
Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan
MIT CSAIL

Abstract—Traditional architectural modeling aims to obtain an accurate estimation of performance, area, and energy of a processor design. With the advent of speculative execution attacks and their security concerns, these traditional ways of modeling architectures fall short when used for security evaluation of defenses against these attacks.

This paper presents Pensieve, a security evaluation framework targeting early-stage microarchitectural defenses against speculative execution attacks. We present a modeling discipline for systematically studying early-stage defenses. Our approach allows us to cover a space of designs that are functionally equivalent while precisely capturing timing variations due to resource contention and microarchitectural optimizations. We implement a model checking framework to automatically find vulnerabilities in designs. We use Pensieve to evaluate a series of state-of-the-art invisible speculation defense schemes, including Delay-on-Miss, InvisiSpec, and GhostMinion, against the speculative non-interference property. Pensieve is able to find Spectre-like attacks, including an implementation flaw in GhostMinion and a new speculative interference attack variant.

I. INTRODUCTION

Speculative execution attacks, such as Spectre [34] and Meltdown [37], have become one of the most critical security threats in the computer architecture community. These attacks exploit the side effects of transient instructions, which due to mis-speculation, are speculatively executed but squashed later, and leak secrets via various micro-architectural structures, including caches [33],[35, 37, 39, 49], TLBs [26], branch predictors [15], and functional units [8]. Researchers have been actively proposing mitigation mechanisms to block speculative execution attacks. For example, several mechanisms aim to achieve invisible speculation by hiding the changes of cache states caused by speculative memory instructions: InvisiSpec [58], SafeSpec [32], GhostLoads [47], Delay-on-Miss [48], and Muontrap [3]. Unfortunately, these plausible mitigation mechanisms were later found to be vulnerable to more advanced attack strategies called speculative interference attacks [7,48]. As a response, GhostMinion [1] was proposed to fix the security problems and specifically mitigate the speculative interference attacks. However, it is unclear whether the new mechanism is really bullet-proof, until either we find a new attack to break it or formally prove what security it guarantees.

The community has been mostly relying on informal analyses to study the security of microarchitectural defenses against speculative execution attacks. In most defense papers, the authors first implement their proposed defense mechanism in a concrete Gem5 [9] model and show that the implementation can practically block one specific attack (e.g., the Spectre gadget). Next, a more general security argument is made: it is argued intuitively why the defense scheme should be secure, and why it does not just block a single attack for one specific microarchitectural model. Given the complexity of micro-architectures, such an analysis is unlikely to get high-assurance security claims.


This paper strives to help architects to formally evaluate the security properties of defense proposals against speculative execution attacks. One of the key obstacles of performing a formal evaluation is the lack of a proper modeling approach of early-stage microarchitectural designs, which are usually loosely described at a high abstraction level using human languages. For the purpose of security evaluation, a desired architectural model should satisfy the following requirements.

First, since speculative execution attacks exploit timing side channels, a proper modeling approach needs to be precise in capturing any timing variations due to resource contention and microarchitectural optimizations. Unfortunately, microarchitectural analytical models (e.g., roofline models), and architecture simulators (e.g., Gem5), are insufficient, as they only try to approximate the precise timing for performance evaluation. For example, when modeling cache performance, a Gem5 simulator can ignore details such as port contention for fast simulation, which has negligible impacts on the reported performance statistics, but can miss important speculative execution vulnerabilities.

Second, a modeling approach of early-stage microarchitecture designs needs to cover a space of designs that are functionally equivalent with different timing. The reason is that computer architects generally expect and even claim that the proposed defense schemes are generic and compatible with different design configurations, instead of targeting a single implementation. For example, a defense scheme evaluated to be secure when a processor has a 1-cycle functional unit is usually expected to work securely when the functional unit takes 2 cycles or more, or no matter whether the functional unit is pipelined or not. Essentially, we need a modeling approach to precisely describe a space of microarchitecture designs. This requirement rules out the approach of encoding a concrete implementation of a defense scheme at the level of synthesizable RTL. Moreover, our decision to not work at the level of RTL is also driven by the fact that...
almost no defense mechanism proposed in the last couple of years [1, 3, 32, 38, 48, 58, 60] has been given with an RTL implementation. A potential reason is that people would not want to spend a huge amount of engineering effort before they have the assurance that a design is secure, which further motivates us to focus on modeling approaches targeting early-stage designs.

Third, the modeling approach should be aligned with the procedure used by computer architects to come up with defense proposals. Computer architects commonly view microprocessors as built by hierarchically composing many different modules, such as branch predictor modules, functional unit modules, and cache modules. Microarchitectural defense schemes are often conceived and intended to be used as a modular add-on to an existing architectural design. Specifically, designers usually select a certain number of the original modules and augment these modules with security-related features that change the modules’ timing behaviors, such as delaying a load upon cache misses [48]. These hardened modules are then plugged back into the existing architectural design, and the resulting design is expected to work securely against a broad set of speculative execution attack variations. Usually many modules of the original design are untouched by the defense scheme. Unfortunately, we find that no existing modeling approach both aligns well with such a modular design procedure and is also amenable to evaluating speculative execution vulnerabilities.

B. This paper

In this paper, we present Pensieve, a security evaluation framework for defenses against speculative execution attacks targeting early-stage microarchitectural designs. Pensieve introduces a modeling discipline that is specifically designed for the purpose of security evaluation of timing side channels, and leverages symbolic execution and bounded model checking to help architects automatically check defense schemes against formal guarantees.

Our modeling discipline describes a microarchitectural design as a modular composition of architectural modules, matching the modular design procedure widely used by computer architects. We further decompose each module into a functionality submodule and a timing submodule and enforce a handshaking interface between different modules. Our modeling discipline allows a designers to express functionality without worrying about timing or RTL synthesizability, and meanwhile allows capturing timing signals at cycle granularity (RTL-level precision) for precisely reasoning about timing variations due to any type of resource contention or microarchitectural optimization. Furthermore, by decoupling functionality and timing and using an abstraction level higher than RTL, our modeling discipline allows designers to focus on what factors affect timing and not on specific implementation details.

