SpecCheck: A Tool for Systematic Identification of Vulnerable Transient Execution in gem5

by

Zack McKevitt

B.S., University of Colorado Boulder, 2022

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2023

Committee Members:

Tamara Lehman, Chair

Ashutosh Trivedi

Qin Lv

McKevitt, Zack (MS, Computer Science)

SpecCheck: A Tool for Systematic Identification of Vulnerable Transient Execution in gem5

Thesis directed by Prof. Tamara Lehman

Speculative execution attacks leverage a processor's speculative execution optimization to leak secret information. Previous attempts to generalize transient execution attacks often analyze specific gadgets in software or look solely at microarchitectural state artifacts to explain the fundamental logic behind these attacks. In this work, we present SpecCheck, a systematic security verification for detecting potential transient data leakage. SpecCheck is based on a description of a generic transient execution attack in the form of a register based Finite State Machine (FSM) that is easily incorporated into commonly used processor simulators. SpecCheck's key insight is the fact that transient execution attacks involve both the software and the hardware to succeed and the only way to verify if a design is capable of mitigating such attacks is by considering both at verification time. As a proof of concept, we implement SpecCheck's FSM in the gem5 simulator to check for suspicious program flows during an arbitrary program's simulation and lay the groundwork for a robust and systematic hardware security verification tool. We show that SpecCheck is able to identify known transient execution gadgets in four of the main Spectre variants while incurring on average only a 4% simulation time overhead.

Dedication

This thesis is dedicated to my parents and their unwavering support during my academic career. Their help throughout my BS and MS degrees has been crucial to my success as an academic, and I would not have been able to write either of my theses without them. I will always be extremely thankful for the opportunities they have provided me, academically or otherwise. Thank you both.

Acknowledgements

I would like to acknowledge my research advisor, Tamara Lehman, for being a fantastic mentor and teaching me everything I know about computer architecture and security throughout my past 3 years in the CU Architecture Research Lab . I would also like to thank the members of my committee, Ashutosh Trivedi and Qin Lv, for supporting and helping supervise both my BS and MS theses. I also want to thank my awesome lab mates/friends, Kidus, Jackson, and Phaedra, for making each day in the lab extremely fun and still (mostly) productive, as well as the rest of the Architecture Lab for always being willing to collaborate and help out when needed. Lastly, I want to thoroughly acknowledge my amazing parents, without whom this thesis would not be possible. Thank you to everyone for supporting me through this journey!

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Transient execution attacks are a new family of complex exploits that leave most modern processors vulnerable to exploitation [44, 48, 11, 79]. These types of attacks leverage speculative execution to transiently **expose** confidential information into microarchitectural state and then **uncover** the resulting microarchitectural state to visible state to infer the secret information. The first variant of the Spectre family of attacks, one of the first examples of a transient execution attack, exploits the branch predictor to mislead the processor into accessing unauthorized memory that uses the value of a secret as an index into an array which is loaded into the cache [44]. Cache state, which is not part of the visible state and thus not flushed after a branch misprediction, is then leveraged via a cache timing attack to reveal the secret. By probing the cache, the attacker learns the secret value by investigating which index of the array experiences a cache hit (shorter access time). As this family of exploits directly targets effective processor optimizations, such as speculative execution and branch prediction, almost all modern processors are vulnerable to this type of attack.

These types of complex vulnerabilities are difficult to verify at design time because they require both software and hardware to work in unison to succeed. On one hand, they require the vulnerability in the processor in the form of the ability to execute instructions transiently and to allow these transient instructions to affect microarchitectural state that is not cleaned up after a mis-speculation is detected. On the other hand, these attacks depend on carefully crafted software instructions—or gadgets—to leak sensitive information. In other words, if viewed in isolation, the hardware design is functioning correctly, as it is not going to impact the normal execution of a benign program; similarly, the software is seemingly doing what it is expected and not accessing an illegal memory location (**i.e.** `if (x < sizeOfArray) array[x]`). Since the key to the

success of transient execution attacks is the ability to exploit microarchitectural vulnerabilities with crafty software gadgets, verification mechanisms must integrate software and hardware features to be effective.

Detection mechanisms for security verification have predominantly focused on either hardware or software features in isolation, leaving prior solutions unequipped to identify a large number of transient execution attacks. For example, state-of-the-art software security verification tools are insufficient to combat this type of vulnerability because of their inability to consider the hardware state in their reasoning [57, 14, 28, 68, 67, 80]. Similarly, hardware security verification tools consider vulnerabilities to be solely in the hardware and fail to consider atypical use cases in which these vulnerabilities are exploited [52, 53, 4, 35, 36, 84].

Recently proposed work that combine software and hardware state to evaluate security are difficult to use for non-security experts. For example, CheckMate [70], which is an effective platform to identify potentially vulnerable regions of code, requires the user to formally describe the details of microarchitecture and the vulnerability in the μSpec Alloy Domain Specific Language (DSL). While effective, CheckMate makes security evaluation difficult for non-security experts who may just be wondering how their new microarchitecture interacts with transient execution attacks. InSpectre [27] is another example of a formal verification tool for secure speculative execution. InSpectre uses a machine independent language (MIL) to define the microarchitectural details to investigate. Once again, while effective, the requirement to use the MIL makes it difficult for non-security experts to adopt. Our approach looks to complement these tools for security experts and to supplement non-security experts the ability to evaluate security without additional efforts.

To improve the detection accuracy and coverage of security verification tools for transient execution attacks, we propose SpecCheck, a finite state machine based classifier to explicate common patterns in Spectre-like vulnerabilities in commonly used processor simulators. SpecCheck is a systematic approach, based on register finite state machines, to analyze program code execution at runtime based on both hardware and software states to identify patterns of speculative execution vulnerabilities. SpecCheck is implemented on a widely used processor simulator, gem5 [5], and is easily adaptable to other processor simulators to help with wider adoption of the approach.

The key design principle behind SpecCheck is that a general pattern of transient execution attacks,

including similar yet undiscovered ones, is inherent in the software and hardware traces of the system under attack with different Spectre-like variants. While mitigation approaches use similar techniques [76, 51, 82, 16, 81] to identify conditions of a transient execution attack, the property that makes SpecCheck novel is the use of register finite state machines to abstract away the details of the attack to widen the coverage of the tool. In addition, prior approaches modify the microarchitecture to mitigate the attacks at runtime, while SpecCheck is a systematic verification tool to be used at design time to verify that these mitigation approaches indeed are effective. To the best of our ability, no prior work has considered using this approach to construct a security verification tool to identify potentially vulnerable microarchitectural designs.

In addition, what makes SpecCheck a unique verification tool is that it does not require any new environment or tool to do a security verification. We posit that commonly used tools in computer architecture research (such as processor simulators) can give us enough information to consider both hardware and software state simultaneously. SpecCheck detects suspicious program flows in arbitrary programs that indicate potential transient execution vulnerabilities in a processor simulator. By tracking each instruction through the processor pipeline, we maintain an internal state to analyze a program's execution flow at each instruction to determine the presence of a speculative execution vulnerability. While the current implementation of SpecCheck is tailored towards the Spectre family of attacks, we envision the future framework to be agnostic to the exact kind of transient execution vulnerability.

Our initial experiments suggest that SpecCheck is able to correctly identify multiple known gadgets across the four main unmitigated Spectre variants: Spectre variant 1 [44], Spectre Variant 2 [44], SpecRSB [46] and Spectre variant 4 [33, 32]. The SpecCheck framework is intended to help both software developers and computer architects to evaluate their own designs against a spectrum of vulnerabilities by using our tool to identify potentially vulnerable implementations.

One of the key motivations for SpecCheck is to enable a simple way for non-security and security experts alike to have easy access to a systematic security evaluation tool that will aid the future of security evaluations. As Spectre and Meltdown have shown, performance optimizations need to have security as a design constraint, as many of these optimizations may compromise previously mitigated vulnerabilities. The difficulty of accessing security evaluation tools, makes security a second-class citizen, creating a never

ending cycle of vulnerabilities and mitigation.

