Pensieve: Microarchitectural Modeling for Security Evaluation

Yuheng Yang
Massachusetts Institute of Technology
Cambridge, MA, USA
yuhengy@mit.edu

Thomas Bourgeat
Massachusetts Institute of Technology
Cambridge, MA, USA
bthom@mit.edu

Stella Lau
Massachusetts Institute of Technology
Cambridge, MA, USA
stellal@mit.edu

Mengjia Yan
Massachusetts Institute of Technology
Cambridge, MA, USA
mengjiay@mit.edu

ABSTRACT

Traditional modeling approaches in computer architecture aim 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 modeling techniques 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. At the core, it introduces a modeling discipline for systematically studying early-stage defenses. This discipline 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 a formally defined security property, speculative non-interference. Pensieve finds Spectre-like attacks in all those defenses, including a new speculative interference attack variant that breaks GhostMinion, one of the latest defenses.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Formal security models**; • **Computer systems organization → Superscalar architectures**.

KEYWORDS

hardware security, speculative execution attacks, microarchitectural model, model checking, uninterpreted function

ACM Reference Format:
Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. 2023. Pensieve: Microarchitectural Modeling for Security Evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589094

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589094

1 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 microarchitectural 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, 24]. 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 microarchitectures, such an analysis is unlikely to get high-assurance security claims.

1.1 Motivation

This paper strives to help architects formally evaluate the security properties of defense proposals against speculative execution attacks. One of the key obstacles to performing a formal evaluation is **the lack of a proper modeling approach of early-stage microarchitectural designs for security evaluation**. For now, early-stage microarchitectural designs are usually loosely described

at a high abstraction level using human languages. However, for the purpose of security evaluation, a desired microarchitectural 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 microarchitecture 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 microarchitectural designs should allow us to use a single model to represent a space of designs that are functionally equivalent and differ in their timing behaviors. 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 microarchitectural 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 microarchitectural 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.

1.2 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 microarchitectural modules, matching the modular design procedure widely used by computer architects. We enforce a handshaking interface between different modules and further decompose each module into a functionality submodule and a timing submodule. Our modeling discipline allows designers to express functionality without worrying about timing or RTL synthesizability, and meanwhile allows capturing timing signals at cycle granularity. Furthermore, our modeling discipline allows the designers to capture a space of designs with potentially complex timing behaviors. The way we achieve this is via the usage of *uninterpreted functions*, a primitive to represent a space of functions. We extensively use uninterpreted functions inside the timing submodules, enabling the designers to focus on what factors affect timing rather than on specific implementation details.

Pensieve is a security evaluation framework built around this modeling discipline. After the user builds the microarchitectural model of a defense mechanism, the framework *automatically* evaluates its security against a *formally* defined property. If the security property is not satisfied, the framework 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 execution attacks that have not been discussed in prior work [7, 24]. Specifically, previous research has described speculative interference attacks as exploiting a code pattern where a transient "interfering" instruction causes contention with a non-speculative "transmitter" instruction that is younger in program order. However, our attack has identified a new code pattern where the two instructions *are not related in program order*. This newly discovered variant can bypass GhostMinion [1], one of the latest defenses that was previously believed to be resilient against speculative interference attacks.

Contributions In summary, this paper makes the following contributions:

- We propose a microarchitectural modeling discipline for formally studying early-stage defenses against speculative execution at-

tacks. 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.

2 BACKGROUND

2.1 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]. The computer architecture community has been actively seeking effective defense solutions. We briefly summarize the sequence of research attempts below.

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 persistent cache states, including replacement bits. If the speculative load misses in L1, it will be delayed until it is non-speculative, when either there is no branch ahead of this instruction or the instruction is guaranteed to commit.

Other invisible speculation schemes, including InvisiSpec [58], SafeSpec [32], GhostLoads [47], and Muontrap [3], make changes to multiple levels of caches. They allow speculative loads to look up 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].

In [7], the authors showed that such timing interference can result in persistent and observable changes to cache states. Their proof-of-concept demonstration uses the timing interference from transient instructions to change the ordering of executing several bound-to-retire instructions. This reordering then triggers specific conditions of the cache replacement policy and leads to measurable cache side effects.

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 microarchitectural structure is augmented with a scheduling or resource allocation policy that gives higher priority to older instructions.

2.2 Model Checking

Model checking [16] is an automatic (or "push-button") technique that can be used to check whether a given 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 [18], sequential circuits [10, 11], and micro-controllers [17]. 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.

2.3 Uninterpreted Function

An uninterpreted function is a function with specified input and output types, but with an *unspecified function body*. It represents an arbitrary function inside a space of functions. For example, an uninterpreted function written as `f: (bool,bool) -> bool` can be viewed as any function among the set of functions performing two-bit operations, such as `AND`, `OR`, and `XOR`.

This space of possible behaviors can be restricted by adding constraints on the function. For example, if we intend to restrict the previous space of functions to exclude the `AND` operation, we can add an assumption `f(x,y)!=x&y`. Overall, uninterpreted functions are maximally flexible, representing any function with any additional constraints.

