Efficient Information-Flow Verification under Speculative Execution

Roderick Bloem[1], Swen Jacobs[2], and Yakir Vizel[3]

[1] Graz University of Technology, Austria
roderick.bloem@iaik.tugraz.at
[2] CISPA Helmholtz Center for Information Security, Germany
jacobs@cispa.saarland
[3] Technion, Israel
yvizel@cs.technion.ac.il

Abstract. We study the formal verification of information-flow properties in the presence of speculative execution and side-channels. First, we present a formal model of speculative execution semantics. This model can be parameterized by the depth of speculative execution and is amenable to a range of verification techniques. Second, we introduce a novel notion of information leakage under speculation, which is parameterized by the information that is available to an attacker through side-channels. Finally, we present one verification technique that uses our formalism and can be used to detect information leaks under speculation through cache side-channels, and can decide whether these are only possible under speculative execution. We implemented an instance of this verification technique that combines taint analysis and safety model checking. We evaluated this approach on a range of examples that have been proposed as benchmarks for mitigations of the Spectre vulnerability, and show that our approach correctly identifies all information leaks.

Keywords: Verification, Information Flow, Speculative Execution, Side Channels

1 Introduction

The Spectre attacks have shown how speculative execution in modern CPUs can lead to information leaks via side channels such as shared caches, even if the program under consideration is secure under a standard execution model [22]. Since speculative execution is an essential optimization for the performance of modern processors, the underlying vulnerability affects the vast majority of all processors in use today, and is not easy to fix without sacrificing a lot of the performance gains of recent years. The general assumption is that processors will remain vulnerable to Spectre for the foreseeable future, and therefore security can only be ensured on the software level, carefully taking into account the vulnerability of the hardware. Based on this observation, we develop a method to detect whether a given program can potentially leak sensitive information due to the interplay of speculative execution and side-channel attacks, or prove that information-flow security properties hold despite these factors.

The Spectre vulnerability is exploited by (i) influencing the predictions that lead to the speculative execution of instructions, (ii) by feeding operands to these instructions that lead to secret data being stored or otherwise reflected in the microarchitectural state of the CPU, and (iii) observing these changes to the microarchitectural state through a side-channel. To achieve (i), the attacker can manipulate the *Pattern History Table* that predicts the choice of conditional branches [20, 22], the *Branch Target Buffer* that predicts branch destination addresses [22], or the *Return Stack Buffer* that predicts return addresses [24]. To achieve points (ii) and (iii), the attacker can target different features of the microarchitecture to be observed through a side-channel, for example port contention [7] or, most commonly, caches [22].

In this paper, we provide a formal model for programs under a speculative execution semantics, and a formal notion of information-flow security that takes into account speculative execution and side-channels. Both the system model and the notion of security are designed to remain as close as possible to existing formalisms, in order to alleviate the adaptation of existing verification approaches to these new concepts. At the same time, they are sufficiently general and flexible to cover a wide range of different types of speculation and side-channels. Based on our new formalisms, we provide an algorithm that detects potential information leaks under speculative execution with cache side-channels, and demonstrate its capabilities on a benchmark set designed to test software for Spectre vulnerabilities [21].

1.1 An Example Problem and Our Solution

The program in Figure 1 is a very basic example for a Spectre-style information leak.[4] Function main receives an input idx from the user. It checks whether the value of idx corresponds to an entry in array1, and if so, uses the value of array1[idx] to determine a position to read from in array2. The value of array2 at that position is then stored in a temporary variable temp.

```
int main(int argn, char* args[]) {
    int temp = 0;

    int idx = getc();
    if (idx < array1_size)
        temp = array2[array1[idx]*512];

    return 0;
}
```

Fig. 1: Vulnerability example

[4] It is, in fact, the basic example used in the Spectre paper [22].

Under standard execution semantics, the piece of code is harmless: since the condition on idx ensures that the access to array1 is within bounds, an attacker (that controls input idx) cannot obtain any information that is not in array1 or in the positions of array2 referenced by values in array1.

Under speculative execution, however, the command in line 6 can be executed before the condition in line 5 is evaluated. This is not a problem regarding the information available at the program level, since the value of temp will only be written into memory *after* it is determined that the speculation was correct. However, the read from array2 will load data into the cache at an address that is determined by the value in array1[idx], thus changing the microarchitectural state *during* speculation. Since the bounds check has not yet been evaluated, the access to array1 can read an arbitrary byte from memory, the value of which can leak to the attacker through a timing attack on the shared cache.