Pensieve is an automatic security evaluation framework built around this modeling discipline. Using the framework, a user can formally evaluate the security of a defense mechanism. If the security property is not satisfied, the model checker generates a counterexample, which consists of an attack code sequence, the initial state of the architectural model, and the execution trace that leaks secrets. The user can then inspect the counterexample to locate the problem in the defense mechanism.

We implement Pensieve in Rosette [51], a solver-aided programming language. We use Pensieve to perform case studies of state-of-the-art defense mechanisms against speculative execution attacks, including Delay-on-Miss [48], InvisiSpec [58], SafeSpec [32], GhostLoads [47], Muontrap [3], and GhostMinion [1]. In these case studies, we were able to identify known vulnerabilities in these defenses by finding Spectre-like attacks and speculative interference attacks.

More excitingly, Pensieve is able to find design flaws which were not known before and discover new variations of speculative interference attacks that were not discussed in prior work [7, 24]. Specifically, we find an implementation flaw in GhostMinion’s [1] timestamp generation scheme. While prior work focuses on exploiting gadget patterns where a transient “interfering” instruction causes contention with a non-speculative “transmitter” instruction that is younger in program order, our tool found a gadget pattern where the two instructions are not related by program order (under certain conditions in which a squashed interfering instruction has a long-lasting, in-flight memory request).

Contributions In summary, this paper makes the following contributions:

- We propose a microarchitectural modeling discipline for formally and systematically studying early-stage defenses against speculative execution attacks. Our modeling approach precisely captures timing variations due to contention and microarchitectural optimizations, covers a space of designs that are functionally equivalent with different timing, and is aligned with the way architects come up with defense proposals.
- We build a security evaluation framework upon this modeling approach that leverages bounded model checking to automatically find speculative execution vulnerabilities in designs.
- We apply Pensieve to evaluate the security properties of a series of state-of-the-art defense mechanisms, including Delay-on-Miss, InvisiSpec, and GhostMinion. We find valid attacks and flaws in our case studies, including an implementation flaw in GhostMinion and a new speculative interference attack variant.

II. BACKGROUND

A. Speculative Execution Attacks and Defenses

Speculative execution attacks are a class of information leakage attacks where attackers exploit the side effects of transient instructions. A transient instruction is an instruction that is speculatively executed on an out-of-order core but is later squashed due to mis-speculation, i.e. under a mis-speculated branch. The side effects of transient instructions
include modifying the states of micro-architectural structures, such as caches [34, 37], TLBs [26], and branch predictors [15], which can be monitored by an attacker program. High-profile speculative execution attacks include Meltdown [37], Spectre [34], and its variants [33, 35, 39, 49].

**Invisible speculation** The first set of defenses focuses on achieving “invisible speculation” [3, 32, 48, 58]. As caches are one of the largest attack surfaces, these defenses attempt to block speculative execution attacks by hiding the side effects of speculative memory accesses on the cache hierarchy.

Delay-on-Miss (DoM) [48] modifies the L1 cache. When serving a speculative load request that hits in the L1 cache, the data will be returned to the core without updating any cache states, including replacement bits. If the speculative load misses in L1, it will be delayed until it is non-speculative.

Other invisible speculation schemes, including InvisiSpec [58], SafeSpec [32], and Muontrap [3], make changes to multiple levels of caches. They allow speculative loads to lookup and fetch data from all levels of caches without changing persistent cache states, and store the fetched data in a small buffer close to the core. These schemes differ in details, such as buffer organization and consistency support.

**Speculative interference attacks** The speculative interference attack [7] is a variation of speculative execution attacks that can bypass all the invisible speculation schemes above and leak secrets via caches. The key insight is that mis-speculated younger instructions (called “interfering instructions”) can change the timing of older, bound-to-retire instructions (called “transmitter instructions”) via transient contention on ALUs and MSHRs. This attack was also observed in SpectreRewind [24]. Moreover, such timing interference can change the order of memory accesses from bound-to-retire instructions. As a result, they can trigger specific conditions of the cache replacement policy to make persistent and observable changes to cache states.

**GhostMinion** As a response, GhostMinion was proposed to defend against speculative interference attacks. The authors define a security property called temporal ordering, which states that the processor only allows committed instructions or older instructions to affect the timing of younger instructions, where “older” and “younger” are defined according to program order. GhostMinion also provides an implementation of temporal ordering. The processor maintains a timestamp for each instruction to track program order, and every micro-architecture structure is augmented with a scheduling or resource allocation policy that gives higher priority to older instructions.

**B. Model Checking**

Model checking [17] is an automatic (or “push-button”) technique that can be used to check whether a given finite-state model satisfies a logical formula (e.g. a property related to functional correctness or security). This technique has been applied to find subtle bugs in complex designs, such as communication protocols [16], sequential circuits [10, 11], and micro-controllers [18]. Bounded model checking checks for all states that can be reached with a transition length of $k$.

When using a model checking tool (also referred to as a model checker), the user needs to provide a clearly defined model of the system and a formulated property to be verified. The model checking tool takes the two inputs and generates one of the following outputs: 1) the tool terminates and indicates the model satisfies the property, 2) the tool terminates with concrete counterexamples to show why the model fails to hold the property, and 3) the tool does not terminate due to scalability issues and cannot provide an answer.

The state-of-the-art model checking techniques use solvers (e.g. Z3 [19], Boolector [44], CVC5 [6], JasperGold [12]) to check whether the generated formula holds and return a counterexample otherwise.

**C. Uninterpreted Function**

An uninterpreted function is a function with specified input and output types, but with an unspecified function body. For example, a function $f : (\text{bool}, \text{bool}) \rightarrow \text{bool}$ can be viewed as a set of functions performing two-bit operations, such as AND or OR. This space of possible behaviors can be restricted by adding constraints on the function. Uninterpreted functions are maximally flexible, representing any function that satisfies any additional constraints.

