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

Zack McKevitt
University of Colorado Boulder
Boulder, USA
zachary.mckevitt@colorado.edu

Ashutosh Trivedi
University of Colorado Boulder
Boulder, USA
ashutosh.trivedi@colorado.edu

Tamara Silbergleit Lehman
University of Colorado Boulder
Boulder, USA
tamara.lehman@colorado.edu

***Abstract*—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). 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. The FSM is easily incorporated into commonly used processor simulators. 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 two of the main Spectre variants, variant 1 (PHT) and 2 (BTB), with a 100% true positives and an average of 14% false positive rate for malicious sequences of code and an average of 19% vulnerable windows identified for the SPEC benchmark suite.**

***Index Terms*—Transient execution attacks, microarchitectural security, processor simulation, security metrics**

I. Introduction

Transient execution attacks are a new family of complex exploits that leave most modern processor vulnerable to exploitation [8], [22], [24], [50]. This type 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 attack, one of the first examples of this type of attacks, leverages the branch predictor to mislead the processor to access unauthorized memory to use the value of a secret as an index into an array which is loaded into the cache [22]. 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 attacks.

This type 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 leverage these details. 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 not equipped 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 [9], [18], [31], [39], [40], [51]. 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 [3], [19], [20], [27], [28], [54].

Recently proposed work that combine software and hardware state to evaluate security are difficult to use for non-security experts. For example, CheckMate [41], 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 [17] 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

979-8-3503-4254-3/23/$31.00 ©2023 IEEE
DOI 10.1109/PACT58117.2023.00030

for non-security experts to adopt. The SpecCheck approach looks to complement these tools for security experts and to supplement non-security experts the ability to evaluate security without additional effort.

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, and 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 [10], [26], [47], [52], [53] 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.

The initial experiments suggest that SpecCheck is able to correctly identify multiple known gadgets across two main unmitigated Spectre variants: Spectre variant 1 [22] and Spectre Variant 2 [22]. 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 the tool to identify potentially vulnerable implementations.

One of the key points of 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 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 a systematic hardware security verification tool in the gem5 simulator. [1]

II. 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.

A. 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 [7], [22], [24], [33], [34], [43], [44], [48]. 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 of accessing different data values, but other types of disclosing gadgets exist [25], [46]. 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 [7].

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]SpecCheck is available here https://zenodo.org/record/8326829

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

Fig. 1: Example Victim Code for a Spectre Variant 1 Attack.

The first transient execution attack published, Spectre variant 1, leverages the branch predictor to direct branches to a desired branch direction [22]. A Spectre variant 1 victim gadget [22] is shown in Figure 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 taken for the branch on line 1. 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.

The second variant of this original attack (Spectre v2), poisons the branch target buffer (BTB) used to speculate around direct and indirect jumps, instead of the PHT as the first variant does [22]. 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 is the same as the first variant.

While the method of exploiting prediction to speculatively execute loads is common to all transient execution attacks (including the Meltdown family of attacks) [7], [8], [22], [24], [29], the prediction mechanisms vary across variants. Several other variants have been discovered that are able to use other microarchitectural components besides the branch predictor, such as the Return Stack Buffer and the Memory Disambiguator, to transiently execute a load instruction that accesses the secret data [22], [29], [33], [34], [43], [44], [48]. 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 [22], [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.

B. Transient Execution Attack Mitigation

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 mitigation for Meltdown [16], the Spectre family still affects a vast majority of processors in use today. One popular software-based mitigation technique, shown in Figure 2, is to use a serializing *load fence* [37]. The load fence instruction (line 4 in Figure 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 [42] 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 [29], and more recently RetPoline has been shown to be ineffective against a novel Spectre variant RetBleed [49].

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 [47] and DOLMA [26] 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 [53] 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 have been considered vulnerable before.

C. The gem5 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 [4] 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

```
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 }
```

Fig. 2: Example victim gadget for a Spectre variant 1 with a load fence as a mitigation (line 4).

by the dynamic instruction object widely used throughout the simulation.

The dynamic instruction object has most of the information required by the SpecCheck FSM, including the state of the instruction through the pipeline (represented as the cycle in which each stage is reached and a zero otherwise), 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 source and renamed destination registers 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.

gem5 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 III-E.