We provide a fully automatic method to detect such potential information leaks. To check whether the program above has potential information leaks under speculative execution, our implementation compiles the source code to LLVM intermediate code, which is then interpreted according to a non-standard speculative execution semantics to obtain a formal representation as a transition system. In its simplest form, the non-standard semantics allows us to ignore conditional statements, as long as we set a flag that signals that we are now operating under speculation. The resulting transition system is also equipped with a leakage model that represents the attacker's possible observations. In our example, the address of any memory access will be leaked to the attacker, modeling their capability to obtain this information through a side-channel attack. On this transition system, we then use a combination of taint analysis and safety model checking to determine whether there are information leaks, and whether they are only possible under speculative execution.

Our implementation correctly determines that there is an information leak in this example. Moreover, we support the insertion of a special stop command into the code that will halt speculative execution, as is done by insertion of serializing instructions such as LFENCE as an existing mitigation of Spectre. For the modified code with a stop added after line 5, our implementation correctly determines that no information leak is possible.

1.2 Contributions

The contributions of this work can be summarized as follows:

1. We provide a formal model for programs under a speculative execution semantics. Programs are modeled as transition systems, where speculation is added as an additional dimension. Moreover, the model supports *speculation barriers*, which can be used to explicitly stop speculative execution in branches that are deemed to handle sensitive information. Since this model is a very natural generalization of a standard operational semantics, it is easy to modify existing verification approaches to handle our model. Moreover, the model is based on a very general notion of speculation and is flexible

with regards to the information that is leaked to an attacker. Therefore, it covers a wide range of security vulnerabilities that are due to speculation and side-channel attacks.
2. We provide a formal notion of information-flow security under speculative execution. Like our system model, it is a natural generalization of existing formalisms, which means that existing approaches to security verification can, at least in theory, be easily adapted to reason about our new notion of security.
3. As an application of our formalisms, we present an algorithm that translates an input program into a transition system according to its speculative execution semantics, based on a leakage model for cache side-channels. The transition system is analyzed using a combination of taint analysis and safety verification that explicitly checks for operations that happen under speculation. This analysis finds potential information leaks that are due to speculation.
4. Finally, we evaluate our algorithm on a set of benchmarks that has been designed to test whether existing mitigations in compilers are effective in preventing Spectre leaks [21]. We show that it correctly detects Spectre-style information leaks in all cases, and correctly proves the absence of leaks after the manual insertion of speculation barriers.

1.3 Related Work

Our work is inspired by the Spectre attacks [22], which combine manipulation of speculative execution and side-channel attacks to read private data. There are several variants of the idea (summarized in [9]), and the approach we present in this paper can detect (potential) vulnerabilities of a program against Spectre variants V1 [22], V1.1 and V1.2 [20], as well as V4 [18]. Moreover, our model is sufficiently flexible to reason about leakage through a port contention side channel [7] and possibly other side channels, as long as observations that are possible through the side channel can be specified.

Several mitigations against Spectre have been proposed and partly implemented in compilers [17, 25], but without any formal guarantee of security, and in some cases already demonstrated to be insufficient in general [21]. Taram et al. [30] introduced a method that introduces fences dynamically. They show that this can greatly reduce the overhead compared to static conservative fencing, but also do not give formal correctness guarantees.

There are several other variants of Spectre that rely on other features of the microarchitecture, such as branch target buffers or return stack buffers that are shared between users [22, 24], or lazy context switching between users for FPU operations [29]. To support these, our model needs to be extended to a programming language with indirect jumps or return instructions, which is out of the scope of this paper.

Formal models for side-channel attacks and speculative execution. A significant number of works have considered side-channel attacks in a formal framework. One approach is to verify that a (cryptographic) program satisfies the *constant-time security* paradigm, which implies resistance against timing-based side-channel attacks. The applied reasoning methods range from deductive verification [3] and static analysis [27] to security type systems [1]. Almeida et al. [2] give a formal semantics for programs together with a flexible specification of side-channels similar to ours, and use an approach based on model-checking to verify information-flow security. While all of these approaches support verification of side-channel security in some form, none of them supports security under speculative execution, which is essential for our approach.

Correctness under speculative execution has been considered (i) at the level of the processor, i.e., aiming to verify that a given processor model with speculative execution satisfies some correctness criterion, and (ii) at the software level, based on a form of non-standard semantics of programs. Regarding (i), we note that to the best of our knowledge all of these works [16, 19, 23, 28, 32] only consider functional correctness properties (that are insufficient to express information-flow security), and also do not support side-channel leakage. One exception to the above is the framework of Arons and Pnueli [4] that allows the user to define which features of the microarchitecure can be used in the specification, in theory allowing to take side-channels into consideration. Interest in (ii) has thus far been very limited, and has also been restricted to functional correctness properties [8].