Uninterpreted functions are widely used in the formal-methods community to abstract away unimportant design details either for generality (i.e. to represent a larger space of designs) or to remove details known to increase verification complexity (while still preserving the soundness of the verification result).

**III. Security Goal and Threat Model**

In this paper, we perform security evaluation using a formal security property called *speculative non-interference*, which has been used in prior work [13, 28, 29]. Intuitively, speculative non-interference requires that a program executing on a speculative machine does not leak more information than when it is running on a non-speculative machine.

Formally, we define speculative non-interference by introducing the following symbols. Given a program $P$ and a memory consisting of both public data $M_{\text{pub}}$ and secret data $M_{\text{sec}}$ executing on a microarchitectural design $\mu$, we denote the attacker’s observation of the system as $O_{\mu}(P, M_{\text{pub}}, M_{\text{sec}})$. We denote $O_{\text{ISA}}$ as an ISA emulator that serves as the non-speculative machine and $O_{\text{O3}}$ as the target speculative out-of-order processor under study. The following property states the property of speculative non-interference for $O_{\text{O3}}$:

$$\forall P, M_{\text{pub}}, M_{\text{sec}}, M'_{\text{sec}}. \quad \text{if} \quad O_{\text{ISA}}(P, M_{\text{pub}}, M_{\text{sec}}) = O_{\text{ISA}}(P, M'_{\text{pub}}, M_{\text{sec}}) \quad \text{then} \quad O_{\text{O3}}(P, M_{\text{pub}}, M_{\text{sec}}) = O_{\text{O3}}(P, M'_{\text{pub}}, M_{\text{sec}})$$

The property states that for any program that runs on two distinct secrets $M_{\text{sec}}$ and $M'_{\text{sec}}$, if the program produces indistinguishable observations in the ISA model, then the observations generated by the $O_{\text{O3}}$ machine should also be indistinguishable.
Speculative non-interference only talks about the behaviors of programs that are secure in the ISA model. If a program were to be already leaky when executing non-speculatively, there would be nothing that a defense against speculative execution attacks could do about it.

**Observation function** Prior work have used at least three variants of observation functions for an O3 processor:

1. A trace of committed instructions and each instruction’s commit time.
2. A trace of the addresses of memory requests and each memory request’s issue time and response time.
3. An ordered trace of the addresses of memory requests with no cycle number.

There have been discussions [1, 7] that using the three observation functions for security evaluation is likely to lead to the same outcome. It is a non-goal for this paper to argue which observation function is better.

The Pensieve evaluation framework is general enough and can work with any of the three observation functions. In the rest of this paper, we use function (1) to explain the case study examples, as using this function usually yields the shortest attack code sequences.

**IV. Modeling Micro-architectural Designs**

Pensieve is a security evaluation framework targeting early-stage microarchitectural defenses against speculative execution attacks. Early-stage microarchitectural defenses are often proposed as modular add-ons to existing designs that change the modules’ timing behaviors, such as delaying a load upon cache misses. In this section, we describe a microarchitectural modeling discipline amenable to studying speculative execution vulnerabilities. We start by presenting two toy examples to give a high-level idea about how our modeling discipline achieves the three desired properties in Section I-A. We then discuss the key techniques used in our modeling discipline and explain how to express defense mechanisms as modular add-ons to a baseline architecture.

**A. Two Toy Examples: A Multiplier and Memory**

We illustrate our approach with a multiplier with an arbitrary number of pipeline stages and computation latency, and a memory hierarchy with an arbitrary number of cache levels, using arbitrary cache associativity and replacement policy. Fig. 1 illustrates the modular view of the two examples. Fig. 2 and Fig. 3 provide an implementation of an abstract memory.
and when data in the buffer can be forwarded to the output.

To see how uninterpreted functions are used, we provide the code for an abstract single-channel delay FIFO in Fig. 3. Lines 2-8 list the buffer’s interface with an input and output channel for data transfers and two pairs of valid and ready signals for hand-shaking. Lines 10-20 show the implementation of a FIFO’s enqueue and dequeue operations. Lines 22-31 show how the timing signals are generated using uninterpreted functions. In this example, the ready and valid signals are generated by the uninterpreted functions \( F_1 \) and \( F_2 \), which take the history of cycle-by-cycle inputs and any additional timing-related signals specified by users, denoted as \( T \)factor.

### B. The Modeling Discipline

We now summarize our modeling discipline. Given a processor, we express the design as a modular composition of architectural modules and think of each module as performing some (potentially unspecified) functionality with some (potentially unspecified) latency, and so further decompose each module into functionality and timing submodules. We use a standardized hand-shaking interface for inter-module communication and uninterpreted functions to capture the “potentially unspecified” nature of each submodule.

**Hand-shaking interface** For each channel at every clock cycle, the ready signal indicates whether a module can take new input at that cycle, and the valid signal indicates whether a module generates an output at that cycle. This is a standard interface that is familiar to RTL designers. We have shown that such an interface helps us to distinguish between functionality and timing signals in the two toy examples.

In addition, the interface mimics wire-level behaviors, e.g., for each input and output channel, only one piece of data can use the channel per cycle. This allows us to capture precise timing activities at cycle granularity with RTL-level precision. This differs from simulators such as Gem5, where a user has the freedom to decide whether some contention details need to be modeled.

**Uninterpreted functions** In Pensieve, we extensively use uninterpreted functions (UFs for short) for two purposes. First, we leverage UFs for generality: to represent a large space of designs. In addition to using them in the timing submodules (recall the multiplier and memory examples in Fig. 1), we also use UFs in functionality modules, i.e., in the fetch module to model a set of branch predictors.