Uninterpreted functions are widely used in the formal-method 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).

3 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_{pub} and secret data M_{sec} executing on a microarchitectural design μ, we denote the attacker's observation of the system as $O_\mu(P, M_{pub}, M_{sec})$. We denote O_{ISA} as an ISA emulator that serves as the non-speculative machine and O_{O3} as the target speculative out-of-order processor under study. The following property states the property of speculative non-interference for $O3$:

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

The property states that for any program that runs on two distinct secrets M_{sec} and M'_{sec}, if the program produces indistinguishable observations in the ISA model, then the observations generated by the $O3$ 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 has 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.

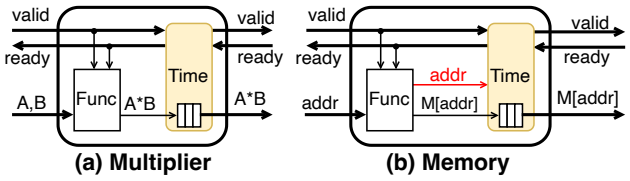

Figure 1: Modeling examples showing the hand-shaking interface and the decomposing of functionality and timing. Red texts denote additional timing-related signals that might affect the timing behavior. Highlighted submodules use uninterpreted functions inside.

4 MICROARCHITECTURAL MODELING DISCIPLINE

In this section, we describe a microarchitectural modeling discipline amenable to studying speculative execution vulnerabilities. We target early-stage microarchitectural defenses. These defenses change one or multiple modules' timing behaviors, such as delaying a load upon cache misses, and work as modular add-ons to existing designs. We start by giving a high-level idea about how our modeling discipline achieves the three desired properties in Section 1.1 via two toy examples. 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 microarchitecture.

4.1 Two Toy Examples

We illustrate our modeling approach using 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. Figure 1 illustrates the modular view of the two examples.

The two modules use a *hand-shaking interface* that consists of an input and output data channel, and each of the data channels is associated with a pair of valid and ready signals. We leverage this interface to explicitly distinguish between functionality and timing signals: we consider the input and output data channels as functionality signals and the valid and ready signals as timing signals.

We then further decompose each module into a functionality submodule and a timing submodule, and make them operate on functionality signals and timing signals respectively. For example, in the multiplier, the functionality submodule only operates on the functionality signals by taking the two operands `A` and `B` as input and computing the product. In the memory example, the functionality submodule maintains an array of data, indexes the array using the input address, and outputs the indexed data.

The timing submodules are implemented using a new modeling primitive, called an *abstract delay buffer*. An abstract delay buffer works by taking data from its input channel, adding some (potentially unspecified) latency, and forwarding the data to its output channel. Inside the buffer, we use uninterpreted functions to determine when the buffer can take in new data and when data in the buffer can be forwarded to the output.

Implementation of an Abstract Memory We provide an implementation of an abstract memory in Figure 2 to describe how the

```
1 module memory (
2   // data channel
3   input addr, output data,
4   // hand-shaking signals
5   input  Din_vld,  output Din_rdy,
6   output Dout_vld, input  Dout_rdy);
7
8   reg Mem[MAX]; wire internal_data;
9   // functionality submodule
10  assign internal_data = Mem[addr];
11  // timing submodule
12  abstract_delay_fifo (internal_data, data,
13    Din_vld, Din_rdy, Dout_vld, Dout_rdy,
14    addr); // addr is the additional
15           // timing-related signal
16 endmodule
```

Figure 2: The implementation of an abstract memory.

```
1 module abstract_delay_fifo (
2   // data channel
3   input  Din, output Dout,
4   // hand-shaking signals
5   input  Din_vld,  output Din_rdy,
6   output Dout_vld, input  Dout_rdy,
7   // additional timing-related signals
8   input  Tfactor );
9
10  // the implementation of a FIFO
11  reg buffer[MAX], head, tail;
12  always @(posedge clk) begin
13    if (Din_vld && Din_rdy) begin //enqueue
14      buffer[tail] <= Din;
15      tail <= tail + 1;
16    end
17    if (Dout_vld && Dout_rdy) // dequeue
18      head <= head + 1;
19  end
20  Dout = buffer[head];
21
22  // generate timing signals
23  reg history[MAX], cnt; //unlimited widths
24  always @(posedge clk) begin
25    history[cnt] <= {1'b1, Din_vld, Dout_rdy,
26                     WIDTH{Din_vld} & Tfactor};
27    cnt <= cnt + 1;
28  end
29  // F1 and F2 are uninterpreted functions
30  Din_rdy  = F1(history);
31  Dout_vld = F2(history) && head!=tail;
32 endmodule
```

Figure 3: An abstract single-channel delay FIFO.

functionality and timing submodules work together. In Figure 2, line 10 describes the behavior of the functionality submodule, which generates the data to be returned by the memory. Lines 12-14 describe the behavior of the timing submodule, which models an abstract buffer to determine the valid and ready signals.