Prior work in security evaluation tools require users to re-implement designs in new programming languages or environments, making their wide adoption problematic. While security experts absolutely need these additional tools to formally verify security guarantees, non-security experts also need to be aware of the security implications of their designs. SpecCheck is a tool that seeks to break this entry barrier to non-security experts to make more thorough security evaluations with tools already at their disposal.

This paper presents the following contributions:

(1) A novel, systematic approach to identify potentially vulnerable hardware implementations by combining software and hardware state at runtime via a register finite state machine.

(2) An easily accessible tool for non-security experts to perform more robust security evaluations for transient execution attacks.

(3) A mechanism to identify potentially vulnerable regions of code executed on top of a specific microarchitectural implementation.

(4) An implementation of the systematic finite state machine in the gem5 simulator.

Chapter 2

Background

This section provides an overview of the key concepts required to understand transient execution attacks, as well as an overview of the tools used in the proposed framework. SpecCheck is designed to identify all variants of transient execution attacks by leveraging microarchitectural state to identify temporally sequenced patterns. The proposed framework takes advantage of the software and hardware state information available in a processor simulator, gem5—a popular open source processor simulator [5].

While the method of exploiting prediction to speculatively execute loads is common to all transient execution attacks (including the Meltdown family of attacks) [44, 9, 55, 11, 48], the prediction mechanisms vary across variants. Several other variants have been discovered that are able to use other microarchitectural components besides the branch predictor, the Return Stack Buffer and the Memory Disambiguator, to transiently execute a load instruction that accesses the secret data [60, 73, 59, 77, 72, 44, 55]. Similarly, the method for disclosing the secret has also been shown to leverage different microarchitectural states besides the cache state, such as the line fill buffer and the floating point unit usage [73, 44]. Given these differences, SpecCheck was designed to be independent of the speculation method and the disclosing method to increase the coverage of the verification.

2.1 Out of Order Execution and Speculation

Modern processors make use of Out of Order (OoO or O3) processing and speculative execution to improve overall CPU efficiency by executing instructions that *may or may not be needed* when they are ready to be executed [31, 69]. These optimizations allow the processor to continue executing while waiting on

various data dependencies at each stage of the pipeline. In an OoO pipeline, instructions can execute as soon as their input operands are ready, i.e. all previous dependencies have completed their execution, before completing themselves. Once an instruction completes, the result of its computation can be forwarded to other dependent instructions down the pipeline, even if instructions are not executed in strict program order. However, instructions are committed to the pipeline in order [37].

Speculative execution allows for instructions to be accessed *speculatively*, meaning that the processor will predict whether or not the instructions will be executed in the future. One particularly common use case, specifically in relation to Spectre variant 1 [44], is the branch prediction unit, which uses program history to predict whether or not conditional branches should be taken before the branching conditional can be fetched from memory and verified. When the speculated code path is incorrect, i.e. misspeculated, the results of the speculative execution need to be flushed from the pipeline and rolled back to a correct state; instructions within this misspeculation window are called *transient*. When the speculative code path is correct, all speculative instructions are committed (or retired) to the pipeline once the speculation is resolved.

2.2 Transient Execution Attacks

Despite speculative execution being functionally correct, transient execution attacks target the microarchitectural states that are not flushed during a mis-speculated event. Transient execution attacks rely on microarchitectural side-channels to transmit secret information [9, 44, 48, 72, 77, 73, 59, 60]. The secret is transmitted through *transiently* executed instructions that are destined to be flushed from the pipeline. In most cases, these vulnerabilities rely on timing differences required to access different data values, but other types disclosing gadgets exist [75, 50]. As side-channels of this type arise from standard hardware behavior, transient execution vulnerabilities are not only difficult to mitigate, but nearly all current mitigation techniques either introduce a significant amount of overhead or do not protect all vulnerable microarchitectural structures [9].

In combination with non-optimal mitigation techniques, the ever increasing number of discoveries of new types of transient execution vulnerabilities makes it imperative that we develop tools to allow developers (both software and hardware) to identify the patterns that enable these attacks.

```
1 if (x < array1.size()) {       // speculative branch
2     secret = array1[x];        // restricted access
3     y = array2[secret*4096];   // transmitter
4 }
```

Figure 2.1: Example Victim Code for a Spectre Variant 1 Attack.

2.2.1 Spectre Variant 1

The first transient execution attack published, the Spectre variant 1, leverages the branch predictor to direct branches to a desired branch direction [44]. A Spectre variant 1 victim gadget [44] is shown in Figure 2.1. The victim gadget is composed of a conditional branch (line 1), a memory access (line 2), and a transmitter (line 3). In this gadget, a Spectre attack will first train the branch predictor to always predict that the branch on line 1 be taken. During the second phase, the processor will speculatively execute instructions on lines 2 and 3 before the conditional branch is resolved. The attacker will pass in a value for x that when added to the base of array1 will match the address of a secret value. The access of `array1` at index `x` (when `x` > `array1.size()`) will be transiently executed but then flushed from the pipeline due to a branch mis-prediction event. However, the `array2` access on line 3 will be transiently executed as well before both instructions are flushed, placing the secret value in the cache via the indexing trick shown on line 3. The attacker, having flushed all indices of `array2` from the cache prior to executing the victim gadget, can then access every index of `array2`, measuring the time to access it, to determine which index is in the cache. The index that takes the least amount of time is the one that was accessed during the attack phase, revealing the secret value implicitly. Hence, this Spectre attack takes advantage of load instructions that *complete* but do not retire as a result of branch mis-prediction.

2.2.2 Spectre Variant 2

The second variant of this original Spectre attack, called the Branch Target Injection (BTI) attack, poisons the branch target buffer (BTB) to induce speculation during direct and indirect jumps instead of the PHT like the first variant [44]. Spectre v2 is a variation of a Return Oriented Programming (ROP) attack,

where an adversary is able to inject a malicious address into the BTB to ensure that transient control flow is directed to a malicious gadget (or multiple gadgets) within the victim address space. The rest of the attack, such as use of a covert channel to recover secret information, is the same as the first variant.

2.2.3 Spectre RSB (Variant 3)

A third variant, SpectreRSB [55] (or SpecRSB), exploits the Return Stack Buffer (RSB) rather than the PHT or BTB. The RSB is a microarchitectural structure that stores virtual memory addresses as return values every time a `call` instruction is invoked, and the address at the top of the RSB is popped when a `ret` instruction is seen. When the RSB is full, the processor needs to speculate on the return address. A mis-speculation using the RSB occurs when the RSB differs from the actual memory addresses returned by the stack, and as the RSB occupies a fixed size, an attacker can induce a mismatch between RSB and real stack addresses by deeply nesting functions inside of each other to overload the RSB.

2.2.4 Speculative Store Bypass (Variant 4)

A fourth variant, Spectre v4 [11], known as Store to Leak Forwarding or Speculative Store Bypass, does not rely on transient control flow to leak secret data, but rather takes advantage of transient data flow (Store to Load (STL) Forwarding, the memory disambiguator [63]) to transiently leak secrets. STL forwarding is a memory disambiguation technique that allows loads to be speculatively executed ahead of in-flight stores assuming the load accesses a different memory location than the store. With STL forwarding, loads may read stale values from the cache ahead from older stores in the memory ordering buffer (MOB) to avoid waiting for the longer memory access of the store to complete. An attacker can leverage this transient execution to load a stale value in the cache, bypassing the older store to the same location. The processor eventually re-executes the load to read the correct value and flushes any incorrect state from the pipeline. However, the stale data from the transient load can be used to reveal sensitive data through a transmitting side-channel (such as the cache state which is not flushed during a mis-speculation event).

2.3 Current Mitigation Techniques

Mitigation for all variants of Spectre attacks at both the hardware and software levels are currently too expensive to become the standard within kernels or microarchitectures today. While the Linux community has successfully released OS-level mitigations for Meltdown [26], the Spectre family still affects a vast majority of processors in use today. One popular software-based mitigation technique, shown in Figure 2.2, is to use a serializing *load fence* [65]. The load fence instruction (line 4 in Figure 2.2) ensures that all loads prior to the fence retire before issuing younger instructions. Load fences are a popular software level mitigation for Spectre vulnerabilities as they are easy to implement and eliminate any Spectre vulnerabilities. However, implementing load fencing incurs a drastic performance overhead.