Second, we use UFs to reduce modeling efforts by hiding unimportant design details. A UF \( f: \text{input} \rightarrow \text{output} \) explicitly states that the variables that are part of the function’s input may influence the output, and anything that does not belong to the input will not influence the output. Therefore, using UFs allows users to focus on what affects the timing of a design, rather than how these factors affect timing.

### C. Covering Complex Designs with Simple Models

Thanks to the fact that a single uninterpreted function can represent a large set of concrete functions, our modeling approach allows us to use simple models to 1) cover large design spaces, and 2) cover complex design features.

As an example, consider the modeling of a large family of multipliers (Fig. 1(a)). We use an UF to generate the output valid signal, and the UF uses the history of the ready and valid signals as arguments. Such a simple model describes any first-in-first-out multiplier module that takes an arbitrary number of cycles to process each request. If we use the instruction’s operands as additional arguments of the UF, the UF states that the operand values can influence the computation latency. Such a simple tweak allows us to cover multipliers with value-based optimizations, such as zero-skip and table lookup of pre-computed values.

To further illustrate the point that simple models in Pensieve can cover the timing behaviors exhibited by complex designs, consider the memory example. When using a UF with the instruction’s address as an argument, the model represents any memory hierarchy whose response latency depends on the history of requested addresses. Such a memory hierarchy is extremely generic, and thus includes memory designs with an arbitrary cache hierarchy with varied cache levels, capacity, associativity, and replacement policy. In addition, it includes any address-based optimizations, such as prefetching and silent data eviction using potentially complex heuristics.

As is traditional in the hardware formal methods community, we can also use UFs to model the functionality of branch predictors. This allows us to model arbitrary branch direction prediction schemes, ranging from a simple tournament policy to a complex TAGE design. Moreover, applying UFs in the issue scheduler models an arbitrary scheduler that reorders instructions using various priority policies.

In summary, most of the microarchitecture designs and optimizations discussed above are fairly complex, but their complexity lies in the details of how certain inputs influence the outputs. Pensieve uses UFs to abstract these details while still allowing the model to cover these design points.

### D. Modeling Defenses As Modular Add-ons

With our modeling discipline, a defense mechanism, in the form of a modular augmentation to a baseline architecture, can be described by making modifications to the models of corresponding modules. Since our model covers a space of designs with equivalent functionality and different timing, modifying a defense corresponds to reducing the space of the allowed timing behaviors: We summarize three ways to apply such modifications:

(a) Carefully selecting the input argument of uninterpreted functions. Removing an element from the argument...
makes the output of the uninterpreted function independent of the removed element.

(b) Wrapping an uninterpreted function with concrete functions into a partially abstract function.

(c) Adding assumptions to an uninterpreted function so that it does not cover arbitrary functions.

We provide a few examples by showing how to model Delay-on-Miss [48] shown in Fig. 4. We start by decomposing the monolithic memory module into two separate modules, i.e., the L1 cache and the rest of the memory. The L1 cache uses uninterpreted functions inside both the functionality and timing submodules, modeling different aspects of the L1 cache. The one in the functionality submodule determines whether a memory request results in a cache hit or miss, modeling the cache configurations that can affect cache hit activities, such as capacity, associativity, and replacement policies. The one in the timing submodule determines when the L1 cache can take on a new request and when a request/response can be forwarded, modeling varied cache lookup, writeback latency, and bank/port contention between concurrent requests.

To model DoM, we make the following changes to the L1 cache module. First, DoM disables any speculative requests to change the L1 cache states that can affect cache hit/miss events. We use approach (a) to model this feature. We take the uninterpreted function for determining whether a memory request is a cache hit or miss and exclude the addresses of speculative memory requests from the history input of this uninterpreted function. As a result, cache hit/miss activities are independent of any previous speculative requests.

Second, DoM disables speculative L1 misses to be forwarded to the rest of the memory hierarchy. We use approach (b) to model this feature by adding a concrete function around the uninterpreted function. Specifically, when the uninterpreted function generates a cache miss and the request is speculative, the added concrete function forces the functionality submodule to send a retry signal to the CPU.

Though our modeling discipline is usually easy to use, we acknowledge that sometimes, due to the ambiguity in describing a defense mechanism using human language, it can be challenging to figure out the precise constraints. Translating human language to a formal microarchitectural model is a challenging research problem that cannot be completely addressed by our modeling discipline, but this paper is helping to make some progress.

E. Abstract Delay Buffer Variation

In addition to the single-channel abstract delay FIFO in Fig. 3, we provide the following variations.

First, multiple input and output channels. We can have abstract delay buffers supporting multiple input and output channels to model scenarios where the buffers take more than one input per cycle, e.g., a dual-port register or cache.

Second, non-FIFO order. The example in Fig. 3 describes an abstract FIFO. We can have a variation of the buffer to forward data using a non-FIFO order. Specifically, we use an uninterpreted function to decide which valid entry should be selected to forward to the output channel. We use this to model the dispatch module in our baseline processor.

Third, reset signals. We use two types of reset signals to the buffer. The first type clears all the data stored in the buffer. The second type additionally resets the timing behaviors of the buffer by clearing the input arguments of all the uninterpreted functions (the variable history in Fig. 3).

V. Pensieve: Framework and Implementation

Pensieve is a framework for automatic security evaluation of microarchitectural defenses against speculative execution attacks. Fig. 5 shows the overview of the framework and its workflow. The workflow follows the standard model checking process with each step being carefully tuned for users to effectively evaluate speculative execution vulnerabilities.

In step 1, the user specifies the input to the framework, which consists of a µarch model and a security property. The challenge of this process lies in converting a defense mechanism, which is usually described in human language, into a formally specified µarch model. Since most defense mechanisms are described as modular add-ons to an unspecified baseline architecture, Pensieve provides a baseline µarch model to reduce the manual effort needed to construct a µarch model for the defense mechanism from scratch.