To the best of our knowledge, there are only two approaches that handle both side-channel attacks and speculative execution in a more or less formal way, both developed concurrently with our own approach. First, Wang et al. [33] have developed a static analysis to detect Spectre and Meltdown vulnerabilities. In contrast to our work, their focus has been on finding a practical solution that is specific to this class of vulnerabilities, while our main goal was to define a general model of speculative execution semantics and a corresponding notion of information-flow security that can be the basis of a wide range of solutions for the problem class. Second, Guarnieri et al. [13] have introduced a general notion of speculative non-interference, which however uses a significantly more involved model of speculative execution semantics, and results in a security notion that requires to reason about knowledge that is derivable from attacker observations. In comparison, our model and security notion are much simpler and closer to existing formalisms.

2 Preliminaries

We formalize the basic problem under consideration: given a program P, verify whether P is secure against timing attacks.

Transition Systems. Let X be a set of variables that is used to describe the program state, and $X' = \{x' \mid x \in X\}$ a copy of X, in the following used to represent the post-state of a transition. A *transition system* is given as a tuple $M = \langle X, \mathit{Init}(X), \mathit{Tr}(X, X')\rangle$ where $\mathit{Init}(X)$ is a (first-order) formula over X representing the initial states, and $\mathit{Tr}(X, X')$ is a formula representing the transition relation.

A formula over the set of variables X is called a *state formula*. A formula $\varphi(X, X')$ such that for every valuation V of X there is exactly one valuation V' of X' with $\varphi(V, V')$ is called a *state update function*. We denote by $\mathsf{unchanged}(Y)$ the state update function $\bigwedge_{x \in Y} x' = x$ for some $Y \subseteq X$. A *state* is a valuation of all variables. For a state A and a variable $x \in X$, we denote by $A[x]$ the value of x in A, and by $A[Y]$ the valuation of a set of variables $Y \subset X$. An *execution* of a transition system is a sequence of states $\pi := A_0, A_1, \ldots, A_n$, such that $A_0 \Rightarrow \mathit{Init}$ and for $1 \leq i \leq n$: $(A_{i-1}, A_i) \Rightarrow \mathit{Tr}$.

Standard Program Semantics. A transition system for a program can be obtained based on its operational semantics (for a fully formalized description see e.g. [2]). Programs in simple programming languages can be translated into a transition system in a straightforward way by encoding each line of the program into either a conditional or unconditional state update formula. A *conditional state update formula* is of the form

$$\tau_i := \mathsf{pc} = i \rightarrow \mathit{cond}(X) \ ? \ \varphi(X, X') : \psi(X, X'),$$

while an *unconditional state update formula* is of the form

$$\tau_i := \mathsf{pc} = i \rightarrow \varphi(X, X').$$

In both cases $i \in \mathbb{N}$ is the line of the program to be encoded, pc is a special program variable (the "program counter"), $\mathit{cond}(X)$ is a condition on the set of variables X, and $\varphi(X, X')$ and $\psi(X, X')$ are state update functions that should be executed when the condition is either true, or false, respectively. We refer to instructions as either conditional or unconditional depending on their corresponding state update formula. Moreover, the set of conditional instructions is denoted by $C \subseteq \mathbb{N}$, s.t. if $i \in C$, then τ_i is a conditional state update formula.

In addition, we consider a programming language that includes the special annotation $\mathsf{assume}(\mathit{cond}(X))$, which is encoded by

$$\tau_i := \mathsf{pc} = i \rightarrow \mathit{cond}(X) \ ? \ (\mathsf{pc}' = \mathsf{pc} + 1 \wedge \mathsf{unchanged}(X \setminus \{\mathsf{pc}\}) : \mathsf{false}.$$

This special annotation requires that at line i the condition $\mathit{cond}(X)$ holds.[5] By conjoining the formulas for all lines of the program, one obtains a symbolic representation of the transition relation. Namely, $\mathit{Tr} := \bigwedge_i \tau_i$.

We will discuss in Section 3 how to obtain a transition system for the program under speculative semantics.

Safety Verification. Given a transition system M and a formula $\mathit{Bad}(X)$, M is *unsafe* w.r.t. Bad when there exists a run $\pi := A_0, A_1, \ldots, A_n$ of M s.t. $A_n \Rightarrow \mathit{Bad}$. Such a run π is called a *counterexample* (CEX). If no such π exists, M is *safe* w.r.t. Bad. The safety verification problem determines if M is *safe* or *unsafe* w.r.t. Bad.

[5] If it does not hold, then the transition relation will evaluate to false for all post-states, i.e., the program halts.

2.1 Information Flow Analysis

Many security properties can be cast as the problem of verifying secure information flow. Namely, by proving that confidential data does not flow to non-confidential outputs (i.e. is not observable) during the execution of a system [12]. More formally, assume that $H \subset X$ is a set of *high-security* (or *secret*) variables and $L := X \setminus H$ is the set of *low-security* (or *public*) variables. Moreover, let $L = L_i \uplus L_o$, where L_i are *input variables* that are controlled by the attacker, and for each *output variable* $x \in L_o$, let $O_x(X)$ be a predicate that determines when x is observable. The set of output variables L_o and their observation predicates O_x depend on the threat model being considered.

