangle}$$

Se-Branch-Symb

$$\frac{\sigma = \langle \mathit{sm}, \mathit{sa}\rangle \qquad \mathit{enabled}(s) \qquad \sigma \xrightarrow{\mathbf{symPc}(\mathit{se}) \cdot \mathbf{pcO}\ n'}_s \sigma' \qquad n = \mathit{mispred}(p, \sigma, n') \qquad s' = \mathit{decr}(s) \cdot \langle \mathit{ctr}, w, n, \sigma\rangle}{\langle \mathit{ctr}, \sigma, s\rangle \overset{\mathbf{symPc}(\mathit{se}) \cdot \mathbf{start}\ \mathit{ctr} \cdot \mathbf{pcO}\ n}{\rightsquigarrow}_s \langle \mathit{ctr}+1, \langle \mathit{sm}, \mathit{sa}[\mathbf{pc} \mapsto n]\rangle, s'\rangle}$$

Se-Rollback

$$\frac{\sigma' \xrightarrow{\tau}_s \langle \mathit{sm}, \mathit{sa}\rangle \qquad \mathit{enabled}(s)}{\langle \mathit{ctr}, \sigma, s \cdot \langle \mathit{id}, 0, \ell, \sigma'\rangle \cdot s'\rangle \overset{\mathbf{rollback}\ \mathit{id} \cdot \mathbf{pcO}\ \mathit{sa}(\mathbf{pc})}{\rightsquigarrow}_s \langle \mathit{ctr}, \langle \mathit{sm}, \mathit{sa}\rangle, s\rangle}$$

Fig. 8. Symbolic speculative semantics for a program p and the "always mispredict" oracle $\mathcal{O}$ with speculative window w

Algorithm 1 Spectector

Input: p is a program and P is the policy

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

```
 1: procedure Spectector($p, P$)
 2:     for each symbolic trace $\tau \in \mathit{traces}(p)$ do
 3:         if Check_Mem_Leak($\tau, P$) $= \top$ then
 4:             return Insecure
 5:         if Check_Ctrl_Leak($\tau, P$) $= \top$ then
 6:             return Insecure
 7:     return Secure

 8: procedure Check_Mem_Leak($\tau, P$)
 9:     $\psi \leftarrow \mathit{pthCnd}(\tau) \wedge \mathit{polEqv}(P) \wedge \mathit{obsEqv}(\tau\upharpoonright_{nse}) \wedge$
            $\neg\mathit{obsEqv}(\tau\upharpoonright_{se})$
10:     return Satisfiable($\psi$)

11: procedure Check_Ctrl_Leak($\tau, P$)
12:     for each prefix $\nu \cdot \mathbf{symPc}(\mathit{se})$ of $\tau\upharpoonright_{se}$ do
13:         $\psi \leftarrow \mathit{pthCnd}(\tau\upharpoonright_{nse} \cdot \nu) \wedge \mathit{polEqv}(P) \wedge$
                $\mathit{obsEqv}(\tau\upharpoonright_{nse}) \wedge \neg\mathit{sameSymbPc}(\mathit{se})$
14:         if Satisfiable($\psi$) then
15:             return $\top$
16:     return $\bot$
```

We remark that considering only the symbolic speculative traces is sufficient, since the non-speculative traces can always be derived from the speculative ones (cf. Proposition 1).

Detecting leaks caused by memory accesses. The procedure Check_Mem_Leak takes as input a speculative trace τ and a policy P and determines whether τ may leak information through symbolic **load** and **store** observations. Check_Mem_Leak reduces this task to checking the satisfiability of a symbolic constraint ψ. This is inspired by approaches for proving a program's non-interference using self-composition [23]. That is, the formula ψ models the values of two possible concretizations of τ. As is standard in self-composition [23], given a symbolic constraint se, we write se_1 (respectively se_2) to denote the constraint obtained by replacing each symbolic variable x with x_1 (respectively x_2). Variables with subscript 1 refer to the first concretization of τ, while variables with subscript 2 refer to the second concretization.