In step 2, a model checker is used to perform automatic security evaluation. The model checker uses symbolic execution to perform bounded model checking of the µarch model against the specified security property. The model checker will either indicate that the security property holds for a given number of cycles, or generate a counterexample to indicate the discovery of a speculative execution vulnerability.

In step 3, the user inspects the counterexample generated by Pensieve to locate the speculative execution vulnerability in the evaluated defense mechanism. A counterexample consists of
an attack program, the initial states of the \( \mu \text{arch} \) model, and a raw execution trace.

### A. Baseline \( \mu \text{Arch} \) Model

We provide a baseline out-of-order processor model that users can conveniently extend with a defense mechanism, expressed as a modular add-on. Our baseline \( \mu \text{arch} \) model, though simple, covers a large design space and potentially complex pipeline scheduling policies. The baseline \( \mu \text{arch} \) model models the following pipeline stages: fetch, decode&rename, dispatch, execute, memory, commit. Section IV-C has listed most of the UFs used in the fetch, dispatch, and execution stages. Importantly, our modeling allows various types of instruction reordering, primarily from two sources. First, as we use UFs inside the timing modules of ALU and memory, each instruction can take an arbitrary latency in the execution stage, and its dependent instructions can be ready at an arbitrary cycle. Second, as we use UFs inside the dispatch stage, the baseline \( \mu \text{arch} \) model models a wide range of issue policies that can reorder loads, stores, branches, and arithmetic instructions as long as their operands are ready.

We note that our baseline \( \mu \text{arch} \) model has concrete decode&rename and commit stages. We show our baseline \( \mu \text{arch} \) model was sufficient to find vulnerabilities in existing defenses in our case studies and discuss its limitations in Section VIII.

### B. Model Checker

Pensieve automatically evaluates the security property of a \( \mu \text{arch} \) model using symbolic execution and bounded model checking. Step 2 in Fig. 5 describes the evaluation process. First, we encode the \( \mu \text{arch} \) model as a finite state machine (FSM) with a set of state bitvectors and a set of update logics. The state bitvectors of the FSM, including the register content and memory content, are initialized using symbolic terms. The update logic includes both concrete cycle-by-cycle update functions and uninterpreted functions in the given \( \mu \text{arch} \) model. This encoding allows us to simulate the execution of symbolic instruction sequences operating on symbolic data.

Next, we execute the \( \mu \text{arch} \) model for a bounded number of cycles to derive a symbolic observation trace. Following the security property definition in Section III, we have a constraint on the software program that it should not leak secrets when executing in an ISA emulator. We add this constraint to the symbolic formula before sending it to the SMT solver.

The SMT solver searches for counterexamples as follows. It tries to assign a concrete value to every symbolic variable (called binding or variable-binding) and assign a concrete function to every uninterpreted function, such that these bindings make the formula evaluated to be false, which in our context means “a counterexample violates the security property.” Specifically, the bindings for the symbolic variables generate the model’s initial states where the instruction memory stores the attack program. The bindings for the uninterpreted functions manifest the raw execution trace, i.e., per-cycle hand-shaking signals for every module.

### Implementation

We implement Pensieve on top of Racket and the Rosette [51] solver-aided programming language, which uses an SMT backend. Specifically, we encode the \( \mu \text{arch} \) model and security property in Rosette, and then leverage Rosette’s symbolic execution and solver backend to search for a counterexample violating the security property. Like most model checkers [20, 31, 40], Pensieve can suffer from performance problems, limiting the number of cycles that can be checked.

### C. Counterexample Inspection

The model checker outputs a counterexample when the \( \mu \text{arch} \) model does not satisfy the security property. A user can inspect the counterexample to pinpoint the speculative execution security vulnerability following three steps.

First, we convert the raw trace to a readable pipeline view. The raw trace includes all the cycle-by-cycle handshaking signals for inter-module communication. It looks similar to the waveform graphs used for debugging synthesizable RTL code and is exhaustive but difficult to read. To ease the inspection process, we convert this raw trace into a readable pipeline view by giving each signal its microarchitectural semantics based on which hardware modules they are connected to. We also correlate each pipeline signal with its corresponding instruction. We perform this process manually, but it can be automated with visualization tools such as Konata [50].

Second, we locate the earliest point at which the two traces defined in our security property deviate. We search all the timing signal traces (for valid and ready signals, not the signals corresponding to the data channels) and locate the one that exhibits the earliest timing deviation.

Third, as a final step to pinpoint the security vulnerability, a user needs to inspect the full information flow to figure out how the secret reaches the earliest deviation point. As the timing signals are usually generated using UFs, we can check the arguments for the UF to locate which modules in the \( \mu \text{arch} \) model introduces the secret-dependent timing variations.

### The DoM counterexample

Fig. 6 shows the counterexample generated by Pensieve for the DoM \( \mu \text{arch} \) model, with the attack code sequence in Fig. 6(a), pipeline view of the execution trace in Fig. 6(b), and trace for signal between the CPU and the L1 cache in Fig. 6(c), called CPU-Mem traces. This counterexample indicates that the DoM \( \mu \text{arch} \) model is vulnerable to a speculative interference attack caused by contention inside the L1 cache.

To see why, we take a look at the attack code, which shows that instructions LD1 and LD2 will commit, and instruction LD3 is a transient load. According to the pipeline execution trace, the transient load LD3 issues a memory request at cycle 2 and lookups the L1 cache. In the CPU-Mem trace, we find the transient instruction LD3’s speculative cache access (the slashed node) influences the committed instruction LD2’s memory access latency and commit time. The counterexample will then drive the user (a computer architect) to think about the cause of the interference and give architectural meaning to the attack. In this DoM case, we think the vulnerability exists.
because DoM allows speculative requests to cause bank/port contention with non-speculative requests.

D. Understanding Pensieve’s Attack Coverage

Due to performance reasons, Pensieve’s attack coverage is constrained by the number of cycles it explores, i.e., 9 cycles as shown later. However, a reader should not directly translate the 9 cycles explored by Pensieve to the cycle count on a concrete processor.