To see how uninterpreted functions are used, we provide the code for an abstract single-channel delay FIFO in Figure 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 `F1` and `F2`, which take the history of cycle-by-cycle input ready and valid signals and any additional timing-related signals specified by users, denoted as `Tfactor`.

Abstract Delay Buffer Variation In the example above, we used a single-channel in-order buffer. To cover more complex designs, we provide the following variations for abstract delay buffers.

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. Specifically, we use an uninterpreted function to decide which valid entry should be selected to forward to the output channel.

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 Figure 3). The first type is used in modules with internal states, such as branch predictors. A reset only cancels the inflight branch predictions but allows previous instructions to influence future branch prediction outcomes. The second type is used in modules that do not keep internal states, such as a multiplier.

4.2 The Modeling Discipline

We now summarize our modeling discipline. Given a processor, we express the design as a modular composition of microarchitectural 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 Figure 1), we also use UFs in

functionality modules, e.g. in the fetch module to model a set of branch predictors.

Second, we use UFs to reduce modeling efforts by hiding unimportant design details. An uninterpreted function `f: input -> 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.

4.3 Covering Complex Designs with Simple Models

One of the key benefits of our modeling discipline is that we are able to *use simple models to cover a large space of complex design features*. This is achieved because using uninterrupted functions allows us to focus on *which* signals affect the timing of a design and abstract implementation details related to *how* these signals affect timing.

As an example, consider the modeling of a large family of multipliers (Figure 1(a)). We use a UF to generate the output valid signal. 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 (i.e. specifying the operands as additional timing-related signals), 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.

Moreover, 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 microarchitectural 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.

Remark Despite the benefits discussed above, using uninterpreted functions can yield a subtle limitation that the users should be aware of. The design spaces covered by the UFs are often much *larger* than what the designers intended to model and can potentially include invalid design points. For example, the multiplier model with value-based optimization allows any value to affect the computation latency, but the intended design may only allow a subset of values to affect the latency. In short, it can be challenging and tedious to precisely model a concrete microarchitectural feature using Pensieve's modeling discipline.

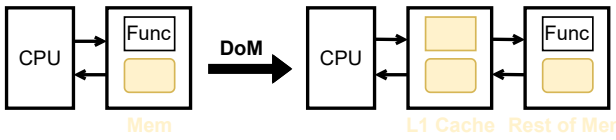

Figure 4: A modeling example of speculative execution defense. Left: a baseline memory hierarchy. Right: Delay-on-Miss. Submodules using uninterpreted functions are highlighted.

The consequence of modeling a space larger than the intended design space is two-fold. First, if the security evaluation on the larger space passes, we know for sure, every design point in this space is secure, including the designers' intended ones. Second, if a counterexample is found on the modeled space and this counterexample is triggered by some unintended design points, we call this counterexample a *spurious* one. We take a standard approach to handle spurious counterexamples, that is, we can rule out spurious counterexamples by adding constraints to UFs in our microarchitectural model.

4.4 Modeling Defenses As Modular Add-ons

With our modeling discipline, a defense mechanism, in the form of a modular augmentation to a baseline microarchitecture, 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 examples to illustrate the approaches above by showing how to model Delay-on-Miss [48]. We start by decomposing the monolithic memory module into two separate modules, i.e., the L1 cache and the rest of the memory, shown in Figure 4. 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

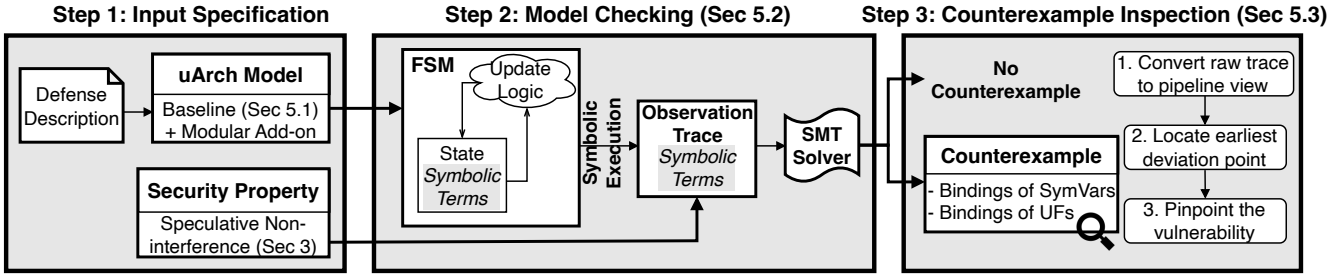

Figure 5: Pensieve Workflow.

concurrent requests.

To model DoM, we make the following changes to the L1 cache module. First, DoM disallows 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 disallows 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.

5 PENSIEVE: FRAMEWORK AND IMPLEMENTATION

Pensieve is an automatic security evaluation framework built upon the microarchitectural modeling discipline in Section 4. It uses bounded model checking to evaluate a defense scheme, encoded as a μarch model, against formally defined security properties, such as the speculative non-interference property in Section 3.