RetPoline [71] is a more efficient software based mitigation to isolate indirect branches from speculative execution, targeting the Spectre BTB variant. Retpoline redirects indirect branches to code regions that make use of the Return Stack Buffer (RSB) in order to leverage an ISA's return instruction as the speculation primitive rather than the BTB. However, in recent work, the RSB has been shown to also be exploitable via speculative execution with SpecRSB [55], and more recently RetPoline has been shown to be ineffective against a novel Spectre variant RetBleed [78].

On the hardware side, many defenses have been proposed and only a few can defend from multiple attacks with smaller performance overheads. For example, NDA [76] and DOLMA [51] instrument the Reorder Buffer (ROB) to prevent data propagation of potentially unsafe instructions. Both of these defenses achieve more reasonable overheads (compared to fences). STT [82] is another hardware defense mechanism, in which the pipeline is instrumented to track data propagation and limit the effects of the data produced during speculative windows. STT results in performance overheads in the order of 14% which is an order of magnitude better than most prior defenses. While these hardware approaches succeed at defending against several of the Spectre variants with reasonable performance overhead, they fail in addressing the vulnerability at its source because of the lack of formalization around the attack. SpecCheck is a systematic verification mechanism that will aid the security evaluation of these mitigation approaches to improve the understanding of the vulnerability at its source and to identify potentially vulnerable designs that may not

```
1 #include <x86intrin.h>
2
3 if(x < array1.size()) {
4     __mm__lfence(); // load fence
5     secret = array1[x];
6     y = array2[secret * 4096];
7 }
```

Figure 2.2: Example victim gadget for a Spectre variant 1 with a load fence as a mitigation (line 4).

have been considered vulnerable before.

2.4 The gem5 Processor Simulator

To be able to consider both software and hardware state in evaluating vulnerabilities, we look at a popular processor simulator used in computer architecture research, gem5. The gem5 [5] simulator is a full system, cycle accurate processor simulator that models various instruction set architectures and provides detailed pipeline information at simulation runtime. The specific information used by SpecCheck is encapsulated by the dynamic instruction object widely used throughout the simulation.

The dynamic instruction object contains most of the information required by the SpecCheck FSM, including the state of the instruction through the pipeline, the program counter corresponding to the instruction, as well as the corresponding micro-ops for a macro instruction. The dynamic instruction object also includes both the renamed input registers and the output register to enable the tracking of the potentially vulnerable data. The use of dynamic instructions gives SpecCheck an omnipotent view of an instruction's lifetime throughout the execution of the program, and easily allows us to reason about hardware and software state in combination.

The gem5 simulator provides another instruction API, static instructions, that provide static information about each instruction. Unlike dynamic instructions that track information about an instruction throughout its lifetime in the processor pipeline, static instructions tell us the type of instruction, e.g. whether it is a load or store, the source and destination architectural registers, and the various microoperations associated with a single macro operation. The static instruction API is especially useful for determining whether or not an

instruction produces microarchitecturally visible effects.

The gem5 simulator uses the DynInst API to direct the execution through the pipeline stages. The last stage is the commit stage which is where instructions exit the pipeline. The commit stage considers instructions in order as this is the architecturally visible point. gem5 forces all instructions (flushed or not) through this stage. Given the in-order and comprehensive nature of this commit stage implementation, this region of the code is where the SpecCheck logic is inserted. A more thorough description of the SpecCheck implementation in gem5 is given in section 3.

Chapter 3

SpecCheck

To identify potentially vulnerable microarchitectures, one needs to evaluate software and hardware in combination. We propose to look at both the instruction sequence as well as the pipeline state at runtime, leveraging the richness offered in processor simulators to determine if a specific microarchitectural implementation running a specific region of code could potentially be leveraged for a transient execution attack. We propose that a potentially vulnerable flow of execution can be identified by flagging mis-speculated windows that contain a completed but not retired load instruction through a series of conditions explained in a register finite state machine.

SpecCheck is a register finite state machine abstraction of a transient execution attack designed to detect potential vulnerable code segments for a specific microarchitectural implementation. To achieve this generalized description of the attacks, SpecCheck focuses entirely on mis-speculated windows that contain potential transmission channels regardless of the method used to induce speculation or to disclose the secret.

The key insight of the SpecCheck framework is to track all data dependencies within each mis-speculated window to determine the possibility of a potential data leak later in the pipeline. SpecCheck records the destination registers of each load operation in the mis-speculation window that completes and analyzes each subsequent instruction to determine the presence of a data dependency on one of these destination registers. Any microarchitecturally visible, μV, instruction that contains a data dependency on a mis-speculated load (directly or indirectly) is deemed as a potential data leak, and SpecCheck will flag all instances of potential data leaks as vulnerable.

3.1 Problem Definition

As explained in Section 2, a transient execution attack can be identified if it satisfies the following conditions:

(1) It occurs during a flushed transient execution window

(2) The transient window contains least one completed load instruction.

(3) A microarchitecturally visible instruction uses a tainted register (directly or indirectly) from one of the completed load instructions.

Given the richness of information within a processor simulator, the proposed mechanism is able to examine the pipeline at each cycle to determine whether or not these conditions exists that would deem the execution flow vulnerable to transient execution attacks.

A transient execution attack is first made possible by a load instruction that has completed but not retired. A completion without retirement implies that this instruction has been flushed from the pipeline and is possibly vulnerable to leaking information through microarchitectural side-channels. This microarchitectural side-channels can be made visible through other instructions that may also have microarchitectural side-effects.

Once an instruction uses the destination register of the completed, but not retired, load as an input source and the consuming instruction is considered to be microarchitecturally visible, then we flag this speculation window as being vulnerable. However, if the flushed sequence of instructions do not contain any microarchitecturally visible instructions we conclude that the instruction sequence does not have a vulnerability.

To determine if instructions within the mis-speculated window have any microarchitectural visibility, we rely on the Instruction Set Architecture (ISA) to include this information as part of the software contract as explained by prior work by Mosier **et.al.** [56]. The current implementation of SpecCheck identifies commonly known instructions that are microarchitecturally visible such as branches (control flow state), memory operations (memory system state) and floating point operations (functional unit state). As per prior

work in Information Flow Tracking (IFT) [64, 47, 19], the vulnerable state of a register is propagated through instructions regardless of whether they are considered to be microarchitecturally visible. This propagation mechanism also includes data dependencies through memory. For example, a store instruction that places some value in memory from a tainted register is considered to be a microarchitecturally visible instruction and thus SpecCheck will flag such a sequence of instructions as being vulnerable.

The linear input stream of instructions encountered throughout a program's lifetime is also representative of a classical string parsing problem typically defined by other finite state automata; by treating each instruction and its corresponding microarchitectural state as a single symbol in a sequence of instructions within a program, a finite state machine approach is able to make simple logical determinations about the nature of a program based solely upon the next instruction symbol when augmented with a taint table storage structure described in section 3.2 to retain memory of prior behavior in a program's lifetime. As we generalize a transient execution attack as a sequence of symbols that satisfy the properties described in this subsection, we believe that treating an arbitrary program as a stream of instruction symbols is sufficient enough to detect all code regions that may be vulnerable to such attacks.

A finite state machine approach was chosen for SpecCheck as it is both scalable and efficient, requiring slightly more computational power than a traditional finite automaton for regular expression parsing. As the number of states and transition functions are kept constant, SpecCheck incurs minimal overhead.

3.2 Finite State Machine Representation

We propose a 4-state automaton (shown in Figure 3.1) to classify mis-speculated code paths as vulnerable to speculative execution attacks. The machine begins in an initial state, q_{ini}, and proceeds through state transitions by consuming an instruction as it moves through the pipeline. Upon reaching the accept state, q_{acc}, the machine automatically reverts back to the initial state without consuming an additional instruction to account for additional speculation windows.