To construct the symbolic formula ψ, the procedure relies on the auxiliary functions $\mathit{pthCnd}(\tau)$, $\mathit{obsEqv}(\tau)$, and $\mathit{polEqv}(P)$. The function $\mathit{pthCnd}(\tau)$ derives a constraint representing the path conditions that must be satisfied by both concretizations of τ. The path condition is obtained by conjoining $\mathit{se}_1 \wedge \mathit{se}_2$ for all $\mathbf{symPc}(\mathit{se})$ in τ. In contrast, $\mathit{obsEqv}(\tau)$ ensures that the observations associated with memory accesses are the same across the two concretizations. This is obtained by introducing a constraint $\mathit{se}_1 = \mathit{se}_2$ for each $\mathbf{loadO}\ \mathit{se}$ or $\mathbf{storeO}\ \mathit{se}$ in τ. Finally, $\mathit{polEqv}(P)$ ensures that the concretizations agree on the non-sensitive values in the policy P.

Check_Mem_Leak computes τ's non-speculative projection $\tau\upharpoonright_{nse}$, which is obtained from τ by removing all substrings $\mathbf{start}\ \mathit{id} \cdot \tau' \cdot \mathbf{rollback}\ \mathit{id}$. Additionally, the procedure computes τ's speculative projection $\tau\upharpoonright_{se}$, which contains all the observations produced by rolled-back transactions. While $\tau\upharpoonright_{se}$ captures the leaks caused only by the speculatively executed instructions, $\tau\upharpoonright_{nse}$ models the leaks caused by the instructions that are executed with respect to the non-speculative semantics (cf. Proposition 1).

To detect leaks, Check_Mem_Leak checks the satisfiability of the conjunction of $\mathit{pthCnd}(\tau)$, $\mathit{polEqv}(P)$, $\mathit{obsEqv}(\tau\upharpoonright_{nse})$, and $\neg\mathit{obsEqv}(\tau\upharpoonright_{se})$. Whenever the formula is satisfied, there are two P-indistinguishable configurations that produce the same non-speculative traces (since $\mathit{obsEqv}(\tau\upharpoonright_{nse})$ is satisfied) but whose speculative traces differ in a memory access observation (since $\neg\mathit{obsEqv}(\tau\upharpoonright_{se})$ is satisfied). Hence, speculative non-interference is violated.

Detecting leaks caused by control-flow instructions. To detect leaks caused by the outcome of branch and jump instructions (i.e., by $\mathbf{pcO}\ n$ observations), Check_Ctrl_Leak looks for two possible concretizations of τ that differ in the value of an observation $\mathbf{symPc}(\mathit{se})$ in τ's speculative projection. Since such a difference directly implies a difference in the program counter and, therefore, in the executed code, Check_Ctrl_Leak compares prefixes of the speculative projection of τ.

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

CHECK_CTRL_LEAK checks, for each prefix $\nu \cdot \mathbf{symPc}(se)$ in τ's speculative projection $\tau\upharpoonright_{se}$, the satisfiability of the conjunction of $pthCnd(\tau\upharpoonright_{nse} \cdot \nu)$, $polEqv(P)$, $obsEqv(\tau\upharpoonright_{nse})$, and $\neg(se_1 \leftrightarrow se_2)$. If the formula is satisfiable for one of the prefixes, CHECK_CTRL_LEAK detects a speculative leak and returns $\top$. Whenever the formula is satisfied, there are two P-indistinguishable configurations that produce the same non-speculative traces, but whose speculative traces differ on program counter observations. Again, this violates speculative non-interference.

Correctness. Theorem 1 states that if SPECTECTOR proves a program secure, the program is speculatively non-interferent.