Figure 5 shows the overview of the Pensieve 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. 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

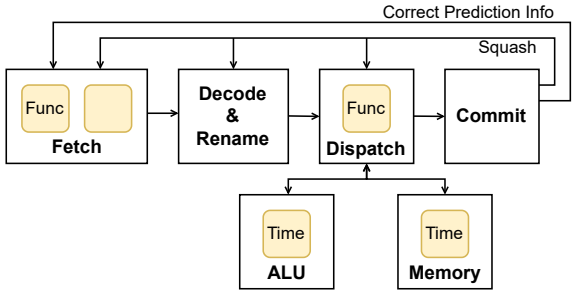

Figure 6: The baseline μarch model with the submodules using uninterpreted functions highlighted.

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.

5.1 Baseline μArch 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. The goal is to reduce the manual effort needed to construct a μarch model for the defense mechanism from scratch.

Our baseline μarch model, though simple, covers a large design space and potentially complex pipeline scheduling policies. In Figure 6, we highlight the submodules that use uninterpreted functions. The fetch module models an arbitrary branch predictor having an arbitrary fetch latency. This accounts for different latencies introduced by varied complexity of the branch predictor and varied instruction memory access latency. The dispatch module uses an uninterpreted function to select which instruction to send to the corresponding execution unit among the instructions whose operands are ready. We use the toy examples from Section 4.1 as the ALU and memory modules, which have been shown to cover a large space.

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 μarch model models a wide range of issue policies that can reorder loads, stores, branches,

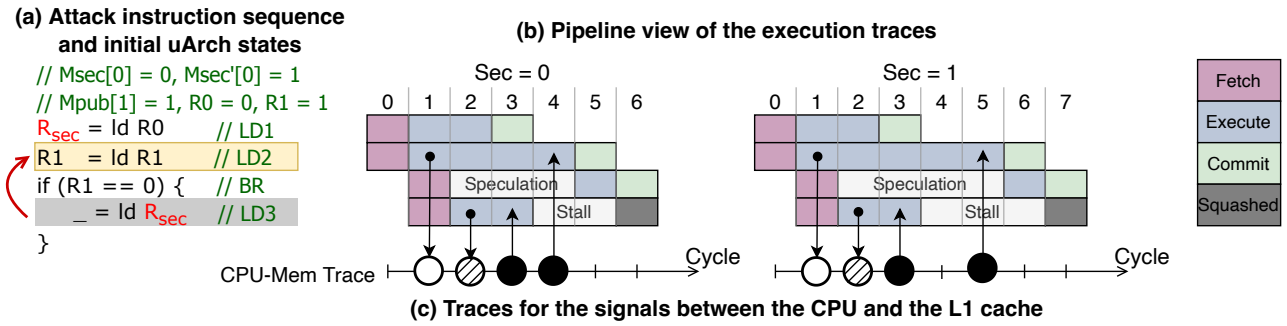

Figure 7: A counterexample generated by Pensieve for a μarch model of DoM. In the CPU-Mem trace, a hollow node indicates a memory request and a solid node is a response. The slashed nodes indicate that the memory requests differ in their addresses.

and arithmetic instructions as long as their operands are ready.

We note that our baseline μarch model has concrete decode&rename and commit stages. We show our baseline μarch model was sufficient to find vulnerabilities in existing defenses in our case studies and discuss its limitations in Section 8.

5.2 Model Checker

Pensieve automatically evaluates the security property of a μarch model using symbolic execution and bounded model checking. Step 2 in Figure 5 describes the evaluation process. First, we encode the μarch 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 μarch model. This encoding allows us to simulate the execution of symbolic instruction sequences operating on symbolic data.

Next, we execute the μarch model for a bounded number of cycles to derive a symbolic observation trace. Following the security property definition in Section 3, 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." 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 μarch 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. We evaluate Pensieve's performance in Section 7.

5.3 Counterexample Inspection

The model checker outputs a counterexample when the μarch model does not satisfy the security property. A user can inspect the counterexample to pinpoint the security vulnerability following three steps. To assist the explanation, we show the counterexample for the DoM μarch model in Figure 7.

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 μarch model introduce the secret-dependent timing variations.

The DoM counterexample Figure 7 shows the counterexample generated by Pensieve for the DoM μarch model, with the attack code sequence in Figure 7(a), pipeline view of the execution traces in Figure 7(b), and traces for the signals between the CPU and the L1 cache in Figure 7(c). We call the trace a CPU-Mem trace and align it with the pipeline execution trace. This counterexample indicates that the DoM μarch 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. The code indicates 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 looks up 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 microarchitectural 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.

5.4 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, as Pensieve can explore "worse-case" designs and executions of a μarch model.

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 priming 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.

We can use the DoM counterexample to illustrate how Pensieve accounts for the three phenomena observed in real-world attacks. For example, in Figure 7, the BR instruction is mispredicted (phenomenon 1). This misprediction is automatically generated by the model checker without needing extra training instructions. Next, the attack needs a long speculation window (phenomenon 2). The model checker finds the binding of the UF in the memory module to delay LD2, which in turn, delays the resolution time of the BR. Finally, a real-world attack needs to fine-tune the issue time of multiple memory instructions to amplify L1 contention (phenomenon 3). As the UFs in the L1 cache allow any input to interfere with the timing of its output, Pensieve's model checker can easily find the pair of inputs (different addresses of LD3) to expose the timing variation.