The SpecCheck FSM augments a simple finite state automaton by including a taint table that stores all possible data dependencies that are the destination of a tainted instruction during the mis-speculation window by tainting its destination registers. When the accept state is reached or the pipeline resolves speculation, the

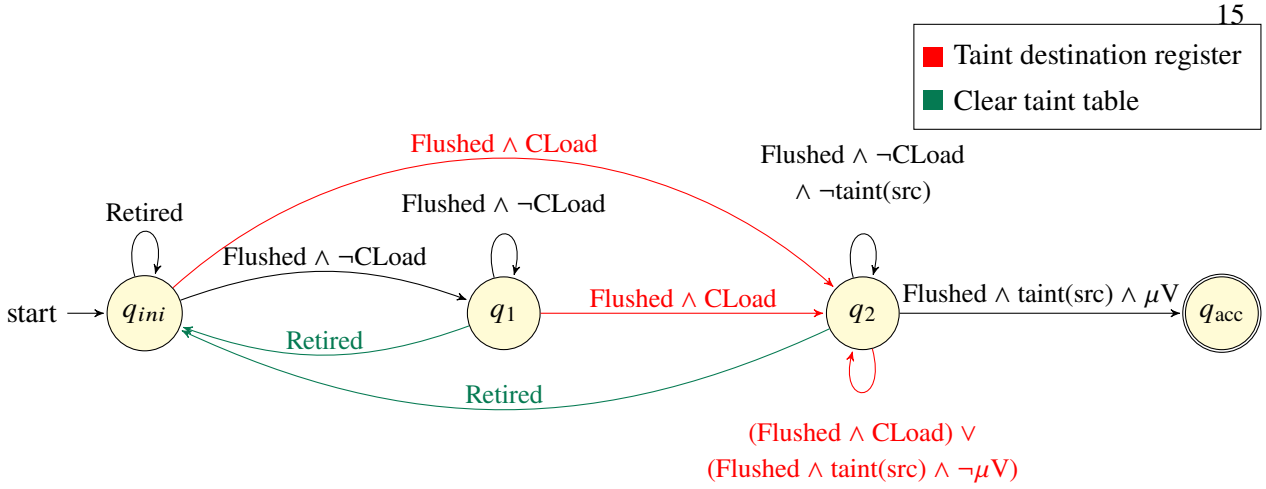

Figure 3.1: The Finite State Machine representation for SpecCheck.

taint table is reset.

3.2.1 The Instruction Tuple

SpecCheck is designed as a monitor to the Reorder Buffer (ROB) that consumes instructions in the order they they are removed from the ROB. For any given instruction, SpecCheck evaluates: (1) the instruction itself, (2) the Program Counter (PC), (3) the *renamed* source and destination registers, and (4) the state of the instruction in both the complete and commit stages of the pipeline. The tracking of software based information, such as the instruction and program counter, as well as hardware information, such as microarchitectural registers and pipeline state, allow SpecCheck to identify side-channel gadgets by tracing data dependencies through the pipeline.

3.2.2 The Taint Table

SpecCheck is further augmented with a taint table to track memory dependencies between instructions as information flows through the pipeline. The maximum size of this register array depends on the number of physical registers that may be used as either destinations or sources to instructions seen by the SpecCheck FSM. Registers are tainted any time they are the target of a flushed load instruction that still reaches the complete stage in the pipeline. The target of non-load instructions are tainted if either of the source registers

are present in the taint table and the instruction completes, thus propagating its taint to dependent instructions. The taint table automatically untaints all registers any time the FSM returns to the initial state, **i.e.** when the transient state is resolved.

3.2.3 The Transition Function

As SpecCheck consumes each instruction, it may:

- Modify the current state
- Modify the taint table

Each state transition is determined by the current instruction and may depend on information provided by the taint table. In addition to changing state, the state machine may also add registers to the taint table, or clean the table when speculation is resolved. As shown in Figure 3.1, this transition logic is denoted by the instruction itself, its final pipeline stage, and the dependence upon a tainted register. Each of the tasks that may be taken in addition to a state transition are each denoted with a different color. We introduce the `CLoad` primitive to denote a completed load instruction as loads that do not start the execution (denoted in gem5 as the completion of the instruction) cannot create microarchitectural side channels on their own. Additionally we introduce the `taint(register)` primitive to represent a lookup in the taint table, where `taint(reg)` indicates that `reg` is tainted and `¬taint(reg)` indicates that `reg` is not tainted.

In Figure 3.1, modifications to the taint table are denoted by different colors: red adds the destination register of the current instruction to the taint table and green resets the taint table. When an instruction reaches the accept state, there is an implied transition back to the initial state, clearing the taint table, without needing to consume an additional instruction.

SpecCheck identifies any mis-speculated code region as vulnerable if it adheres to each of the principles listed in Section 3.1. SpecCheck will only accept a mis-speculated code region as vulnerable if it satisfies all principles and will immediately return to the initial state upon breaking any of these principles. Specifically, SpecCheck will reject any mis-speculated code region that does not contain a load instruction that completes (breaking principle 2) or any microarchitecturally visible instruction with a data dependency

on a completed load instruction (breaking principle 3). Upon breaking either of these principles, the automaton will revert back to the initial state and reset the taint table without flagging that mis-speculation window as vulnerable.

3.2.4 Automaton States

SpecCheck will remain in the initial state until it encounters the first instance of a flushed instruction, where it will transition to one of two intermediary states depending on the type of instruction as well as its microarchitectural state: q_1 will be reached if the flushed instruction is benign, i.e. does not satisfy property (2), and q_2 will be reached if this instruction is a load instruction that completes (CLoad), i.e. satisfies property (2), and the target register will be tainted and added to the taint table. In either state, property (1) is satisfied.

If in state q_1, the state machine will stay until property (2) is satisfied, whereupon the state machine will transition to q_2 and taint the target of this instruction. Any instruction satisfying property (2), i.e. a CLoad, while the state machine is in state q_2 will continue to taint target registers to maintain a complete list of all possible data leaks that are instigated in this mis-speculation window. The q_2 state simultaneously checks for any microarchitecturally visible instructions that satisfy property (3) by using a tainted register as a source. If the mis-speculation window resolves before property (3) can be satisfied, the state machine will return to the initial state and reset the taint table. The register taint propagation is done on the physical register name rather than the architectural register, as only true data dependencies matter when it comes to transient execution attacks.

State	Property		
	1	2	3
q_{ini}	✗	✗	✗
q_1	✓	✗	✗
q_2	✓	✓	✗
q_{acc}	✓	✓	✓

Table 3.1: Properties as defined in subsection 3.1 that are guaranteed to be satisfied at each SpecCheck state

```
O3PipeView:fetch:1000000:0x150e: SAL_R_I : slli rdx, rdx, 0x20 // fetch tick:PC:asm:uop:registers
O3PipeView:decode:1001000                                     // decode end-tick
O3PipeView:rename:1002000                                     // rename end-tick
O3PipeView:dispatch:1003000                                   // dispatch end-tick
O3PipeView:issue:1004000                                      // issue end-tick
O3PipeView:complete:1005000                                   // complete end-tick
O3PipeView:retire:0:store:0                                   // retire end-tick (0->FLUSHED)
```

Figure 3.2: An example of a flushed DynInst instruction on a simulated out of order x86 processor.

3.2.5 The Accept State

When the state machine reaches the accept state, the program can be determined to have an exploitable gadget within its source code. As there are likely many vulnerable code regions in a program, the state machine does not halt and exit the program, but instead returns back to the initial accept state without consuming an additional instruction to ensure a new mis-speculation window can be evaluated immediately succeeding the resolution of the previous one. Since code regions are likely to be identified multiple times, our gem5 implementation will only raise an alert the first time a unique mis-speculated window reaches the final accept state, but will continue tracking the total number of vulnerable mis-speculation windows that are encountered. More implementation details are discussed in section 3.3.

3.3 gem5 Implementation

SpecCheck was designed with software and hardware co-design as the primary goal, and the finite state machine lays a preliminary framework for a robust security debugger to help computer architects in creating secure hardware. To evaluate the effectiveness and feasibility of SpecCheck's finite state machine as a design tool, it was implemented on the cycle-accurate processor simulator gem5 [5]. In this section, we describe the implementation specific details and provide an overview of how this tool can be used as an effective security debugging suite for secure hardware design.