Theorem 1. *Whenever* SPECTECTOR(p, P) = SECURE, *the program p satisfies speculative non-interference with respect to the "always mispredict" oracle with window w and the policy P.*

Proof. If SPECTECTOR(p, P) = SECURE, for all symbolic traces τ, CHECK_MEM_LEAK$(\tau) = \bot$ and CHECK_CTRL_LEAK$(\tau) = \bot$. Let σ and σ' be two arbitrary P-indistinguishable initial configurations producing the same concrete non-speculative trace τ. From Proposition 1.(b), σ and σ' produce two concrete speculative traces τ_c and τ_c' with the same non-speculative projection. From Propositions 2.(b), τ_c and τ_c' correspond to two symbolic traces τ_s and τ_s'. Since CHECK_CTRL_LEAK$(\tau_s) = \bot$, CHECK_CTRL_LEAK$(\tau_s') = \bot$, and $\tau_c\upharpoonright_{nse} = \tau_c'\upharpoonright_{nse}$, speculatively executed control-flow instructions produce the same outcome in τ_c and τ_c'. Hence, the same code is executed in both traces and $\tau_s = \tau_s'$. From CHECK_MEM_LEAK$(\tau_s) = \bot$, the observations produced by speculatively executed **load** and **store** instructions are the same. Thus, $\tau_c = \tau_c'$. Hence, whenever two initial configurations produce the same non-speculative traces, then they produce the same speculative traces. Therefore, p satisfies speculative non-interference. □

Theorem 2 states that the leaks found by SPECTECTOR are valid counterexamples to speculative non-interference.

Theorem 2. *Whenever* SPECTECTOR(p, P) = INSECURE, *the program p does not satisfy speculative non-interference with respect to the "always mispredict" oracle with window w and the policy P.*

Proof. If SPECTECTOR(p, P) = INSECURE, there is a symbolic trace τ for which either CHECK_MEM_LEAK$(\tau) = \top$ or CHECK_CTRL_LEAK$(\tau) = \top$. In the first case, $pthCnd(\tau) \wedge polEqv(P) \wedge obsEqv(\tau\upharpoonright_{nse}) \wedge \neg obsEqv(\tau\upharpoonright_{se})$ is satisfiable. Then, there are two models for the symbolic trace τ that (1) satisfy the path condition encoded in τ, (2) agree on the non-sensitive registers and memory locations in P, (3) produce the same non-speculative projection, and (4) the speculative projections differ on a **load** or **store** observation. From Proposition 2.(a), the two concretizations correspond to two concrete runs, with different traces, whose non-speculative projection are the same. By combining this with Proposition 1.(a), there are two configurations that produce the same non-speculative trace but different speculative traces. This is a violation of speculative non-interference.

In the second case, there is a prefix $\nu \cdot \mathbf{symPc}(se)$ of $\tau\upharpoonright_{se}$ such that $pthCnd(\tau\upharpoonright_{nse} \cdot \nu) \wedge polEqv(P) \wedge obsEqv(\tau\upharpoonright_{nse}) \wedge \neg(se_1 \leftrightarrow se_2)$ is satisfiable. Hence, there are two symbolic traces τ and τ' that produce the same non-speculative observations but differ on a program counter observation **pcO** n in their speculative projections. Again, this implies, through Propositions 2.(a) and 1.(a), that there are two P-indistinguishable initial configurations producing the same non-speculative traces but distinct speculative traces, leading to a violation of speculative non-interference. □