6 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. 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.

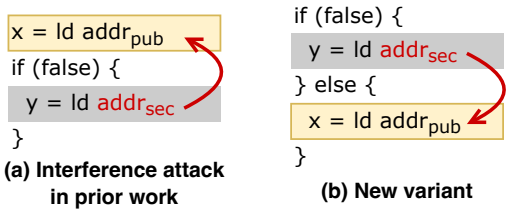

Figure 8: Comparing the attack code pattern exploited in prior work and the new variant found by Pensieve. Interfering instructions are highlighted in gray, and transmitter instructions are highlighted in yellow.

A New Speculative Interference Attack Variant Before showing the detailed security evaluation process of GhostMinion [1], let us first present 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 bound-to-commit instruction whose timing is influenced as a *transmitter instruction*.

Figure 8(a) shows the gadget pattern exploited by prior work. The interfering instruction (colored in gray) is younger than the transmitter instruction (colored in yellow) in program order. Instead, Figure 8(b) shows the gadget pattern found by our tool. There does *not* exist a program order between the two instructions, such as when the two instructions are from different sides of a branch.

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 drop any responses corresponding to squashed instructions. Such a design is widely used in modern processors for its high performance. With such a processor, after a transient instruction squashes, its memory request can still be in-flight in the memory system, clogging resources such as MSHRs, and causing contention with the memory requests issued by bound-to-commit instructions.

GhostMinion fails to achieve its desired security property because it has not considered our speculative interference attack variant. Our case study highlights the necessity of *formally* evaluating defense mechanisms, rather than reasoning about security using a limited set of known attack code patterns.

We now describe in detail how this counterexample was found by presenting our modeling of GhostMinion in Section 6.1, discussing the counterexample inspection process in Section 6.2, and proposing a security fix in Section 6.3.

6.1 Modeling GhostMinion

As with most defense proposals, GhostMinion [1] was described in human language with unavoidable ambiguity about some design details. We attempt to precisely model GhostMinion's timing behaviors and minimize ambiguities.

First, GhostMinion uses an invisible speculation scheme similar to Muontrap [3]. It allows speculative memory accesses to fetch data from all levels of caches without changing persistent cache states. The fetched data is stored in a small buffer (called GM) next to the L1 cache and can only be accessed by speculative memory requests. In our μarch model, we model both the L1 cache and the GM buffer. The L1 cache is modeled as with DoM by making cache hit/miss independent of speculative memory requests, and the GM buffer only affected by speculative memory requests. Besides, upon the commit of a speculative memory access instruction, its corresponding memory request will then be used to change the L1 cache state and thus influence future requests.

Second, GhostMinion enforces an ordering of using microarchitecture resources based on program order. The processor maintains a timestamp for each instruction. A scheduling policy is enforced at each microarchitecture structure to give higher priority to older instructions, which are associated with smaller timestamps. GhostMinion also introduces a *leapfrogging* operation, i.e., cancel an ongoing request or steal resources from the ongoing request, which is essential to enforce the ordering. This priority ordering is applied to ALU units, MSHRs in caches, and any other structures that can introduce speculative interference vulnerabilities. Note that the leapfrogging operation is an extremely complex scheme whose feasibility of being implemented is debatable. Yet, Pensieve can use a simple model to capture this complex design.

We model the priority ordering and leapfrogging operation as follows. We use an abstract delay buffer with a non-FIFO ordering 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 that the latency of a request with a smaller timestamp is independent of all the requests with larger timestamps. Such a model precisely represents GhostMinion's design by focusing on which factors affect timing and avoids all the tedious implementation details related to the leapfrogging operation.

Modeling Timestamp Generation One challenge in modeling 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 instruction 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 reset to match the timestamp of the last committed instruction. 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.

6.2 Counterexample Inspection

We run the model checker on the μarch model of GhostMinion with the Monotonic and the Reset-upon-squash timestamp generation schemes. We find counterexamples in both cases, shown in Figure 9. We mark each instruction with a timestamp before or 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. We also highlight interfering instructions and transmitter instructions with different colors.

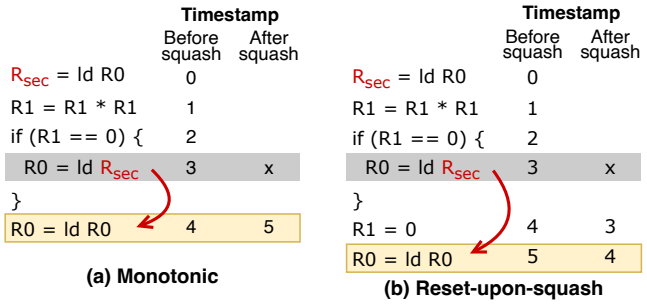

Figure 9: Attack code sequences that work on GhostMinion with vulnerable timestamp generation schemes. Interfering instructions are highlighted in gray, and transmitter instructions are highlighted in yellow.