3.3.1 gem5 Security Debugger

The gem5 processor simulator was chosen as the evaluation suite for SpecCheck as it is the most used

simulator for computer architecture researchers and is an open-source development project with an active repository on GitHub [5]. As is, gem5 provides a vast suite of debugging tools tailored towards hardware architects to develop novel hardware solutions without needing to implement their solutions on bare metal hardware, but this debugging suite currently lacks sufficient support for hardware security primitives to defend against complex vulnerabilities such as Meltdown, Spectre, and Row-Hammer [41]. As Meltdown is considered to be successfully mitigated [49] and the nature of a Row-Hammer attack is based on the physical properties of DRAM, we consider them to be out of scope for this work.

SpecCheck was implemented as an additional debugging tool within the gem5 simulator, and can be accessed using the `--debug-flags=SpecCheck` debug flag when running a gem5 simulation. The SpecCheck Proof-of-Concept is implemented on gem5's out of order (O3) CPU, consisting of 7 pipeline stages regardless of the underlying architecture: fetch, decode, rename, dispatch, issue, complete, and retire. This microarchitectural consistency ensures that SpecCheck is compatible with x86, ARM, and RISC V processors. As there is no explicit pipeline stage for *execute*, we denote an executed instruction as one that has successfully been *issued* from the reorder buffer to the appropriate functional unit for execution, and the retire stage is analogous to the commit stage of a traditional processor pipeline where correctness is preserved. As Spectre attacks are completely visible through the system call interface, there is no need to simulate an operating system in Full System (FS) mode, so all evaluations were done purely in system emulation (SE) mode to save on computation and simulation time.

The gem5 implementation makes extensive use of the dynamic instruction (DynInsts) API [5] to access all relevant microarchitectural state information about an instruction through a single interface: the high-level assembly instruction, the corresponding micro-operation, the instruction's program counter, the *renamed* target and source registers, and the processor tick of each pipeline stage's completion (or a 0 if that stage does not finish). An example of a DynInst in gem5 can be seen in Figure 3.2 where an output trace is obtained by running a gem5 simulation with the `O3PipeView` flag enabled, allowing the contents of each DynInst to be written to an external trace file. This DynInst demonstrates a SAL (left shift) assembly instruction occurring between a source register value (**rdx**) and an immediate value (**0x20**), and the result written to the destination register **rdx**, with a corresponding **slli** micro-operation. This trace also provides the

program counter associated with the SAL instruction (**0x150e**), as well as the micro-operation code (**0**), and sequence number, both of which are ignored in the SpecCheck implementation. Lastly, this trace contains the tick of the completion of each pipeline stage, showing that the instruction successfully completes but gets flushed from the pipeline as the retire stage never finishes. As DynInst instructions reference architectural registers, an additional step is needed to access the corresponding renamed microarchitectural registers.

SpecCheck is implemented as a monitor in the ROB during the commit stage of the O3 pipeline where it consumes instructions as they are removed from the buffer. As the DynInst API provides an omnipotent view of the pipeline, a ROB monitor ensures that SpecCheck evaluates instructions in program order. Since the SpecCheck proof of concept only considers userspace programs under syscall emulation, the SpecCheck state machine only consider instructions that execute within `main`'s execution, including all called/linked functions. To invoke the SpecCheck monitor in gem5, the user passes the SpecCheck debug flag, which initializes the FSM and activates the monitor within the commit stage.

The SpecCheck module implementation consists of a single source/header code file combination within the O3 namespace. SpecCheck maintains two key global structures to keep track of current state, the register taint table (vector), and a list of all unique PCs (misspeculation regions) that are marked vulnerable throughout the program's lifetime (vector). When an instruction is ready to be removed from the ROB (squashed or not), it is passed to the state machine through an `advance_FSM()` function invocation. This will change the state of the FSM and update the taint table as necessary. Any time a unique misspeculation window is marked as vulnerable, it is added to the vector of unique, vulnerable PCs. When a unique PC is found to be vulnerable, SpecCheck will raise an alert to stdout, notifying the user of the potentially vulnerable PC.

3.3.2 Defining Micro-Visible Instructions

Microarchitecturally visible (μV) instructions are instructions that create externally observable effects. As defined in Section 3.1, memory, floating point, and control flow operations all influence microarchitectural state, making instructions of this type useful for transmitting information via side channels. As demonstrated by Mosier **et.al.**, the ISA itself needs to define the microarchitectural side-effects of the instruction, leaving

it up to the ISA to identify this type of instructions [56]. To identify μV instructions, the SpecCheck FSM also examines each instruction's StaticInst pointer which maintains static information about each instruction type, e.g. whether it is microarchitecturally visible or not. All StaticInst instructions that are of memory, floating point, or control flow type are marked as μV.

Chapter 4

Evaluation

4.1 Evaluation Methodology

The SpecCheck implementation in gem5 was evaluated using a variety of programs spanning multiple Spectre variants, Spectre V1 and V2 [44], SpecRSB [46], Speculative Store Bypass [32], and a mitigated Spectre v1 using fences to demonstrate a strong security guarantee and 12 programs from the SPEC CPU 2017 [7] benchmark suite to show simulated time overhead. SpecCheck was evaluated using an out of order x86 CPU across all experiments, identifying known side-channel gadgets that have been shown to leak data through transient execution attacks and evaluating the simulation overhead of the SpecCheck security debugging module. We additionally evaluate SpecCheck on a series of microbenchmarks that do not have speculative execution, i.e. simple arithmetic or I/O programs. We further demonstrate that SpecCheck does not find vulnerable gadgets within these programs.

Each proof of concept and microbenchmark is evaluated using gem5's system call emulation mode configured to use the `Deriv03` CPU with `LTAGE` branch prediction [61], 8GB of memory, 32kB 8 way associative L1 instruction/data caches, and a 2MB 16 way associative L2 cache. Each program was statically compiled, and microbenchmarks were compiled without optimization. When SpecCheck is enabled, the FSM will only analyze instructions that execute during the program's lifetime, i.e. throughout the scope of the main function and all invoked functions. Each program was simulated for a maximum of 150,000,000 instructions, just long enough to ensure that each program's execution reaches its known exploitable gadget.

4.2 Identifying Known Gadgets

To showcase the correctness of the SpecCheck debugging module, the finite state machine was run on known Spectre gadgets, such as the one shown in Figure 2.1, to ensure that it is capable of detecting obvious side channels. In addition to counting the number of mis-speculated code regions that are vulnerable to data leaks, the SpecCheck debugger is also capable of tracking the PC value of the beginning of each mis-speculation window that gets flagged as vulnerable. To ensure that SpecCheck identifies the vulnerabilities in the expected code regions of the Spectre PoCs, each flagged PC was compared to the disassembled Spectre binaries to determine if the mis-speculation window induced by each gadget was classified correctly.

We show that SpecCheck is able to correctly classify malicious transient execution in proof of concept code for Spectre variant 1 [2], variant 2 [10], SpecRSB [10], and Speculative Store Bypass (variant 4) [10]: variants that remain largely unmitigated [78]. Each Spectre variant was compiled statically to an x86 binary to ensure proper compatibility with the gem5 simulator's system emulation (SE) mode, leading SpecCheck to analyze all statically linked library code in addition to the exploit code in the PoCs themselves, but only analyzing gadets that occur during the actual execution of the given program. SpecCheck was able to identify the desired victim gadget in each of the Spectre variants, ensuring that the debugging module is capable of providing a an accurate depiction of the vulnerabilities in the transient execution paths. As SpecCheck is evaluated on statically linked programs, it has also discovered numerous, malicious mis-speculated code paths in the C standard library as well as other vulnerable code regions within each PoC distinct from the chosen exploit gadget.