Example 8. SPECTECTOR detects the speculative leak in the program from Example 1. When analyzing the symbolic trace $\tau_2^s := \mathbf{symPc}(\mathtt{y}_s \geq \mathtt{size}_s) \cdot \mathbf{start}\ 0 \cdot \mathbf{pcO}\ 3 \cdot \mathbf{loadO}\ \mathtt{A}_s + \mathtt{y}_s \cdot \mathbf{loadO}\ \mathbf{read}(sm_0, \mathtt{B}_s + (\mathtt{A}_s + \mathtt{y}_s) * 512) \cdot \mathbf{rollback}\ 0 \cdot \mathbf{pcO}\ 2 \cdot \mathbf{pcO}\ 10$ produced by the symbolic semantics (see Example 7), CHECK_MEM_LEAK detects a leak caused by the observation $\mathbf{loadO}\ \mathbf{read}(sm_0, \mathtt{B}_s + (\mathtt{A}_s + \mathtt{y}_s) * 512)$. Specifically, CHECK_MEM_LEAK detects that $\mathbf{read}(sm_0, \mathtt{B}_s + (\mathtt{A}_s + \mathtt{y}_s) * 512)$ depends on sensitive information that is not disclosed by the non-speculative projection of τ_2^s. ■

C. Implementation

We implemented Algorithm 1 on top of the CIAO logic programming system [24] and an interface to external SMTLIB compatible SMT solvers. SPECTECTOR is available at `https://spectector.github.io`.

x86 front-end. SPECTECTOR analyzes μASM programs. To analyze x86 programs, we developed a front-end that translates (a fragment of) the x86 instruction set into μASM. The tool supports both AT&T/GAS and Intel style assembly files.

Our front-end covers a subset of x86 instructions. It supports instructions for copying data between registers and memory (**mov**, etc.), logical and arithmetic instructions (**xor**, **add**, etc.), branching and jumping instructions (**jae**, **jmp**, etc.), as well as conditional moves (**cmovae**, etc.). Additionally, we model the update of flag bits for both signed and unsigned integer comparisons (e.g., CF, ZF, SF) resulting from comparison instructions, like **cmp** and **test**, and logical and arithmetic instructions. Finally, our tool supports **push** and **pop** instructions for accessing the stack, and it supports function calls using **call** and **ret** instructions. The front-end supports 64-bit registers and a 64-bit address space.

Front-end limitations. Our translation, however, makes some simplifications. First, we support neither sub-registers (like `eax`, `ah`, and `al`) nor unaligned memory accesses, i.e., we assume that only 64-bit words are read/written at each address without overlaps. Second, the program instructions are not stored in the main memory. That is, code is not self-readable or self-modifiable. Finally, our implementation of **call** and **ret** instructions only models the so-called “near calls”, where the called function is in the same code segment as the instruction pointed by the program counter. These limitations do not affect the results we present in Section VII. A more accurate translation seems feasible with moderate effort. While it would imply more involved symbolic traces, we do not expect that it could negatively affect the scalability of the tool.

Checking constraint satisfiability. Our implementation uses the Z3 SMT solver [19] to check the satisfiability of symbolic constraints over the theories of Arrays and Bitvectors. This allows us to precisely reason about memory configurations and operations over 64-bit words. The solver is invoked during the computation of symbolic traces as well as in the CHECK_MEM_LEAK and CHECK_CTRL_LEAK procedures.

VII. CASE STUDIES

Here, we report the results of using SPECTECTOR to analyze a corpus of 210 programs, which are obtained from the 15 variants of the SPECTRE v1 attack by Kocher [17].

A. Experimental setup

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

We compile the programs using two different *optimization levels* (`-O0` and `-O2`) and three *mitigation levels*:

- Unpatched (UNP): we compile without any SPECTRE mitigations.
- With fences (FEN): we compile with automated injection of speculation barriers.[4]
- Speculative load hardening (SLH): we compile using speculative load hardening.[5]

We compile each of the 15 examples from [17] with each of the 3 compilers, each of the 2 optimization levels, and each of the 2-3 mitigation levels, obtaining a corpus of 210 x64 assembly programs.[6]

For each of these programs, we manually specify a security policy that flags as “low” all registers and memory locations that can either be controlled by the attacker 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 9 depicts the results of applying SPECTECTOR to the 210 examples. We highlight the following findings:

[4]Insertion of fences is supported by CLANG with the flag `-x86-speculative-load-hardening-lfence`, by ICC with `-mconditional-branch=all-fix`, and by VISUAL C++ with `/Qspectre`