Example 1. In the example from Figure 1, we have $L_i = \{\mathsf{argn}, \mathsf{args}, \mathsf{idx}\}$, since these are inputs that can be chosen by the attacker. Moreover, we have $L_o = \{x\}$ for an auxiliary variable x that, upon every array read a[i] is updated to the value of i, and is observable whenever it has been updated.[6] This models an attacker that uses a cache side-channel attack.

The *information flow problem* asks whether there exists a run of M such that the value of variables in H affects the value of a variable $x \in L_o$ in a state where $O_x(X)$ holds. Intuitively, this means that variable x "leaks" secret information.

Definition 1. *Let M be a transition system and let $H \subset X$ be a set of high-security variables and $L := X \setminus H$ a set of low-security variables with $L := L_i \uplus L_o$. M* leaks secret information, *denoted by $H \rightsquigarrow_M L$, iff there exists two executions $\pi^1 = A_0^1, \ldots, A_n^1$ and $\pi^2 = A_0^2, \ldots, A_n^2$ of M s.t. the following formulas hold:*

$$\forall 0 \leq i \leq n, x \in L_i \cdot A_i^1[x] = A_i^2[x]$$
$$\exists 0 \leq i \leq n, x \in L_o \cdot (A_i^1 \Rightarrow O_x) \wedge (A_i^2 \Rightarrow O_x) \wedge A_i^1[x] \neq A_i^2[x]$$

When M does not leak secret information, we write $H \not\rightsquigarrow_M L$. Moreover, when M is clear from the context, we write $H \rightsquigarrow L$ (and respectively, $H \not\rightsquigarrow L$). Note that our definition can be instantiated to match a standard notion of non-interference that requires two executions with equal low-security values in the initial state to result in equal low-security values in the final state: assume that variables in L_i are not changed during execution (otherwise generate a copy of the variable in L_o), and let all variables in L_o only be observable in the final state.

There are two standard techniques that can be used to perform information flow analysis: *taint analysis* and *self-composition*.

Taint Analysis. Taint analysis instruments a program with taint variables and taint tracking code, which is used to simulate the flow of confidential data to public outputs. It operates by marking high-security variables with a "taint" and checking if this taint can propagate to low-security variables. It can be performed statically by analyzing the program's code [15, 26] or symbolically when

[6] We assume that the compiler separates nested array reads into two separate reads.

described as a safety verification problem [34]. While taint analysis is efficient, it is imprecise since it over-approximates the possible leaks of the program.

Self-Composition. In the most common use-case of safety verification, the goal is to prove a safety property that can be checked on a *single* run, such as functional correctness. Secure information flow, however, is a property that involves a comparison of two executions of the system, i.e., it is a *hyper-property* [11]. One approach to handling hyper-properties is self-composition [6], where the program P is composed with one or more copies of itself. This composition results in a new program P' such that a single run of P' represents several runs of P in parallel. By that, reasoning about hyper-properties is reduced to reasoning about properties of individual runs of the self-composed program P'. This allows to reason about hyper-properties using standard software model checkers [10, 14]. However, in many cases, this approach does not scale to real-life programs [31].

3 A Formal Model for Speculative Execution

Speculative execution is an optimization technique for concurrent execution inside CPUs that reduces the idling time of a CPU by executing code *speculatively*, i.e., without knowing whether the given part of the program will actually be reached. To this end, it makes assumptions on branching conditions that may or may not turn out to be true in the future. Speculative execution is one of the cornerstones of modern out-of-order execution, and is to a large extent responsible for the dramatic performance improvement of CPUs in recent years.

We are interested in proving information-flow properties of programs that are executed in such an environment. To this end, we need a formal execution model that takes speculative execution into account. In the following, we introduce two variants of such a model and explain how to transform a given program into its formal representation.

3.1 Simple Speculative Execution Semantics

We explain the idea of a simple speculative execution semantics on an example.