Spectre V1

We first analyze the Spectre v1 exploit to verify that SpecCheck can successfully identified the potential vulnerability induced by the code gadget shown in Figure 2.1 by analyzing a known proof of concept exploit [2]. To start, SpecCheck flags the PC of the first mis-speculated instruction in the vulnerable window, i.e. the mis-predicted branch direction. Then, SpecCheck successfully identifies the first instruction of the predicted branch target, a `lea` instruction at PC `0x401d3f`, as vulnerable to a side-channel attack. The disassembled assembly code for the Spectre v1 gadget is shown in Figure 4.1 with the branch target PC

```
# Branch conditional
0x401d31 <+12>: mov    0xde3c9(%rip),%eax
0x401d37 <+18>: mov    %eax,%eax
0x401d39 <+20>: cmp    %rax,-0x8(%rbp)
0x401d3d <+24>: jae    0x401d72 <victim+77>

# Branch target
0x401d3f <+26>: lea    0xde3da(%rip),%rdx
0x401d46 <+33>: mov    -0x8(%rbp),%rax
0x401d4a <+37>: add    %rdx,%rax
0x401d4d <+40>: movzbl (%rax),%eax
0x401d50 <+43>: movzbl %al,%eax
0x401d53 <+46>: shl    $0x9,%eax
0x401d56 <+49>: cltq
0x401d58 <+51>: lea    0xe1b01(%rip),%rdx
0x401d5f <+58>: movzbl (%rax,%rdx,1),%edx
0x401d63 <+62>: movzbl 0xe05f6(%rip),%eax
0x401d6a <+69>: and    %edx,%eax
0x401d6c <+71>: mov    %al,0xe05ee(%rip)

# End of branch / not taken
0x401d72 <+77>: leave
```

Figure 4.1: Disassembled victim code for a Spectre variant 1 attack gadget shown in Figure 2.1

```
# Misspeculation
0x0000000000402225 <_Z11move_animalP6Animal>:
  0x402225 endbr64
  0x402229 mov    (%rdi),%rax
  0x40222c jmpq   *(%rax)
```

Figure 4.2: Disassembled gadget code for the Spectre BTB (variant 2) POC [10].

directly matching the mis-speculated load instruction flagged by SpecCheck.

Spectre V2

SpecCheck is also able to identify information leaks that occur during branch target injection (BTB) attacks, i.e. Spectre v2. Our analysis uses the Spectre v2 POC code provided by the transientfail repository [10] where we use the same address space, in place variant. Spectre v2 relies on poisoning the target of indirect jumps to induce speculation [44] which SpecCheck is able to identify. Figure 4.2 shows the

exploitable gadget in the Spectre v2 POC that contains the indirect jump instruction at PC `0x40222c` that SpecCheck annotates as vulnerable.

SpecRSB

```
1 int __attribute__ ((noinline)) call_leak() {
2     // Manipulate stack to not return here
3     call_manipulate_stack();
4     // Architecturally shouldnt return here
5     // Encode data in covert channel
6     cache_encode(SECRET[idx]);
7     return 2;
8 }
```

Figure 4.3: SpecRSB (same address space, out of place) gadget to leak information via RSB misspeculation courtesy of the transientfail [10] repository.

Unlike Spectre variants 1 and 2, SpecRSB induces speculation through the return stack buffer rather than branch misprediction [55]. We demonstrate that SpecCheck is able to identify gadgets in any speculated code path, regardless of the reason for misspeculation. As in Spectre v2 and v4, SpecCheck was analyzed on SpecRSB POC code provided by the transientfail repository [10].

We are similarly able to detect malicious transient execution in the SpecRSB exploit as SpecCheck successfully identifies the mis-speculated return target as vulnerable to a side-channel vulnerability. In a SpecRSB gadget shown in Figure 4.3, SpecCheck correctly annotates the resumption of execution within `call_leak()` after the call to `call_manipulate_stack()` as vulnerable. The `call_manipulate_stack()` function induces speculation on the function's return address which speculatively returns control to `call_leak()`. SpecCheck correctly identifies this misspeculation window as susceptible to leakage with the use of the `cache_encode()` mechanism.

Speculative Store Bypass (V4)

SpecCheck is lastly able to identify Speculative Store Bypass, or Spectre v4, attacks that exploit store to load forwarding to induce speculation [11]. Figure 4.4 shows a simplified gadget to leak secret information via store to load bypasses provided by transientfail [10]. SpecCheck is able to correctly identify the misspeculated code path on line 15 as vulnerable to potential data leaks as this instruction induces the

speculation and loads the secret. Unlike the previous Spectre variants, Speculative Store Bypass attacks do not rely on speculative control flow, thus demonstrating SpecCheck's ability to identify leakage regardless of the reason for speculation.

```
1  char access_array(int x) {
2    // store secret in data
3    strcpy(data, SECRET);
4
5    // flushing the data increases
6    // probability of speculation
7    mfence();
8    flush(data);
9
10   // ensure data is flushed at this point
11   mfence();
12
13   // overwrite data
14   (*(*data_))[x] = OVERWRITE;
15
16   // Encode stale value in the cache
17   cache_encode(data[x]);
18 }
19
```

Figure 4.4: Simplified Speculative Store Bypass gadget [10]

Mitigated Spectre v1

SpecCheck was also evaluated on a mitigated Spectre v1 proof of concept using a load fence as shown in figure 2.2. SpecCheck no longer identifies the victim function as vulnerable to speculative leaks as the `lfence` ensures that the loads prior to the fence are forced to complete, thus mitigating any possibility of side channel leakage. This further demonstrates SpecCheck's ability to evaluate proposed Spectre defenses in a systematic way, as a successful mitigation, whether in architecture or software, will no longer be marked as vulnerable by SpecCheck.

4.3 Microbenchmark Evaluation

SpecCheck was also evaluated on a small series of microbenchmarks that do not induce speculation. These microbenchmarks include a program to store and load simple variables to/from memory, perform arithmetic operations such as addition, and utilize the `printf()` function to print a string. As each microbenchmark is compiled statically, SpecCheck only analyzes code that runs during the execution of the program, i.e. everything invoked throughout `main`'s lifetime. Additionally, each microbenchmark was compiled without any optimizations to ensure that none of the benchmark code is optimized out. SpecCheck identified no potential leaks in either the store/load or arithmetic microbenchmarks, and finds a single potential leak in the `IO_puts` function call related to the `printf()` benchmark. These microbenchmarks demonstrate a low false positive rate and SpecCheck's use case of solely analyzing speculatively executed instructions.

```
1 int main () {
2     int a = 1;
3     int b = 2;
4     int c = a + b;
5     return 0;
6 }
```

Figure 4.5: Example of an arithmetic microbenchmark

4.4 False Positives

Variant	True Positives	False Positives	True Negatives	False Negatives	FP rate
Variant 1	1	59	157	0	0.273
Variant 2	1	43	119	0	0.265
RSB	1	35	108	0	0.245
SSB	1	45	128	0	0.260

Table 4.1: SpecCheck confusion matrix results for all misspeculation windows across each Spectre variant.

To provide a strong security guarantee, SpecCheck aims to avoid all instances of false negatives where a malicious gadgets is not identified as potentially malicious and instead prefers a slight false positive

rate. To identify false positive and false negative rates, a table of confusion matrix results for each Spectre variant evaluated with SpecCheck is given in table 4.1. Each entry in the confusion matrix table represents the number of unique misspeculated PCs: true positives are correctly identified as vulnerable and reach SpecCheck's accept state, i.e. known gadgets, false positives are incorrectly identified as vulnerable, true negatives are correctly identified as benign, i.e. never reach the accept state, and false negatives are incorrectly identified as benign.

The 0% false negative rate is indicative of SpecCheck's strong security guarantees and is evident by SpecCheck's ability to identify known gadgets across multiple Spectre variants. Ensuring that SpecCheck is comprehensive enough to detect all known variants incurs a relatively high false positive rate between 24-27%, but each of these annotated regions could, *in theory*, be used to transiently transmit secret data. However, for the sake of evaluation, each PC that is not known to be vulnerable a priori is deemed as a false positive. We leave further analysis of each annotated region and techniques for minimizing the false positive rate for future work.