Consider that real-world attacks often involve a large number of instructions primarily for the following purposes: 1) precondition microarchitecture states, e.g., mistraining the branch predictor or prefilling the caches; 2) increase speculative execution window size by introducing extra delays before a mispredicted branch; 3) manipulate the ordering of instructions across speculation windows to expose or amplify the side effects of transmitter instructions. Pensieve can generate compact counterexamples that reassemble real-world attacks by exploring a “worst-case” design and execution of the model such that an attack with a minimal number of instructions can work. This feature shares some similarities with how CheckMate [52] models hardware and finds security vulnerabilities by exploring only 6 instructions.

For example, in the DoM counterexample (Fig. 6), the BR instruction is mispredicted. This misprediction is automatically generated by the model checker without needing extra training instructions (addressing 1). Next, the attack needs a long speculation window for LD3 to have sufficient time to interfere with LD2. The model checker finds the binding of the UF in the memory module to make the LD2’s latency long enough to delay the resolution time of the BR (addressing 2). Finally, a real-world attack needs to fine-tune the issue time of multiple memory instructions to amplify L1 contention. However, the UF’s in L1 cache allow any input to interfere with the timing of the output, and the model checker can easily find the pair of inputs (different addresses of LD3) to expose the timing variation (addressing 3).

VI. CASE STUDY: GHOSTMINION

We used Pensieve to evaluate the security property of a series of defense mechanisms, including DoM [48], InvisiSpec and its variants [3, 32, 47, 58], and GhostMinion [1]. We successfully found vulnerabilities and generated valid attacks.

Fig. 6: A counterexample generated by Pensieve for a µarch model of DoM. In the CPU-Mem trace, a hollow node indicates a memory request and a solid node is a response. The slashed nodes indicate that the memory requests differ in their addresses.

Fig. 7: Comparing the attack code pattern exploited in prior work and the new variance of speculative execution attacks.

Most of the counterexamples reassemble the speculative interference attacks described in [7, 24], showing that Pensieve can systematically find known vulnerabilities. Due to space limitations and for the reader’s interest, we provide one detailed case study to showcase that Pensieve can also find new vulnerabilities.

We present a case study evaluating GhostMinion [1], the state-of-the-art defense scheme that is claimed to be resilient against speculative interference attacks. Using Pensieve, we find GhostMinion does not achieve the speculative non-interference property due to an implementation flaw in its timestamp generation scheme. The generated attack leads to the discovery of a new speculative interference attack variant.

A New Speculative Interference Attack Variant First, we summarize the new variant of speculative interference attack found by our framework. Following the terminology used in [7], we call a transient instruction that causes contention as an interfering instruction, and a non-speculative instruction whose timing is influenced as a transmitter instruction.

Prior work focuses on exploiting a gadget pattern where the interfering instruction is younger than the transmitter instruction in program order (Fig. 7(a)). Instead, our tool found a gadget pattern where there does not exist a program order between the two instructions, such as when the two instructions are from different sides of a branch (Fig. 7(b)).

To see how speculative interference attacks work with the new code pattern, consider the following squash mechanism: a processor quickly resumes the execution following the correct code path without waiting for the responses of outstanding memory requests to return. The processor then relies on some bookkeeping mechanisms at the load/store queue to
After

[4x63]R0 <- ld
[4x78]if (R1 == 0) {
[4x92]R1 = R1 * R1
[4x107]R
[10x104]sec
[39x13](a) Monotonic
[49x99]•
[49x135]•
[49x150]three versions of the timestamp generation process.
[49x162]clearly specify how the timestamp is generated. We model
[49x186]requests with larger timestamps.
[49x201]add constraints to uninterpreted functions to ensure the latency
[49x237]FIFO ordering (Section IV-E) for each timing submodule
[49x258]Since the interfering instruction has a smaller timestamp, it
[49x270]in the memory system, causing interference to the transmitter.
[49x282]request, which was issued before the squash, is still in-flight
[49x294]though the interfering instruction is squashed, its memory
[49x306]re-enters the pipeline and gets a new timestamp of 5. Even
[49x318]and 4, respectively. After the squash, the transmitter instruction
[49x330]are both speculatively issued and assigned a timestamp of 3
[49x342]Before the squash, the interfering and transmitter instructions
[49x366]corresponds to the corrected path.
[49x390]corresponds to the case when the instruction enters the pipeline
[49x402]twice and so are given two timestamps. The first timestamp
[49x414]instructions after the branch merge point enter the pipeline
[49x426]timestamp before and after a pipeline squash. Remark that the
[49x438]shown in Fig. 8, where each instruction is marked with a
timestamp before and after a pipeline squash. Remark that the
instructions after the branch merge point enter the pipeline
twice and so are given two timestamps. The first timestamp

corresponds to the case when the instruction enters the pipeline

as part of the mispredicted path, and the second timestamp
corresponds to the corrected path.

Fig. 8(a) illustrates why the monotonic scheme is insecure.
Before the squash, the interfering and transmitter instructions
are both speculatively issued and assigned a timestamp of 3
and 4, respectively. After the squash, the transmitter instruction
re-enters the pipeline and gets a new timestamp of 5. Even
though the interfering instruction is squashed, its memory
request, which was issued before the squash, is still in-flight
in the memory system, causing interference to the transmitter.
Since the interfering instruction has a smaller timestamp, it
has the priority to use any microarchitecture structure.

We model the priority ordering and leapfrogging operation
as follows. We use an abstract delay buffer with a non-
FIFO ordering (Section IV-E) for each timing submodule
and configure it to always select the data with the smallest
timestamp to forward to the output channel. In addition, we
add constraints to uninterpreted functions to ensure the latency
of the requests with smaller timestamps is independent of the
requests with larger timestamps.

Modeling Timestamp Generation One challenge in model-
ing GhostMinion is that the GhostMinion paper does not
clearly specify how the timestamp is generated. We model
three versions of the timestamp generation process.