Example 2. Consider the two programs that appear in Figure 2. The program to the left (Figure 2a) represents the standard execution semantics where the array is updated at index i only if i is in the range, and otherwise location 0 is updated. The program to the right (Figure 2b), however, behaves differently. The condition that guards the array update is replaced with a non-deterministic choice, and either the `then` or `else` branch is taken. Moreover, the execution of a branch depends on both the condition and the possibility of speculative execution. This behavior is forced by the assumptions added in each branch. Note that the possibility of speculative execution is captured by a non-deterministic Boolean variable `spec`.

```
int insert(int[] A, int len) {
    int v=f(), i=g();
    assume (i >= 0);

    if (i < len) {

       A[i] = v;
    } else {

       A[0] = v;
    }
    return v;
}
```

(a) Standard execution

```
int insert(int[] A, int len) {
    int v=f(), i=g();
    assume (i >= 0);
    bool spec = *;
    if (*) {
       assume(i < len ^ spec);
       A[i] = v;
    } else {
       assume(i >= len ^ spec);
       A[0] = v;
    }
    return v;
}
```

(b) Speculative execution

Fig. 2: Execution semantics example

Translation to Simple Speculative Execution Semantics. We now formalize the above example. Let $M = \langle X, \mathit{Init}, \mathit{Tr}\rangle$ be the transition system for a program P according to its standard program semantics. To account for speculative execution we define a new transition system $\hat{M} = \langle X \cup \{\mathsf{spec}\}, \mathit{Init}, \hat{\mathit{Tr}}\rangle$, where spec is a new Boolean variable. spec is initialized non-deterministically and its post-state is defined by

$$\tau_i^s := \mathsf{pc} = i \to (\neg\mathsf{spec} \land i \in C) \ ?\ \mathsf{spec}' = * :\ \mathsf{spec}' = \mathsf{spec}.$$

Informally, this means that speculative execution can start whenever a conditional instruction executes, but then it will remain enabled for the rest of the program execution.

The effect of speculative execution is encoded by replacing every conditional instruction τ_i with:

$$\begin{aligned}\hat{\tau}_i := {} & \mathsf{pc} = i \to * \ ? \\ & (\neg\mathsf{spec} \Rightarrow (\mathit{cond}(X) \veebar \mathsf{spec}')) \land \varphi(X, X') : \\ & (\neg\mathsf{spec} \Rightarrow (\neg\mathit{cond}(X) \veebar \mathsf{spec}')) \land \psi(X, X')\end{aligned}$$

where $\veebar$ represents an *exclusive or*. Note that a conjunct is added to the then and else parts of the conditional instruction. The conjunct forces the value of spec to become true when a branch is executed speculatively. Namely, the branch executes even though its condition does not hold. We emphasize that the added conjunct is guarded by the current-state value of spec. This is needed for the case of nested branches. If spec is already true when executing the conditional instruction, then the conjunct holds. Hence, if the program is already in speculative execution mode, then either branch can be taken. However, if spec is false, then the exclusive or ensures that either the guard (e.g. a condition of an if statement) for that branch holds, or the branch is executed speculatively (by forcing spec to become true), but both cannot happen simultaneously. This captures our intention of only detecting information leaks caused by misprediction.

```
int insert(int[] A, int len) {
    int v=f(), i=g();
    assume (i >= 0);

    if (i < len) {

       A[i] = v;

    } else {

       A[0] = v;

    }
    return v;
}
```

(a) Standard execution

```
int insert(int[] A, int len) {
    int v=f(), i=g();
    assume (i >= 0);
    int spec = *;
    if (*) {
       assume(i < len ^ 0 < spec < sigma);
       A[i] = v;
       if spec > 0 then spec += w();
    } else {
       assume(i >= len ^ 0 < spec < sigma);
       A[0] = v;
       if spec > 0 then spec += w();
    }
    return v;
}
```

(b) Bounded speculative execution

Fig. 3: Execution semantics with bound example

Unconditional instructions are kept unchanged, namely $\hat{\tau}_i := \tau_i$. The transition relation is then defined as $\hat{Tr} := (\bigwedge_i \hat{\tau}_i \wedge \tau_i^s)$.

In addition to the standard constructs of a simple language as sketched above, we assume a special command stop that acts as a barrier for stopping speculative execution. The stop command is defined by

$$\tau_i := \mathsf{pc} = i \rightarrow \neg\mathsf{spec}\ ?\ (\mathsf{pc}' = \mathsf{pc} + 1 \wedge \mathsf{unchanged}(X \setminus \{\mathsf{pc}\}) : \mathsf{false},$$

which is the same as $\mathsf{assume}(\neg\mathsf{spec})$.

3.2 An Architecture-Dependent Speculative Execution Semantics

In order to execute instructions speculatively, a recovery mechanism for undoing speculated instructions on miss-predicted branches or on exceptions is required. This "rollback" mechanism is implemented using a *reorder buffer* (ROB), which stores the results of instructions that are executed speculatively. When a miss-prediction is discovered, the ROB is flushed, otherwise a *commit* is performed. Note that the size of the ROB is fixed, hence it implies a bound on the number of instructions that can be executed speculatively.