Figure 9(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.

Pensieve generates a very similar attack code for GhostMinion using the reset-upon-squash scheme, shown in Figure 9(b). The only difference is the existence of a dummy instruction before the transmitter instruction. With such an attack code, the interfering instruction gets the same timestamp as before, and the transmitter instruction gets a timestamp of 4 after the squash. Similar to before, the transmitter instruction's timestamp is larger than the interfering instruction's, and speculative interference attacks still work.

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 that the simulator open-sourced by the authors [2] does not implement the leapfrogging operations. Instead, the simulator collects statistics about how frequently a leapfrogging operation is needed. Such 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 1 that microarchitectural 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,

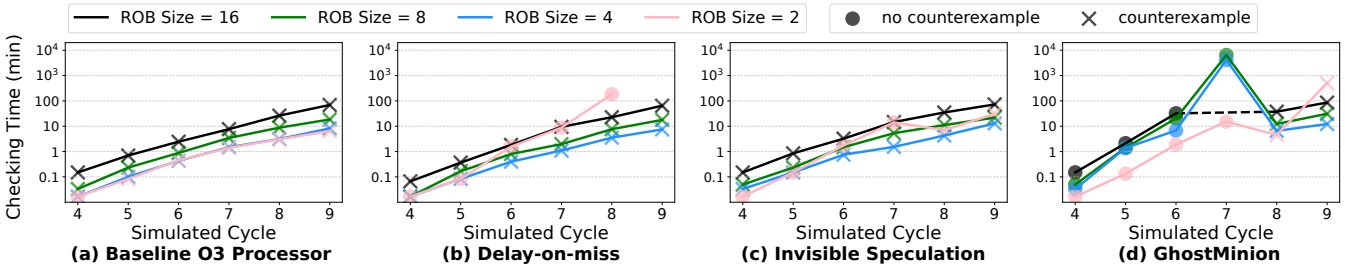

Figure 10: The time taken by the model checker to evaluate 4 μarch models with different ROB sizes and simulation cycles.

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.

6.3 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 <speculative window ID, 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 microarchitecture, where a branch 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 microarchitecture 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 microarchitecture resource usage is an effective defense mechanism to achieve the speculative non-interference property. However, it is non-trivial to have a correct hardware implementation of temporal ordering that work for a large space of processor designs. The implementation presented in the GhostMinion paper is incorrect."

7 PERFORMANCE EVALUATION

7.1 Experiment Setup

We evaluate Pensieve's performance on four μarch models: 1) the baseline microarchitecture, 2) Delay-on-Miss [48], 3) Invisible Speculation (a μarch model covering InvisiSpec [58], SafeSpec [32], GhostLoads [47], Muontrap [3]), and 4) GhostMinion [1].

These μarch models support 5 types of instructions: load immediate, load register, store register, branch, and register-to-register operation. Each model has a 4-entry 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 using ~400 and ~2400 lines of code (LOC), respectively. Adding Delay-on-Miss, InvisiSpec, and GhostMinion on top of the baseline processor required approximately 100, 100, and 250 LOC, respectively. The performance results were obtained on a server machine 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. In our μarch model, we need to decode the opcode for each instruction and then trigger different execution logic accordingly. We use the `split-case` interface in Rosette to perform concretization in decoding instruction opcodes. This optimization improves performance as it avoids monotonically accumulating constraints on a single large symbolic formula, enables evaluating multiple shorter formulas, and reduces the workload for the backend solver.

7.2 Evaluation Results

Figure 10 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 10^4 minutes (~7 days), it is considered as a timeout and will not be shown in the figure.

From Figure 10(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 μarch model with 16 ROB entries and increasing the number of simulation cycles from 6 to 7 and 8, checking time increases from 2.5 minutes to 7.5 minutes and 26 minutes, respectively.

For different defense mechanisms, it takes different numbers of simulation cycles to find counterexamples. It takes no more than 4 cycles to find a counterexample on the baseline, DoM, and InvisiSpec and at least 8 cycles on GhostMinion.

We also observe that the ROB size affects both the checking time and whether we can find a counterexample. Since the ROB size directly affects the model complexity, the checking time generally 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 Figure 10(b), when evaluating DoM, we only find counterexamples when the ROB size is at least 4.

Finally, Figure 10(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 7 and 8 cycles. This is because the tool terminates as soon as a counterexample is found, but in the case of no counterexamples, the model checker studies 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. 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 2-3 times each time we double the ROB size.

8 LIMITATIONS AND FUTURE WORK

This paper has several limitations. First, we made several simplifications to our baseline μarch model. These simplifications limit 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 microarchitectures would need to extend the μarch model by, e.g., further decomposing the concrete modules and using more abstract delay buffers.

Relatedly, 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].

9 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 1.1, 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 μhb 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 and Properties 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].