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

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

- 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];
```

With optimization level `-O0`, this is translated by all compilers to a (vulnerable) branch instruction, and with level `-O2` to a (safe) conditional move, thus closing the leak. See Appendix D-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 CPUs 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 [17]. Our analysis differs from Kocher’s only on Example #8, where the latest version of the compiler introduces a safe conditional move, as explained above. Moreover, when using `-O0` (which is not considered in [17]), SPECTECTOR can establish the security of Examples 3 and 5.
- SPECTECTOR can prove the security of speculative load hardening in Clang (see the SLH column for CLANG), except for Example 10 and 15 with `-O2`.

We now discuss speculative load hardening in more detail.

Example 10 with Speculative Load Hardening: Example #10 differs from the example in 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, with `-O2`, CLANG outputs the following code that SPECTECTOR reports as insecure.

Ex.	VISUAL C++				ICC				CLANG					
	UNP		FEN		UNP		FEN		UNP		FEN		SLH	
	-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. 9. Analysis of Kocher's examples [17] compiled with compilers and options. For each of the 15 examples, we analyzed the unpatched version (denoted by UNP), the version patched with speculation barriers (denoted by FEN), and the version patched using speculative load hardening (denoted by SLH). Programs have been compiled without optimizations (-O0) or with compiler optimizations (-O2) using the compilers CLANG, ICC, and VISUAL C++. ○ denotes that SPECTECTOR detects a speculative leak, whereas ● indicates that SPECTECTOR proves the program secure.

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

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

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

Example #15 follows a similar pattern (albeit with -O2 and -O0 exchanged). The interested reader is referred to Appendix D-A1 for more information.

C. *Performance*

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

The SPECTECTOR implementation terminates within less than 30 seconds on all of our examples, except for Example #5 in modes SLH -O0 and SLH -O2 for CLANG and mode FEN -O0 for ICC (with several examples being analyzed in about 0.1 seconds). In the exceptional cases, SPECTECTOR consumes 1-2 minutes for proving security. After manually inspecting the code, we believe this is due to the path explosion caused by Example #5's complex control-flow, which may lead to loops involving several branch instructions.

VIII. DISCUSSION

A. *Exploitability*

In practice, exploiting speculative execution attacks require an adversary to (1) trigger mis-speculation for a sufficiently large time window, (2) encode information into the microarchitectural state, and (3) decode the leaked information afterwards. SPECTECTOR is exclusively concerned with identifying code that enables (2). In particular, following the terminology of [25], speculative non-interference is a semantic characterization of those *disclosure gadgets* that encode more information than what the standard semantics allows.

For minimizing the placement of performance-degrading countermeasures, compilers try to account also for criteria (1) and (3). Integrating these criteria into a semantic definition of security is a challenge for future work.

B. Scope of Model

The results obtained by SPECTECTOR are only valid to the extent that the speculative semantics in conjunction with the "constant-time" observer model accurately captures the additional leakage induced by speculative execution.

In particular, SPECTECTOR may incorrectly classify a program as secure if the speculative semantics does not implicitly[7] capture all the additional observations an attacker may make due to speculative execution on an actual microarchitecture. For example, it would in principle be conceivable for microarchitectures to speculate on the value of the condition of a conditional update, which our speculative semantics currently does not permit.

Similarly, a secure program could be classified as insecure if the speculative semantics admits speculative executions that are not actually possible on an actual microarchitecture. This might be the case under speculative load hardening in Paul Kocher's Examples 10 and 15, as discussed in Section VII.

We note, however, that the speculative semantics can always be adapted to more accurately reflect reality, once better documentation of processor behavior becomes available. The notion of speculative non-interference 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 "constant-time" observer of the program execution that sees the location of memory accesses and the location of jump targets. This observer could be replaced by a weaker one, taking into account more detailed models of a processor's memory hierarchy, and SPECTECTOR could be adapted accordingly. 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 [14], [15].