In order to take this mechanism, of either rollback or commit, into account and make our analysis more precise, we introduce an advanced model where the system has a *bound* on the maximal depth of speculative execution (i.e., the number of steps that can be taken before execution under speculation is halted). We model this bound as a value $\sigma \in \mathbb{N} \cup \{\infty\}$, called the *speculative execution parameter.*

Example 3. The programs in Figure 3 show the difference between standard semantics and bounded speculative execution semantics. As before, the left-hand side (Figure 3a) represents the standard execution semantics. The right-hand side (Figure 3b) again replaces the condition for the array update with a non-deterministic choice, but now the execution contains a condition on not only whether speculation has been enabled before, but also that the number of steps under speculation is below our bound σ. Note that the type of variable spec has changed from a Boolean to an integer. It is also important to note that spec is incremented by some value if it is enabled. The reason for this arbitrary value (represented by $\mathsf{w()}$ in our example) is that every instruction is translated into a different number of micro-ops. Hence, it occupies a different number of slots in the ROB.

Translation to Architecture-Dependent Speculative Execution Semantics. To formalize the idea, let again $M = \langle X, \mathit{Init}, \mathit{Tr}\rangle$ be a transition system for a program P. To model bounded speculative execution, we define a new speculative transition system $\check{M} = \langle X \cup \{\mathsf{spec}\}, \mathit{Init}, \check{\mathit{Tr}}\rangle$, where spec is a new variable of type $\mathbb{N}$. spec is initialized non-deterministically to either 0 or 1 and its post-state is defined by

$$\begin{aligned}\check{\tau}_i^s := \mathsf{pc} = i \rightarrow \mathsf{spec}' = (&\mathsf{spec} > 0 \ ? \ \mathsf{spec} + w(i) \ : (i \in C \ ? \ \{0, w(i)\} : 0)) \\ &\wedge \mathsf{spec}' < \sigma\end{aligned}$$

where $\{0, w(i)\}$ is interpreted as non-deterministic choice between integers 0 and $w(i)$, and $w(i)$ represents the number of slots occupied in the ROB by instruction i. Informally, this means that speculative execution can start whenever a conditional instruction executes, but then it will remain enabled until we reach the speculation bound σ.

The effect of speculative execution is encoded by replacing every conditional instruction τ_i by:

$$\begin{aligned}\check{\tau}_i := \mathsf{pc} = i \rightarrow * \ &? \ (\mathsf{spec} = 0 \Rightarrow (\mathit{cond}(X) \veebar \mathsf{spec}' = w(i)) \wedge \varphi(X, X') \\ &: (\mathsf{spec} = 0 \Rightarrow (\neg\mathit{cond}(X) \veebar \mathsf{spec}' = w(i)) \wedge \psi(X, X')\end{aligned}$$

and keeping unconditional instructions unchanged, namely $\check{\tau}_i := \tau_i$. The transition relation is defined by $\check{\mathit{Tr}} := (\bigwedge_i \check{\tau}_i \wedge \check{\tau}_i^s)$. Similarly, we support a barrier command stop, which is in this case is defined by

$$\tau_i := \mathsf{pc} = i \rightarrow \mathsf{spec} = 0 \ ? \ (\mathsf{pc}' = \mathsf{pc} + 1 \wedge \mathsf{unchanged}(X \setminus \{\mathsf{pc}\}) : \mathsf{false},$$

equivalent to $\mathsf{assume}(\mathsf{spec} = 0)$.

4 Information-Flow Security under Speculative Execution

Our formal model for speculative execution represents the program as a transition system, which makes it amenable to many standard verification techniques. Moreover, our notion information-flow security (Definition 1) has been chosen with speculative execution in mind: since changes to the program state (in memory) based on speculative computations are only applied when it has been established that speculation was correct, the standard notion of non- interference is not affected by speculation at all. Therefore, we have to consider a notion that takes into account information about the microarchitectural state of the system that can be detected through side-channels. In the following, we generalize this notion of security to systems with speculative execution semantics.

Example 4. To motivate our definition, consider again the example in Figure 1. The program receives an index idx from the user, accesses array1 at location idx, and then array2, based on the value of array1. To prevent an illegal access to array1, the access is guarded by an if statement. However, during speculative execution, the arrays may be accessed even when the condition does not hold. While this access will not change the state of the program in memory, it may alter the state of the cache, and therefore may leak sensitive information through a cache side-channel. If our attacker model includes cache side-channels, then the set of variables L_o will contain a variable that is updated with every memory access a[i] to the address i, and is observable when the access happens.