10 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 this problem. Pensieve is a security evaluation framework targeting the analyses of early-stage microarchitectural defenses against speculative execution attacks. We presented a microarchitectural modeling discipline for early-stage defenses that precisely captures timing variations due to contention and microarchitectural optimizations. We used Pensieve to automatically analyze a series of state-of-the-art defenses. Pensieve can not only synthesize known attacks, but also find subtle bugs that were not known before in complex systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was funded in part by the National Science Foundation (NSF) under grants CNS-2046359 and CCF-2217099, the Air Force Office of Scientific Research (AFOSR) under grants FA9550-20-1-0402 and FA9550-22-1-0511, MIT Research Support Committee (RSC), and gifts from Amazon and Intel.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact is the implementation of the Pensieve framework (Figure 5) for the security evaluation of microarchitectural defenses against speculative execution attacks. The artifact includes μarch models for a variety of defense mechanisms encoded in a solver-based programming language (i.e., Rosette), and scripts for performing bounded model checking on these μarch models for counterexample detection. This artifact release can be used to reproduce the two attack examples as shown in Figure 7 and Figure 9(a), as well as the performance evaluation results for the checking time as shown in the plots in Figure 10.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Docker engine and docker compose. Alternatively, for a physical machine setup, it requires Racket, Rosette, Boolector, and Dask (or HTcondor).
- **Hardware:** One 16(or more)-core machine with 2GB memory per core.
- **Output:** Terminal outputs and a figure. Expected results are included.
- **Experiments:** Running provided scripts in a provided container will reproduce the results. Results might vary due to the black-box SMT solver.
- **How much disk space required (approximately)?:** 3GB.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes.
- **How much time is needed to complete experiments (approximately)?:** On a 16(or more)-core machine, it takes ~12 hours to complete most (125 of 130) experiments and ~7 days to complete all experiments.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT license.
- **Archived (provide DOI)?:** `10.5281/zenodo.7751221`

A.3 Description

A.3.1 How to access. The artifact version designed for reproducing results in the paper can be downloaded from https://doi.org/10.5281/zenodo.7751221. The newest version of the Pensieve framework is available at https://github.com/CSAIL-Arch-Sec/Pensieve.

A.3.2 Hardware dependencies. The framework only requires CPU and memory resources. When using it to evaluate a specific μarch model configuration for a specific bounded number of cycles, it will only use 1 core. We also provide scripts to run batches of experiments for different configurations in parallel and thus recommend a 16(or more)-core machine to finish them in a shorter time. 2GB memory per core is required.

A.3.3 Software dependencies. To try out our framework or to reproduce the results in the paper, only the docker engine and docker compose are required. You can also set up physical machines for further exploration by referring to the `Dockerfile` provided. Specifically, it will guide you to install Racket, Rosette, Boolector, and Dask (or HTcondor).

A.4 Installation

Make sure that the docker engine is installed on your system and `docker-compose` command is available. The commands below will take ~10 minutes to build a container and start a `bash` inside it.

```
cd Pensieve
docker-compose up -d
docker-compose exec env bash
cd /vagrant
```

The `Pensieve` folder has been mounted at `/vagrant` in the container. All commands below should be run when `pwd` is this folder. After the experiment, running `docker-compose down -rmi all` can clean up everything.

A.5 Evaluation and expected results

Reproduce the Attack Example in Figure 7 Running the command below will generate terminal output expected to be similar to `result/DoM.log` with ~10 seconds. It will only use 1 core.

```
raco test \
  ++arg --param-saved-params ++arg DoM \
  ++arg --param-saved-sizes ++arg DoM \
  src/main_veriSpec.rkt
```

To understand the terminal output, the "`Finish SMT Result Evaluation`" splits the output into two parts. The output below it summarizes the attack instruction sequence and the initial μarch model states, which are used to draw Figure 7(a). The output above it includes detailed execution traces of running the attack program with two different secrets, which are used to draw Figure 7(b)(c). Please refer to `README.md` for more documentation on the terminal output and how to customize the output to get more information about the execution traces.

Reproduce the Attack Example in Figure 9(a). Running the command below will generate terminal output expected to be similar to `result/GhostMinion.log` with ~10 minutes. It will only use 1 core.

```
raco test \
  ++arg --param-saved-params ++arg GhostMinion \
  ++arg --param-saved-sizes ++arg GhostMinion \
  src/main_veriSpec.rkt
```

The terminal output is organized the same as the last one. The attack instruction sequence is used to draw Figure 9(a). The execution traces can help you understand the steps of the attack.

Reproduce the Performance Evaluation in Figure 10. Running the commands below will reproduce a figure expected to be similar to `result/Figure10/performance.pdf` with ~7 days. However, most (123 out of 128) experiments will finish with ~10 hours, which are enough to plot most data points in the figure. The experiments will use all cores on the machine by default. You can also limit the core number to 16 by editing `script/run_helper_local.py` (i.e. commenting `Line 8` and uncommenting `Line 9`). After ~10 hours, only 5 cores will be busy for the rest 5 experiments.

```
# You will reproduce two files in this folder
cp -r result/Figure10 result/artifact_eval
rm result/artifact_eval/summary.json
rm result/artifact_eval/performance.pdf

# This command will take ~7 days in background
(python3 -u result/artifact_eval/run_local.py \
  > terminal.log 2>&1 &)

# Plot a partial figure after ~10 hours
python3 result/artifact_eval/summary.py
python3 result/artifact_eval/plot.py

# Re-run two lines of commands above
# after ~7 days for a whole figure
```

Running the commands above will generate the following file: `result/artifact_eval/performance.pdf` This file is expected to be similar to Figure 10. Specifically, at each configuration, the reproduced result should be consistent with Figure 10 on whether a counterexample exists. However, the checking time and whether a configuration runs timeout after 7 days can vary due to the implementation of the black-box SMT solver and the performance of the machine.