• Monotonic: The timestamp is incremented when an instruc-
tion enters the decode&rename module. This is the version
implemented in the GhostMinion Gem5 simulator.

• Reset-upon-squash: The timestamp is incremented in the
same way as the Monotonic scheme, but upon squash, is re-
set to match the timestamp of the last committed instruction.

B. Counterexample Inspection

We run the model checker on the µarch model of GhostMin-
ion with the Monotonic and the Reset-upon-squash timestamp
generation schemes. We find counterexamples in both cases,
shown in Fig. 8, where each instruction is marked with a
timestamp before and after a pipeline squash. Remark that the
instructions after the branch merge point enter the pipeline
twice and so are given two timestamps. The first timestamp
corresponds to the case when the instruction enters the pipeline

as part of the mispredicted path, and the second timestamp
corresponds to the corrected path.

Fig. 8(a) illustrates why the monotonic scheme is insecure.
Before the squash, the interfering and transmitter instructions
are both speculatively issued and assigned a timestamp of 3
and 4, respectively. After the squash, the transmitter instruction
re-enters the pipeline and gets a new timestamp of 5. Even
though the interfering instruction is squashed, its memory
request, which was issued before the squash, is still in-flight
in the memory system, causing interference to the transmitter.
Since the interfering instruction has a smaller timestamp, it
has the priority to use any microarchitecture structure.

We model the priority ordering and leapfrogging operation
as follows. We use an abstract delay buffer with a non-
FIFO ordering (Section IV-E) for each timing submodule
and configure it to always select the data with the smallest
timestamp to forward to the output channel. In addition, we
add constraints to uninterpreted functions to ensure the latency
of the requests with smaller timestamps is independent of the
requests with larger timestamps.

Modeling Timestamp Generation One challenge in model-
ing GhostMinion is that the GhostMinion paper does not


This is our interpretation when reading the GhostMinion
paper [1] and is also suggested as a feasible scheme in [7].

• 2-tuple: Using Pensieve, we find that the previous two
schemes are insecure. We fix the vulnerability by proposing
a secure scheme that works on our baseline µarch model.

Problems of Using Gem5 For Security Evaluation To

further demonstrate the validity of our discovered attack
examples, we planned to implement a proof-of-concept attack
on the GhostMinion Gem5 simulator. However, we find the
simulator opensourced by the authors [2] does not implement
the leapfrogging operations. Instead, the simulator collects statistics about how frequently a leapfrogging operation is needed. Such a modeling is roughly sufficient for estimating the performance overhead introduced by the leapfrogging operation, but leaves the GhostMinion implementation vulnerable to the original speculative interference attacks.

This experience re-emphasizes our claim in Section I that architectural simulators designed for performance evaluation are not trustworthy for security evaluation. Even though Gem5 is one of the few simulators that model speculative execution in detail, when modified by a user to augment a defense mechanism, critical timing constraints may be omitted for convenience or better simulation speed. It is essential to design a proper microarchitectural modeling approach that captures precise timing characteristics for security evaluation, like the one proposed in this paper.

C. Fixing The Vulnerability

Leveraging the counterexamples, we try to propose a secure timestamp generation scheme to fix the security vulnerability in GhostMinion. A straightforward idea is to use a tuple of \(<\text{speculative window ID}, \text{instruction ID}>\) as a timestamp. The speculative window ID is incremented when a squash happens, and the instruction ID is incremented either following the monotonic or reset-upon-squash scheme. A priority is given to timestamps with a larger speculative window ID and a smaller instruction ID. Note that, this scheme only works for our baseline architecture, where a branch only squashes when it reaches the head of ROB, and will not work with processors implementing nested speculative branch squashes.

We find the 2-tuple timestamp generation scheme on our baseline architecture reassembles a concrete implementation of the Temporal Ordering, discussed in the GhostMinion paper. Therefore, we conclude a fair comment on GhostMinion as follows: “Enforcing temporal ordering on microarchitectural resource usage is an effective defense mechanism to achieve the speculative non-interference property. However, the implementation presented in the GhostMinion paper is incorrect.”

VII. PERFORMANCE EVALUATION

A. Experiment Setup

We evaluate Pensieve’s performance on four \(\mu\text{arch} \) models: 1) the baseline microarchitecture, 2) Delay-on-Miss [48], 3) Invisible Speculation (a \(\mu\text{arch} \) model covering InvisiSpec [58], SafeSpec [32], GhostLoads [47], Muontrap [3]), and 4) GhostMinion [1]. We use a 1-cycle L1 cache in the DoM \(\mu\text{arch} \) model to stress the performance of our model checker since it takes much longer to find the vulnerability due to speculative interference on the ALU module, which we will show shortly.

Our \(\mu\text{arch} \) models supports 5 types of instructions: load immediate, load register, store register, branch, and register-to-register operation. Each model has a 4-cycle register file, a 16-entry instruction memory, and a 4-entry data memory. The ISA emulator and baseline out-of-order processor were implemented in Rosette in \(\approx200 \) and \(\approx1500 \) lines of code (LOC), respectively. Adding Delay-on-Miss, InvisiSpec, and GhostMinion on top of the baseline processor required approximately 150, 300, and 400 LOC, respectively. The performance results were obtained on a server machine equipped with an Intel Xeon 5220R processor running at 2.2 GHz.

Like most model checking tools [20, 31, 36, 40], Pensieve can suffer from performance problems that limit the number of cycles that it can check. Our implementation of the model checker includes a commonly used optimization [4, 43], that is, concretization by forcing case analysis on symbolic terms.

B. Performance Evaluation

Fig. 9 shows the end-to-end execution time of the model checking tool for each model with varied ROB sizes (different lines) and evaluation cycles (the x-axes). Dots represent the cases when no counterexamples are found and crosses (“x”) are when counterexamples are found. If the execution time is above 1000 minutes (\(\sim17 \) hours), it is considered as a timeout and will not be shown in the figure.