4.5 Simulation Overhead

The finite state machine design for the SpecCheck debugger was chosen for its lightweight implementation to incur minimal overhead in simulation time when using gem5. As SpecCheck is currently a purely software defined tool without any modifications to the simulated hardware, the module's overhead was determined by an increase in *simulation* time on the host when using the SpecCheck debugger versus a normal simulation workload. These statistics were measured in overall seconds taken to run the simulation on the host as provided by the gem5 statistics API,

Unlike the vulnerability evaluation, each benchmark is simulated for 500 million instructions during the warmup period, followed by a subsequent 500 million instructions. Across all 12 benchmarks, SpecCheck incurs only a 4.05% timing overhead on average while incurring as low as 1.8% overhead in the best case (mcf), and 10 of the 12 benchmarks sustain overheads of less than 5%. Even in the worst case performance, SpecCheck experiences a maximum overhead of 9.8% in simulation time which we deem as reasonable for simulation purposes.

Chapter 5

Discussion

5.1 Related Work

Speculative Taint Tracking (STT) [82] presents a hardware protection scheme for mitigating Spectre attacks by tracking all speculatively accessed data. Like SpecCheck, STT separates Spectre attacks into two phases, the access and transmit phases, and allows speculative instructions to execute until they are able to form a covert channel, at which point dependent instructions must block until the access is resolved (i.e. non speculative). These instructions are tainted based on the initial access instruction similarly to SpecCheck's taint mechanism. However, STT presents a hardware based defense mechanism for mitigating Spectre attacks, while SpecCheck is designed to systematically evaluate the security of a proposed architecture. As SpecCheck is not a proposed defense, has an omnipotent view of the pipeline, and does not consider architectural performance constraints (unlike STT), it is able to easily verify the overall effectiveness of security focused designs such as STT.

Spectify [58] presents another transient execution defense approach based on a FSM characterization of a Spectre attack. However, the Spectify FSM is solely limited to cache based side channel attacks, whereas SpecCheck aims to be agnostic to the underlying transmission gadget used during speculation, as various other, non-cache side channels exist [25, 74, 73, 44]. Additionally, Spectify only considers speculation as a result of branch misprediction, leaving variants that make use of other types of misspeculation, such as Spectre v4 [11], as vulnerable even with the Spectify detection mechanism. We show that this type of FSM design is not enough to reason about Spectre v4, a variant that is detectable with the SpecCheck FSM.

Axiomatic Hardware-Software Contracts for Security [56] introduces Leakage Containment Models

(LCMs) that describe how data can leak microarchitecturally given a hardware implementation. This work presents a formal approach to hardware/software contracts that expands upon memory consistency models to define microarchitectural leakage models by similarly analyzing both architectural and microarchitectural states. The final contribution, CLOU, is a static analysis tool integrated with LLVM to create LCMs and identify microarchitectural leakage for a given program. This is a key contrast to SpecCheck which acts as a runtime verification tool that is tightly integrated within the gem5 simulator. Additionally, the analysis of CLOU considers solely Spectre v1 [44], v1.1 [43], and v4 [11], while neglecting to consider other key variants, such as Spectre v2 [44] and SpecRSB [55], that are identified by SpecCheck.

CheckMate [70] presents a similar tool for microarchitecture analysis to determine vulnerable regions of the hardware design that may be exploited by transient execution vulnerabilities. By providing a hardware specification, CheckMate introduces the "microarchitecturally happens before" (μhb) primitive to create an ordered event graph at the microarchitectural level to detect subtle event sequences that are exploited by transient execution attacks. The user then supplies a μhb subgraph that defines an exploit behavoir, and CheckMate will analyze the ordered event graph for vulnerabilities matching the provided exploit behavior. Our approach differs from the work in CheckMate as SpecCheck (1) analyzes software patterns by means of Spectre gadgets in combination with pipeline state information and (2) examines the pipeline in isolation from other microarchitectural structures such as the L1 cache. For this reason, SpecCheck is primarily focused towards identifying vulnerabilities that arise during the software/hardware codesign process where CheckMate is exclusively used to verify a microarchitectural design implementation. Additionally, SpecCheck is built directly into the gem5 simulator and is designed be used in conjunction with various CPU configurations and binary programs specified by the end user.

Fadiheh et al. [20] present work on a formalized approach characterizing transient execution vulnerabilities by extending previous work *Unique Program Execution Checking* (UPEC) [21] to cover out of order execution. However, similarly to CheckMate, this work is exclusively concerned with microarchitectural structures, differing from SpecCheck's analysis of software with pipeline state. The experimental methodology was only carried out on a single *Berkeley Out of Order Machine* (BOOM) [13] and was able to detect multiple Spectre variants along with Meltdown. SpecCheck aims to build upon this formalization

by using a state machine to better fit the debugging framework offered by the gem5 simulator, and eschews microarchitectural structures such as the ROB in favor of the pipeline for the same reason.

Various other proposed mechanisms for defending against Spectre have been proposed in recent work [81, 76, 16, 66]. However, like many of the examples above, these works present active defense mechanisms for transient execution attacks, and are the perfect candidates for security evaluation using SpecCheck.

5.2 Future Work

To design a more robust debugging tool for developing secure architectures, we plan to extend the SpecCheck platform to seamlessly integrate into full system simulation to account for software level defenses to transient execution vulnerabilities. As Spectre attacks exploit speculative execution via avenues such as branch prediction, the SpecCheck tool only analyzes mis-speculated code regions to determine the presence of potential data leaks that may have side effects later on in the pipeline. However, this current approach is primarily focused on architecture design and does not consider software based defenses when analyzing transient execution paths. Expanding the SpecCheck debugging module to include operating system level logic will help both computer architects and software engineers to develop secure hardware and software in combination, targeting different areas of the systems stack to implement mitigations.

SpecCheck currently incurs a high false positive rate to ensure that no false negatives arise during the evaluation process. In the future, we plan to analyze each of these false positive regions of code in depth to determine the presence of other, potentially novel, side channel attacks. We believe that a significant number of these false positive regions may actually be vulnerable to Spectre attacks and therefore marked as true positives, but further analysis is required to support this claim. In that case, the number of false positives would decrease while continuing to guarantee no false negatives occur.

We also plan to extend our SpecCheck framework to successfully debug a more diverse set of microarchitectures as the proof of concept tool is currently only implemented for an out of order, 64 bit processor. As gem5 separates the ISA from the microarchitecture as much as possible, SpecCheck remains ISA agnostic as it is solely tied to the gem5 out of order (O3) microarchitecture. The SpecCheck tool will

be extended to reason about other types of microarchitectures, such as in order processors and processors with different pipeline designs, in addition to the current O3 model. To do so, SpecCheck will be extended by generalizing the O3 pipeline stages into generic ”families” of stages used by other microarchitectures, e.g. any *execute* stage will undergo the same logic, despite the naming convention used by that exact implementation. As any processor capable of speculative execution, including in order processors, are vulnerable to Spectre attacks, we do not want to limit SpecCheck to reasoning solely about out of order pipelines and hope to generalize the tool to perform security analysis on any modern processor pipeline design. Evaluating different branch predictors included in gem5 (BiModeBP, LocalBP, and TournamentBP) as well as varying data cache implementations will provide a wider range of debugging opportunities for secure hardware design using SpecCheck.

To reach the full capability of our hardware security debugger, we plan to implement fuzzing functionality to fully explore the entire program space for any code regions that may be vulnerable and reachable by an attacker. By adding fuzzing capabilities to the final SpecCheck platform, the debugger will provide a completely thorough evaluation of transient execution vulnerabilities for any program ran on any modern architecture.

Chapter 6

Conclusion

SpecCheck introduces a novel framework for characterizing code regions as being vulnerable transient execution attacks through a finite state machine approach. The PoC implementation in the gem5 simulator on an out of order processor proves to be a lightweight method of identifying data leakage through transient execution and provides significant groundwork towards a comprehensive security debugger for developing secure microarchitectures.