In addition, raw numbers shown in the figure are saved in `result/artifact_eval/summary.json` and can be printed on the terminal by running `python3 result/artifact_eval/print.py`.

A.6 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM. https://doi.org/10.1145/3466752.3480074

[2] Sam Ainsworth. 2022. Artefact Evaluation for GhostMinion. https://github.com/SamAinsworth/reproduce-ghostminion-paper. Accessed 26 October 2022.

[3] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-like Attacks by Capturing Speculative State. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. https://doi.org/10.1109/ISCA45697.2020.00022

[4] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association.

[5] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. https://doi.org/10.1109/PACT.2019.00020

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. https://doi.org/10.1007/978-3-030-99524-9_24

[7] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3445814.3446708

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM. https://doi.org/10.1145/3319535.3363194

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* (2011). https://doi.org/10.1145/2024716.2024718

[10] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. 1986. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Trans. Comput.* (1986). https://doi.org/10.1109/TC.1986.1676711

[11] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (1994). https://doi.org/10.1109/43.275352

[12] Cadence Design Systems, Inc. 2022. Jasper RTL Apps. http://www.jasper-da.com/products/jaspergold-apps. Accessed 26 October 2022.

[13] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3385412.3385970

[14] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. https://doi.org/10.1109/CSF.2019.00027

[15] Md Hafizul Islam Chowdhuryy and Fan Yao. 2021. Leaking Secrets through Modern Branch Predictor in the Speculative World. *IEEE Trans. Comput.* (2021). https://doi.org/10.1109/TC.2021.3122830

[16] Edmund Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking, second edition.* MIT Press.

[17] E.M. Clarke, D.E. Long, and K.L. McMillan. 1991. A language for compositional specification and verification of finite state hardware controllers. *Proc. IEEE* (1991). https://doi.org/10.1109/5.97298

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* (1986). https://doi.org/10.1145/5397.5399

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag.

[20] David L. Dill. 1996. The Murphi verification system. In *Computer Aided Verification*. Springer Berlin Heidelberg.

[21] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. https://doi.org/10.1109/DAC18072.2020.9218572

[22] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. 2019. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. https://doi.org/10.23919/DATE.2019.8715004

[23] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Muller, Jorg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2022. An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors. *IEEE Trans. Comput.* (2022). https://doi.org/10.1109/TC.2022.3152666

[24] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*. ACM. https://doi.org/10.1145/3411504.3421216

[25] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. 2021. IntroSpectre: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. https://doi.org/10.1109/ISCA52012.2021.00073

[26] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[27] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM. https://doi.org/10.1145/3372297.3417246

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-

software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP40001.2021.00036

[29] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP40000.2020.00011

[30] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3466752.3480087

[31] Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture Verification by Compositional Model Checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*. Springer-Verlag.

[32] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM. https://doi.org/10.1145/3316781.3317903

[33] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2019.00002

[35] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association.

[36] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[38] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

[39] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. https://doi.org/10.1145/3243734.3243761

[40] KL McMillan and J Schwalbe. 1992. Formal verification of the Gigamax Cache Consistency Protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*.

[41] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

[42] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM. https://doi.org/10.1145/3470496.3527412

[43] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. https://doi.org/10.1145/3341301.3359641

[44] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* (2014). https://doi.org/10.3233/sat190101

[45] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. ACM. https://doi.org/10.1145/3503222.3507729

[46] Gururaj Saileshwar and Moinuddin K Qureshi. 2019. Cleanupspec: An" undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[47] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost Loads: What is the Cost of Invisible Speculation?. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM. https://doi.org/10.1145/3310273.3321558

[48] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM. https://doi.org/10.1145/3307650.3322216

[49] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I*. Springer-Verlag. https://doi.org/10.1007/978-3-030-29959-0_14

[50] Ryota Shioya. 2022. Konata. https://github.com/shioyadan/Konata. Accessed 26 October 2022.

[51] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM. https://doi.org/10.1145/2509578.2509586

[52] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. https://doi.org/10.1109/MICRO.2018.00081

[53] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/TSE.2019.2953709

[54] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

[55] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358306

[56] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3314221.3314647

[57] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *Network and Distributed System Security Symposium*.

[58] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/MICRO.2018.00042

[59] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction for Safe and Efficient Speculative Execution. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE Press. https://doi.org/10.1109/ISCA45697.2020.00064

[60] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358274