From Fig. 9(a)-(d), we observe as expected that the checking time increases exponentially as the number of simulated cycles increases (the y-axes use log-scale). For example, when evaluating the baseline \(\mu\text{arch} \) model with 16 ROB entries, when we increase the number of simulation cycles from 5 to 6 and 7, checking time increases from 1.5 minutes to 3.5 minutes and 15 minutes, respectively.

For different defense mechanisms, it takes different numbers of simulation cycles to find counterexamples. It takes at least 4 cycles to find a counterexample on the baseline, 8 cycles on DoM, 5 cycles on InvisiSpec, and 7 cycles on GhostMinion.

We also observe that the ROB sizes affect both the checking time and whether we can find a counterexample. Since the ROB size directly affects the model complexity, the checking time increases as the ROB size increases. Besides, the ROB
size determines the number of concurrent instructions and the level of timing interference between instructions. In Fig. 9(b), when evaluating DoM, we only find a counterexample when the ROB size is at least 8.

Finally, Fig. 9(d) shows that it can sometimes take less time to find a counterexample than to guarantee that no counterexamples can be found, e.g., comparing the checking time of simulating 6 and 7 cycles when evaluating GhostMinion with 8 or 16 ROB entries. This is because the tool terminates as soon as a counterexample is found, but in the case of no counterexamples, the model checker study all reachable states.

Discussion From the analysis above, we identify three key factors that affect the performance of our model checker. First, the number of simulated cycles is the most important limiting factor, as the checking time increases exponentially with the simulated cycles. Second, it usually takes significantly more time to evaluate a “more secure” design. If a µarch model satisfies the security property, the checker needs to study all reachable states, resulting in a long execution time. For example, we frequently observe timeout when evaluating the DoM µarch model with a 1-cycle L1 cache since it takes more simulation cycles to reveal a counterexample. Third, increasing the ROB size in a µarch model will increase the number of state bits of the model and thus increases the checking time. We observe that the checking time is increased by around 4 times each time we double the ROB size.

VIII. LIMITATIONS AND FUTURE WORK

This paper has several limitations. First, we made several simplifications to our baseline µarch model that limits the strength of the security claim in the case that no counterexamples are found. For example, our current baseline µarch model has concrete decode&rename and commit stages, and supports a limited number of instruction types, registers, and memory entries. A defense designer wanting to evaluate the security of a defense on a larger space of baseline architectures would need to extend the µarch model by, e.g., further decomposing the concrete modules and using more abstract delay buffers.

Relatively, we only conduct bounded model checking up to 9 steps for performance reasons and thus, if a counterexample is not found, can not guarantee the absence of vulnerabilities taking more than 9 steps to exploit. This performance problem arises from the formal analysis method used in Pensieve, i.e., bounded model checking. Future work can address the scalability issues by using symbolic execution optimizations [4, 43] or, better yet, using more scalable formal analysis techniques such as invariant-based induction proofs in conjunction with Pensieve’s modeling approach. However, these techniques tend to require more manual effort and formal methods expertise, as opposed to a “push-button” tool that computer architects can conveniently use.

In addition, Pensieve focuses on early-stage designs and it was a non-goal to evaluate the security of RTL designs. Pensieve can be used to derive the specifications for RTL modules. Checking whether an RTL implementation implements a given µarch model is orthogonal but important future work.

Finally, we only used Pensieve to evaluate designs against speculative non-interference. It would be a straightforward extension to substitute similar security properties such as speculative data obliviousness, so that we would be able to evaluate schemes such as STT [60], NDA [55], etc [5, 38, 59]. However, Pensieve only models deterministic designs and does not support probabilistic security properties in designs, and will not be able to evaluate designs such as CleanupSpec [46].

IX. RELATED WORK

We discuss related work on performing formal security evaluation of microarchitectural defenses against speculative execution attacks. We identify that previous approaches differ along two axes: 1) the modeling approach, 2) the security evaluation techniques and properties.

Modeling Microarchitectures A number of verification projects including IntroSpector [25] and UPEC [21]–[23] directly operate on synthesizable RTL and thus verify a concrete implementation. However, as discussed in Section I-A, this is an orthogonal problem to evaluating early-stage designs as architects usually do not want to implement RTL before being confident about what security property their design guarantees.

On the abstract modeling side, CheckMate [52] and axiomatic LCM [42] use an axiomatic modeling approach to describe a processor design as a µlb graph. Hsiao et al further demonstrate that such a model can be automatically synthesized from RTL [30]. Guarnieri et al. [28, 29] and Guanciale et al. [27] define an operational model of out-of-order processors. Their models are monolithic and do not have an explicit notion of timing. A difference between those two kinds of work is also the style in which the models are written. [42, 52] use an axiomatic style, while Guarnieri et al. [28, 29] and Guanciale et al. [27] follow an operational style. Both modeling approaches are far from the way computer architects propose early-stage designs.

Security Evaluation Techniques Pensieve and UPEC [21]–[23] both use automatic model checking techniques. UPEC combines model checking with manual invariants to obtain a security guarantee on an unbounded number of cycles. Guarnieri et al. [28] use manual proof. CheckMate [52] uses relational model finding. CheckMate’s security property is derived from security litmus tests and the tools try to find new exploits from those patterns. Another set of work uses fuzzing to find new vulnerabilities in RTL or blackbox hardware, including IntroSpectre [25], Medusa [41], Speechminer [57], Osiris [54], and Revisor [45]. These approaches are effective but do not provide formal reasoning of the designs.

In addition to the work on verifying hardware defenses, there is extensive work on verifying software against speculative execution attacks [13, 14, 42, 53, 56].

X. CONCLUSION

Correctly modeling and evaluating a defense mechanism usually requires extensive formal-methods expertise and it is difficult to reason about whether the modeling reflects the designer’s ideas. Pensieve is a step towards addressing