C. Limitations of Automation

Complexity. SPECTECTOR symbolically executes programs to derive all possible symbolic traces. Whenever the program under analysis contains large, complex execution paths, SPECTECTOR may not terminate due to the path explosion problem that affects symbolic execution. Another source of computational complexity is checking the satisfiability of symbolic expressions. Even though we restrict ourselves to the decidable theories of bitvectors and arrays, determining the satisfiability of symbolic expressions may still be computationally challenging (or lead to non-termination). Nevertheless, SPECTECTOR is able to reason about all the 210 programs in our case studies. Additionally, the implementation can be easily extended to work with specialized theories and alternative solvers.

Prediction oracles. SPECTECTOR currently supports only the "always mispredict" oracle, which always mispredicts the outcome of branch instructions. This oracle has been implicitly used in prior work [13] since it is often the worst-case scenario in terms of leakage (it always speculatively execute the wrong branch before backtracking and executing the correct one). We remark that the prediction oracle is a key component of the security analysis, as it affects which instructions are speculatively executed. Hence, changing the prediction oracle may modify SPECTECTOR's results.

IX. RELATED WORK

Speculative execution attacks. These attacks exploit the effects of speculatively executed instructions to leak information. After SPECTRE [2], [4], [6], a number of speculative execution attacks have been discovered that differ in the exploited speculation sources [1], [5], [18], the covert channels [26], [3], [27] used, or the target platforms [28]. We refer the reader to [29] for a survey of speculative execution attacks and their countermeasures.

Here, we overview only SPECTRE v1 software-level countermeasures. AMD and Intel suggested inserting **lfence** instructions after branches [7], [30]. These instructions effectively act as speculation barriers, and prevent speculation leaks. The Intel C++ compiler [10], the Microsoft Visual C++ compiler [11], and CLANG [12] can automatically inject this countermeasure at compile time. An alternative technique that has been proposed is to introduce artificial data dependencies [31], [9]. Speculative Load Hardening (SLH) [9], implemented in the CLANG compiler [12], 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 [32]. This countermeasure cannot be automatically applied to arbitrary programs.

Detecting speculative leaks. oo7 [13] is a binary analysis tool for detecting speculative leaks. The tool looks for specific vulnerable code patterns. Hence, it misses some speculative leaks, like Example 4 from Section VII. 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 our semantic notion of security against speculative attacks. As a result, it can detect speculative leaks or prove their absence.

Disselkoen et al. [33] present an execution model supporting speculation and use it to model SPECTRE attacks. They neither provide a security notion nor a detection technique.

Formal architecture models. Armstrong et al. [34] present formal models for the ARMv8-A, RISC-V, MIPS, and CHERI-

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

MIPS instruction-set architectures. Degenbaev [35] and Goel et al. [36] develop formal models for parts of the x86 architecture. Such models enable e.g. 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.

Static detection of side-channel vulnerabilities. A number of approaches have been proposed for statically detecting side-channel vulnerabilities in programs [37], [38], [39]. 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.

X. 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, Matt Miller, and Matthew Parkinson for helpful discussions. This work was supported by a grant from the Intel Corporation, Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2015-70713-R DEDETIS and TIN2015-67522-C3-1-R TRACES, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

[1] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[3] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," 2018.

[4] G. Maisuradze and C. Rossow, "Speculose: Analyzing the security implications of speculative execution in CPUs," 2018.

[5] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[6] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018.

[7] 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.

[8] "What spectre and meltdown mean for webkit," https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.

[9] C. Carruth, "Speculative load hardening," 2018.