Note that in order to take advantage of speculative execution, an attacker needs to be able to train the branch predictor, giving him some amount of control over speculative execution. As a conservative approximation, we assume that speculative execution is completely controlled by the attacker. Therefore, the new variable spec that indicates if speculative execution is performed in our model of speculative execution semantics should be controlled by the attacker. This very naturally leads us to the following definition of speculative information-flow:

Definition 2. *Let P be a program, and let $\check{M}$ be the corresponding speculative transition system of P. Let $H, L \subset X$ be the sets of high and low security variables in X with $L := L_i \uplus L_o$ and* spec $\in L_i$*. We say $\check{M}$* leaks information under speculative execution *if $H \rightsquigarrow_{\check{M}} L$.*

That is, we can reduce speculative information leaks to non-speculative information leaks by considering a model with speculative execution semantics and assuming that spec is under control of the attacker. Intuitively, if a path in P that is only enabled due to speculative execution leaks private data, then we can detect this by comparing runs of that path with different values for H but identical values for L_i, including identical speculation behavior. Note that if spec remains 0 (or false) for the whole run, then this notion of security coincides with Definition 1.

Since our notion of information-flow security under speculative execution is based on transition systems and is a natural generalization of existing notions, in theory it is not hard to extend existing approaches for verification of information flow such as self-composition or taint analysis to cover it. In practice however, the additional behaviors introduced by speculation will likely make an approach based on naive self-composition intractable. An implementation that handles non-trivial programs will require a specialized and more scalable solution.

4.1 Detecting Speculative Execution Leaks

We present an approach that uses the above formal model for detecting potential information leaks under speculative execution. More precisely, we are looking for *Spectre leaks*, i.e., leaks through memory accesses that can be influenced by the user, using a cache side channel. To this end, it tries to find memory accesses a[i] such that i can be chosen or influenced by the attacker, and checks whether these accesses can happen under speculative execution. A high-level description of our approach appears in Algorithm 1.

```
1  $\check{M} \leftarrow$ GetSpecTr($P, \sigma$)
2  $I \leftarrow$ AnalyzeInfFlow($\check{M}$)
3  while $I \neq \emptyset$ do
4      pick $i \in I$
5      if $i$ is memory access then
6          $res \leftarrow$ CheckSpeculation($\check{M}, i$)
7          if $res = \top$ then
8              return possible Spectre leak
9      $I \leftarrow I \setminus \{i\}$
10 return no Spectre leaks
```

Algorithm 1: Detecting Spectre Attacks

The algorithm receives as an input a program P to be analyzed, and a speculation bound σ. It then starts by creating a transition system $\check{M}$, which includes speculative execution semantics as described in Section 3 (line 1). Then, information flow analysis is performed to identify all instructions that may be affected by an attacker (line 2). All affected instructions are returned in I. This is done by tracking the effect low security variables have on high security variables. There are various ways to achieve this goal, such as *taint analysis* [26] and *self-composition* [5]. Since performing self-composition is often intractable, here we implemented a variant of the algorithm that uses taint analysis for this task.

Once I is computed, the algorithm then analyzes all memory accesses in I. For every such memory access, a safety verification problem is generated and verified (line 6). This is achieved by adding an assertion that checks wether speculative execution is enabled at the location of the memory access (i.e. assert($spec = 0$)). If any of these verification problems is unsafe and a counterexample is generated, then the algorithm concludes that the memory access can be executed speculatively. Moreover, this speculative execution can be manipulated by an attacker.

Theorem 1. *Algorithm 1 is sound.*

Proof idea. If Algorithm 1 returns “no Spectre leaks” for inputs P and σ, then in $\check{M}$ there is no memory access that can be influenced by the user and execute speculatively.

We emphasize that Algorithm 1 can prove the absence of Spectre-like attacks. However, it can not conclusively determine that such an attack exists. This is due to two reasons. First, identifying that a memory access can execute speculatively does not necessarily imply that it leaks secret information. Second, in the variant we implemented, we use taint analysis to compute such memory accesses that can be affected, and hence the set I is an over-approximation. Making the algorithm more precise is left as an avenue for future work.

5 Evaluation and Conclusion

We have implemented our algorithm on top of the SeaHorn verification framework [14]. Taint analysis and the instrumentation of speculative execution semantics are implemented on top of LLVM.

For our evaluation, we used the examples from Paul Kocher's blog post [21] that have been designed to test whether existing mitigations against Spectre are sufficient to prevent information leaks. Our implementation identifies all of these examples as unsafe (possible leak due to speculative execution). In addition, we created a few SAFE examples (based on those from [21]). On these examples as well, our implementation identified the programs as SAFE.

Due to the size of these examples the runtime was small (about 1s). We note that while these examples are only short code snippets, they cover a large range of cases in which speculation-based attacks may be possible.

To conclude, in this paper, we have introduced novel formalisms that allow us to reason about security vulnerabilities in the presence of speculative execution and side-channels, and give formal correctness guarantees by proving the absence of such vulnerabilities.

Our formalisms are designed to be as simple as possible, while still covering a wide range of Spectre-type attacks. They are also very natural generalizations of existing concepts, which makes it easy to extend existing verification algorithms to our more general setting, as demonstrated in our example algorithm based on taint analysis and safety verification.