By modeling speculative execution vulnerabilities with a formal state machine, SpecCheck is able to generalize the covert channel primitive inherent in all Spectre attacks to not only identify known vulnerabilities but also detect potentially novel side channels that may transiently leak information. SpecCheck provides a high security guarantee, successfully identifying vulnerable code regions in all four main Spectre variants. SpecCheck leverages the pipeline state information to pinpoint exact regions in a program's execution flow that may be vulnerable and further provides support for analyzing speculative execution implementation by tracking all areas of mis-speculation in a programs simulation. This feature allows users to identify code regions with disproportionately high rates of mis-speculation in addition to providing a strong foundation for security research. SpecCheck's implementation provides lightweight debugging support to the gem5 simulator for developing secure architectures, incurring a 1.8% simulation overhead in the best case and a 4.05% overhead on average while ensuring a 0% false negative rate in each Spectre variant.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Erik August. Spectre example code. https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6, 2018.

[3] Erik August. Spectre v1 example code. \https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6/, 2018.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual. INRIA, version, 6(11), 1999.

[5] 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. The gem5 simulator. SIGARCH Comput. Archit. News, 39(2):1–7, aug 2011.

[6] Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. Information and Computation, 182(2):137–162, 2003.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pages 41–42, 2018.

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In 28th USENIX Security Symposium (USENIX Security 19), pages 249–266, Santa Clara, CA, August 2019. USENIX Association.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In USENIX Security Symposium, 2019. extended classification tree at https://transient.fail/.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In 28th USENIX Security Symposium (USENIX Security 19), pages 249–266, 2019.

[12] Anton Cao. Spectre variant 2 poc. https://github.com/Anton-Cao/spectrev2-poc/, 2020.

[13] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley, Sep 2017.

[14] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), pages 288–28815. IEEE, 2019.

[15] François Chollet et al. Keras. https://keras.io, 2015.

[16] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy. In International Symposium on Microarchitecture (MICRO), 2021.

[17] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 351–366, 2007.

[18] Jonas Depoix and Philipp Altmeyer. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. Advanced Microkernel Operating Systems, page 75, 2018.

[19] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS), 2014.

[20] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6, 2020.

[21] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 994–999, 2019.

[22] E Mark Gold. Complexity of automaton identification from given data. Information and Control, 37(3):302–320, 1978.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

[24] Google. Google safeside. https://github.com/google/safeside/, 2021.

[25] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In 27th USENIX Security Symposium (USENIX Security 18), pages 955–972, Baltimore, MD, August 2018. USENIX Association.

[26] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, Engineering Secure Software and Systems, pages 161–176, Cham, 2017. Springer International Publishing.

[27] Roberto Guanciale, Musard Balliu, and Mads Dam. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In Conference on Computer and Communications Security (CCS), 2020.

[28] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1–19. IEEE, 2020.

[29] Berk Gülmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning. CoRR, abs/1907.03651, 2019.

[30] Boris Hanin. Which neural net architectures give rise to exploding and vanishing gradients? Advances in neural information processing systems, 31, 2018.

[31] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[32] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[33] Intel. Speculative store bypass cve20183639 intelsa00115, 2018.

[34] Intel. Intel pcm, Updated 2022 [Online].

[35] Yier Jin. Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits. In 2014 IEEE Computer Society Annual Symposium on VLSI, pages 19–24. IEEE, 2014.

[36] Yier Jin, Bo Yang, and Yiorgos Makris. Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing. In 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pages 99–106. IEEE, 2013.

[37] Mike Johnson. Superscalar multiprocessor design. Prentice-Hall, Inc., 1991.

[38] Johannes Jurgovsky, Michael Granitzer, Konstantin Ziegler, Sylvie Calabretto, Pierre-Edouard Portier, Liyun He-Guelton, and Olivier Caelen. Sequence classification for credit-card fraud detection. Expert Systems with Applications, 100:234–245, 2018.

[39] Michael Kaminski and Nissim Francez. Finite-memory automata. Theoretical Computer Science, 134(2):329–363, 1994.

[40] R.E. Kessler. The alpha 21264 microprocessor. IEEE Micro, 19(2):24–36, 1999.

[41] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. ACM SIGARCH Computer Architecture News, 42(3):361–372, 2014.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[43] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

[44] 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. Spectre attacks: Exploiting speculative execution. In 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[45] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. Communications of the ACM, 63(7):93–101, 2020.

[46] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT 18), 2018.

[47] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. Poisonivy: Safe speculation for secure memory. In International Symposium on Microarchitecture (MICRO). IEEE, 2016.

[48] 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. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), 2018.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. Communications of the ACM, 63(6):46–56, 2020.

[50] J. Longo, E. De Mulder, D. Page, and M. Tunstall. Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. Cryptology ePrint Archive, Report 2015/561, 2015. https://ia.cr/2015/561.

[51] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. {DOLMA}: Securing speculation with the principle of transient {Non-Observability}. In 30th USENIX Security Symposium (USENIX Security 21), pages 1397–1414, 2021.

[52] Eric Love, Yier Jin, and Yiorgos Makris. Enhancing security via provably trustworthy hardware intellectual property. In 2011 IEEE international symposium on hardware-oriented security and trust, pages 12–17. IEEE, 2011.

[53] Eric Love, Yier Jin, and Yiorgos Makris. Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. IEEE Transactions on Information Forensics and Security, 7(1):25–40, 2011.

[54] Jason Lowe-Power. Visualizing spectre with gem5. http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html/, 2018.

[55] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 2109–2122, 2018.

[56] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. In Proceedings of the 49th Annual International Symposium on Computer Architecture, pages 72–86, 2022.

[57] George C Necula. Proof-carrying code. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 106–119, 1997.

[58] Arash Pashrashid, Ali Hajiabadi, and Trevor E Carlson. Fast, robust and accurate detection of cache-based spectre attack phases. In Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, pages 1–9, 2022.

[59] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. International Symposium on Computer Architecture (ISCA), 2021.

[60] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In European Symposium on Research in Computer Security. Springer, 2019.

[61] André Seznec. A new case for the tage branch predictor. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, page 117–127, New York, NY, USA, 2011. Association for Computing Machinery.

[62] Ralf C. Staudemeyer and Eric Rothstein Morris. Understanding lstm – a tutorial into long short-term memory recurrent neural networks, 2019.

[63] Sam S Stone, Kevin M Woley, and Matthew I Frank. Address-indexed memory disambiguation and store-to-load forwarding. In 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05), pages 12–pp. IEEE, 2005.

[64] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. ACM Sigplan Notices, 2004.

[65] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 395–410, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, pages 109–120, 2009.

[67] Saeid Tizpaz-Niari, Pavol Černỳ, Sriram Sankaranarayanan, and Ashutosh Trivedi. Efficient detection and quantification of timing leaks with neural networks. In International Conference on Runtime Verification, pages 329–348. Springer, 2019.

[68] Saeid Tizpaz-Niari, Pavol Cerny, and Ashutosh Trivedi. Data-driven debugging for functional side channels. The Network and Distributed System Security (NDSS) Symposium, 2022.

[69] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of research and Development, 11(1):25–33, 1967.

[70] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 947–960. IEEE, 2018.

[71] Paul Turner. Retpoline: a software construct for preventing branch-target-injection, 2018.

[72] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In Proceedings of the 27th USENIX Security Symposium. USENIX Association, August 2018. See also technical report Foreshadow-NG [77].

[73] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In Security and Privacy (S&P), May 2021.

[74] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In 31st USENIX Security Symposium (USENIX Security 22), pages 679–697, 2022.

[75] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In Proceedings of the 34th Annual Computer Security Applications Conference, pages 393–406, 2018.

[76] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 572–586, 2019.

[77] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018. See also USENIX Security paper Foreshadow [72].

[78] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In 31st USENIX Security Symposium (USENIX Security 22), pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[79] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. ACM Computing Surveys (CSUR), 54(3):1–36, 2021.

[80] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 99–110, 2010.

[81] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In International Symposium on Computer Architecture (ISCA), 2020.

[82] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 954–968, 2019.

[83] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. Neural computation, 31(7):1235–1270, 2019.

[84] Xuehui Zhang and Mohammad Tehranipoor. Case study: Detecting hardware trojans in third-party digital ip cores. In 2011 IEEE International Symposium on Hardware-Oriented Security and Trust, pages 67–70. IEEE, 2011.