[10] 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.

[11] A. Pardoe, "Spectre mitigations in msvc," https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[12] "rl336990 - [slh] introduce a new pass to do speculative load hardening to mitigate spectre variant #1 for x86," https://reviews.llvm.org/rL336990, 2018.

[13] 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. [Online]. Available: http://arxiv.org/abs/1807.05843

[14] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2005, pp. 156–168.

[15] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium*, 2016.

[16] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

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

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

[19] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

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

[21] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE S&P*, 2007.

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

[23] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, 2004, pp. 100–114.

[24] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *TPLP*, vol. 12, no. 1–2, pp. 219–252, 2012, http://arxiv.org/abs/1102.5497.

[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. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," 2018.

[27] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," 2018.

[28] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre attacks: Stealing intel secrets from SGX enclaves via speculative execution," 2018.

[29] 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," *ArXiv e-prints*, Nov. 2018.

[30] 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.

[31] 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. [Online]. Available: http://arxiv.org/abs/1805.08506

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

[33] "Code that never ran: modeling attacks on speculative evaluation," https://github.com/chicago-relaxed-memory/spec-eval, 2018.

[34] 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," in *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, vol. 3, no. POPL. New York, NY, USA: ACM, January 2019, pp. 71:1–71:31.

[35] U. Degenbaev, "Formal specification of the x86 instruction set architecture," Ph.D. dissertation, 2012.

[36] S. Goel, W. A. Hunt, and M. Kaufmann, *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. Cham: Springer International Publishing, 2017, pp. 173–209. [Online]. Available: https://doi.org/10.1007/978-3-319-48628-4_8

[37] 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, p. 4, 2015.

[38] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "CaSym: Cache aware symbolic execution for side channel detection and mitigation," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[39] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

Appendix A
Non-speculative semantics

Given a μASM program $p \in \mathit{Prg}$, we formalise its semantics using the evaluation relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Obs} \times \mathit{Conf}$ defined in Figure 10.

Appendix B
Speculative and non-speculative 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 produced by the speculative semantics, 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 $\textbf{loadO}\ se$, $\textbf{storeO}\ se$, $\textbf{pcO}\ n$, or $\textbf{symPc}(se)$, $(\textbf{start}\ i \cdot \tau)\restriction_{nse} = \tau\restriction_{nse}$ if $\textbf{rollback}\ i$ is not in τ, $(\textbf{commit}\ i \cdot \tau)\restriction_{nse} = \tau\restriction_{nse}$, $(\textbf{start}\ i \cdot \tau \cdot \textbf{rollback}\ i \cdot \tau')\restriction_{nse} = \tau'\restriction_{nse}$, 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 $\textbf{loadO}\ se$, $\textbf{storeO}\ se$, $\textbf{pcO}\ n$, or $\textbf{symPc}(se)$, $(\textbf{start}\ i \cdot \tau)\restriction_{se} = \tau\restriction_{se}$ if $\textbf{rollback}\ i$ is not in τ, $(\textbf{commit}\ i \cdot \tau)\restriction_{se} = \tau\restriction_{se}$, $(\textbf{start}\ i \cdot \tau \cdot \textbf{rollback}\ i \cdot \tau')\restriction_{se} = \mathit{filter}(\tau) \cdot \tau'\restriction_{nse}$, and $\tau\restriction_{nse} = \varepsilon$ otherwise, where $\mathit{filter}(\tau)$ denotes the trace obtained by dropping all extended observations $\textbf{start}\ id$, $\textbf{commit}\ id$, and $\textbf{rollback}\ id$ from τ.

Appendix C
Symbolic semantics

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

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

Symbolic non-speculative semantics. We formalize the symbolic non-speculative semantics for a μASM program p using the relation $\rightarrow_s$ defined in Figure 11.

Appendix D
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 (line 4 below), which is a source of speculation. Hence, the resulting program is vulnerable to SPECTRE-style attacks, and SPECTECTOR correctly detects the possible leak of sensitive information.