As a future work, we aim at implementing other variants of our approach, which use our formalism and provide more precision and guarantees while remaining efficient.

Acknowledgements. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Agat, J.: Transforming out timing leaks. In: POPL. pp. 40–53. ACM (2000). https://doi.org/10.1145/325694.325702
2. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security. pp. 53–70. USENIX Association (2016), `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida`
3. Almeida, J.B., Barbosa, M., Pinto, J.S., Vieira, B.: Formal verification of side-channel countermeasures using self-composition. Sci. Comput. Program. **78**(7), 796–812 (2013). https://doi.org/10.1016/j.scico.2011.10.008

4. Arons, T., Pnueli, A.: A comparison of two verification methods for speculative instruction execution. In: TACAS. Lecture Notes in Computer Science, vol. 1785, pp. 487–502. Springer (2000). https://doi.org/10.1007/3-540-46419-0_33
5. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Computer Security Foundations Workshop, (CSFW-17). pp. 100–114 (2004)
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193
7. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: Smotherspectre: exploiting speculative execution through port contention. CoRR **abs/1903.01843** (2019), http://arxiv.org/abs/1903.01843
8. Boudol, G., Petri, G.: A theory of speculative computation. In: ESOP. Lecture Notes in Computer Science, vol. 6012, pp. 165–184. Springer (2010). https://doi.org/10.1007/978-3-642-11957-6_10
9. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. CoRR **abs/1811.05441** (2018)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
11. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: POST. pp. 265–284 (2014)
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
13. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: SPECTECTOR: principled detection of speculative information flows. CoRR **abs/1812.08639** (2018), http://arxiv.org/abs/1812.08639
14. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: CAV. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015)
15. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Sec. **8**(6), 399–422 (2009)
16. Hosabettu, R., Gopalakrishnan, G., Srivas, M.K.: Verifying advanced microarchitectures that support speculation and exceptions. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 521–537. Springer (2000). https://doi.org/10.1007/10722167_39
17. Intel: White paper: Intel analysis of speculative execution side channels. Tech. Rep. 336983-001, Revision 1.0, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
18. Intel: Q2 2018 speculative execution side channel update (2018), https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, online. Accessed May 2019
19. Jhala, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: CAV. Lecture Notes in Computer Science, vol. 2102, pp. 396–410. Springer (2001). https://doi.org/10.1007/3-540-44585-4_40
20. Kiriansky, V., Waldspurger, C.: Speculative buffer overflows: Attacks and defenses. CoRR **abs/1807.03757** (2018), http://arxiv.org/abs/1807.03757
21. Kocher, P.: Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

22. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. CoRR **abs/1801.01203** (2018), `http://arxiv.org/abs/1801.01203`
23. Lahiri, S.K., Bryant, R.E.: Deductive verification of advanced out-of-order microprocessors. In: CAV. Lecture Notes in Computer Science, vol. 2725, pp. 341–353. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_33
24. Maisuradze, G., Rossow, C.: ret2spec: Speculative execution using return stack buffers. In: CCS. pp. 2109–2122. ACM (2018). https://doi.org/10.1145/3243734.3243761
25. Pardoe, A.: Spectre mitigations in msvc (2018), `https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/`, online. Accessed May 2019
26. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: ECOOP. pp. 362–386 (2005)
27. Rodrigues, B., Pereira, F.M.Q., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: CC. pp. 110–120. ACM (2016). https://doi.org/10.1145/2892208.2892230
28. Sawada, J., Jr., W.A.H.: Processor verification with precise exeptions and speculative execution. In: CAV. Lecture Notes in Computer Science, vol. 1427, pp. 135–146. Springer (1998). https://doi.org/10.1007/BFb0028740
29. Stecklina, J., Prescher, T.: Lazyfp: Leaking FPU register state using microarchitectural side-channels. CoRR **abs/1806.07480** (2018), `http://arxiv.org/abs/1806.07480`
30. Taram, M., Venkat, A., Tullsen, D.M.: Context-sensitive fencing: Securing speculative execution via microcode customization. In: ASPLOS. pp. 395–410. ACM (2019). https://doi.org/10.1145/3297858.3304060
31. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 352–367. Springer (2005)
32. Velev, M.N.: Formal verification of VLIW microprocessors with speculative execution. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 296–311. Springer (2000). https://doi.org/10.1007/10722167_24
33. Wang, G., Chattopadhyay, S., Gotovchits, I., Mitra, T., Roychoudhury, A.: oo7: Low-overhead defense against spectre attacks via binary analysis. CoRR **abs/1807.05843** (2018), `http://arxiv.org/abs/1807.05843`
34. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: CAV. Lecture Notes in Computer Science, vol. 10982, pp. 136–156. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_11