```
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 contrast, when we compiled in the UNP -O2 mode, the conditional operator is translated as a conditional move operation (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
```

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

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

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

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

$$\textsc{ConditionalUpdate-2}\;\frac{p(a(\mathbf{pc})) = x \xleftarrow{e'} e \qquad [\![e']\!](a) \neq 0 \qquad x \neq \mathbf{pc}}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle} \qquad \textsc{Terminate}\;\frac{p(a(\mathbf{pc})) = \bot}{\langle m, a\rangle \to \langle m, a[\mathbf{pc} \mapsto \bot]\rangle}$$

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

$$\textsc{Beqz-1}\;\frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad a(x) = 0}{\langle m, a\rangle \xrightarrow{\mathbf{pcO}\ n} \langle m, a[\mathbf{pc} \mapsto n]\rangle} \qquad \textsc{Beqz-2}\;\frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad a(x) \neq 0}{\langle m, a\rangle \xrightarrow{\mathbf{pcO}\ a(\mathbf{pc})+1} \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle}$$

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

Fig. 10. μASM semantics for a program p

```
5    lea     1(%rdx), %rax
6    cmova   %rax, %rcx
7    mov     A(%rcx), %rax
8    shl     $9, %rax
9    mov     B(%rax), %rax
10   and     %rax, temp
```

1) Example #15 in SLH *mode:* In this example, the attacker provides the input via the pointer `*y`:

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

When this example is compiled in the `-O0` SLH mode, CLANG does not correctly secure the resulting program. 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 attacker can exploit the second memory access to speculatively leak the content of `A[0xFF...FF]`. In our experiments, SPECTECTOR correctly detected such speculative 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 the result of the memory access `A[*y]` (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
```

Expression evaluation

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

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

CONDITIONALUPDATE-CONCR-2
$$\frac{\begin{array}{c}p(sa(\mathbf{pc})) = x \xleftarrow{e'} e \qquad [\![e']\!](sa) = n \qquad n \in \mathit{Vals}\\ n \neq 0 \qquad x \neq \mathbf{pc}\end{array}}{\langle sm, sa\rangle \to_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1]\rangle}$$

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-CONCR
$$\frac{\begin{array}{c}p(sa(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc}\\ n = [\![e]\!](sa) \qquad n \in \mathit{Vals}\end{array}}{\langle sm, sa\rangle \xrightarrow{\mathbf{loadO}\ n}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1, x \mapsto sm(n)]\rangle}$$

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

STORE-CONCR
$$\frac{p(sa(\mathbf{pc})) = \mathbf{store}\ x, e \qquad n = [\![e]\!](sa) \qquad n \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\mathbf{storeO}\ n}_s \langle sm[n \mapsto sa(x)], sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1]\rangle}$$

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

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

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

BEQZ-CONCR-2
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad sa(x) \neq 0 \qquad sa(x) \in \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\mathbf{symPc}(\top)\cdot\mathbf{pcO}\ sa(\mathbf{pc})+1}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1]\rangle}$$

BEQZ-SYMB-2
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, n \qquad sa(x) \notin \mathit{Vals}}{\langle sm, sa\rangle \xrightarrow{\mathbf{symPc}(sa(x)\neq 0)\cdot\mathbf{pcO}\ sa(\mathbf{pc})+1}_s \langle sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1]\rangle}$$

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

JMP-SYMB-1
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad [\![e]\!](sa) \notin \mathit{Vals} \qquad n \in \{n' \in \mathit{Vals} \mid p(n') \neq \bot\}}{\langle sm, sa\rangle \xrightarrow{\mathbf{symPc}([\![e]\!](sa)=n)\cdot\mathbf{pcO}\ n} \langle sm, sa[\mathbf{pc} \mapsto n]\rangle}$$

JMP-SYMB-2
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad [\![e]\!](sa) \notin \mathit{Vals} \qquad se = \bigwedge_{n \in \{n' \in \mathit{Vals} \mid p(n') \neq \bot\}} [\![e]\!](sa) \neq n}{\langle sm, sa\rangle \xrightarrow{\mathbf{symPc}(se)\cdot\mathbf{pcO}\ \bot} \langle sm, sa[\mathbf{pc} \mapsto \bot]\rangle}$$

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