The gem5 simulator provides another instruction API, static instructions, that provides the static information from the ISA description 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 and the various micro-operations associated with a single macro operation. SpecCheck uses this API to embed into each instruction if they have any microarchitecturally visible effects which is important for determining the possibility for a transient execution attack. As argued in prior work by Mosier *et.al.* [30], we expect future ISA definitions to include the information on whether or not an instruction has microarchitecturally visible effects.

III. 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 and 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 potentially 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 recover the secret.

The key insight of the SpecCheck framework is to consider the state of both the software and the hardware in combination at runtime to systematically determine the potential of success for a transient execution attack. SpecCheck accomplishes this goal by tracking the state of a register finite state machine described in Figure 3. Upon encountering a mis-speculated code region, the first condition of the FSM is a load instruction that completes but does not retire. The reason for this condition is that all variants of transient execution attacks rely on the unauthorized access to a sensitive piece of data that is eventually squashed.

Once this initial condition of the completed but not retired load instruction exists, SpecCheck tracks 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 register 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.

A. The Anatomy of a Transient Execution Attack

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

1) Occurs within flushed transient execution window
2) Contains at least one completed load instruction.
3) Executes a microarchitecturally visible instruction that uses a tainted register (directly or indirectly) by 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.

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 does 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.* [30]. The current implementation of SpecCheck identifies commonly known instructions to be microarchitecturally visible, such as branches (control flow state), memory operations (memory system state) and floating point operations (functional unit state). The information of microarchitectural visibility can be included in the instruction definition, similar to how memory instructions are identified in their own category.

The vulnerable state of a register is propagated through instructions regardless of whether they are considered to be microarchitecturally visible, similar to prior work in Information Flow Tracking (IFT) [11], [23], [36]. In the example of the Spectre variant 1 (shown in Figure 1), the maliciously loaded value is tracked through the multiplication operation (i.e. a left shift) by tainting the destination register of the shift instruction. Once this register is used by the transmitting gadget, where it is used as the source of a microarchitecturally visible move instruction, SpecCheck successfully marks the window as vulnerable.

B. Finite State Machines

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 single fixed sized storage structure (described in section III-C) 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 section III-A, 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 instruction parsing. As the number of states and transition functions are kept constant, SpecCheck incurs minimal overhead.

C. The SpecCheck Finite State Machine

We propose a 4-state automaton (shown in Figure 3) 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 reverts back to the initial state once the speculation window is resolved.

The SpecCheck FSM augments a simple finite state automaton by including a taint table that tracks all physical registers that are either the destination of a completed but not retired load instruction or the destination of an instruction that uses a previously tainted physical register. When the accept state is reached or the speculation window is resolved, the taint table is reset.

The Instruction Tuple. SpecCheck is designed as a monitor to the Reorder Buffer (ROB) simulation and consumes instructions in order as they are removed from the ROB. For any given instruction, SpecCheck evaluates: (1) the instruction itself (both macro and micro operations), (2) the *renamed* (i.e., physical) source and destination registers, and (3) the state of the instruction in the execute, 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 the pipeline state, allows SpecCheck to identify side-channel gadgets by tracing data dependencies through the pipeline.

The Taint Table. SpecCheck is further augmented with a taint table to track data dependencies between instructions as information flows through the pipeline. The size of this register array depends on the number of physical registers defined by the underlying microarchitecture. Registers are tainted any time they are the target of a flushed load instruction that 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. If a tainted register is overwritten and none of the input operands are tainted (e.g., loading an immediate value), we untaint the overwritten register. The taint table automatically untaints all registers any time the FSM returns to the initial state, *i.e.* when the transient state is resolved.

The Transition Function. As SpecCheck consumes an instruction, the finite state machine may have three different effects: 1) do nothing, 2) change the current state to a new one and/or, 3) 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 or remove registers to/from the taint table, or clean the table when speculation is resolved. As shown in Figure 3, 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 load that has completed and therefore notifies dependent instructions, but may or may not commit its result to the pipeline. As a CLoad can propagate the result of its load to dependent instructions without committing, side-channels may be used to transmit this information. Additionally we introduce the

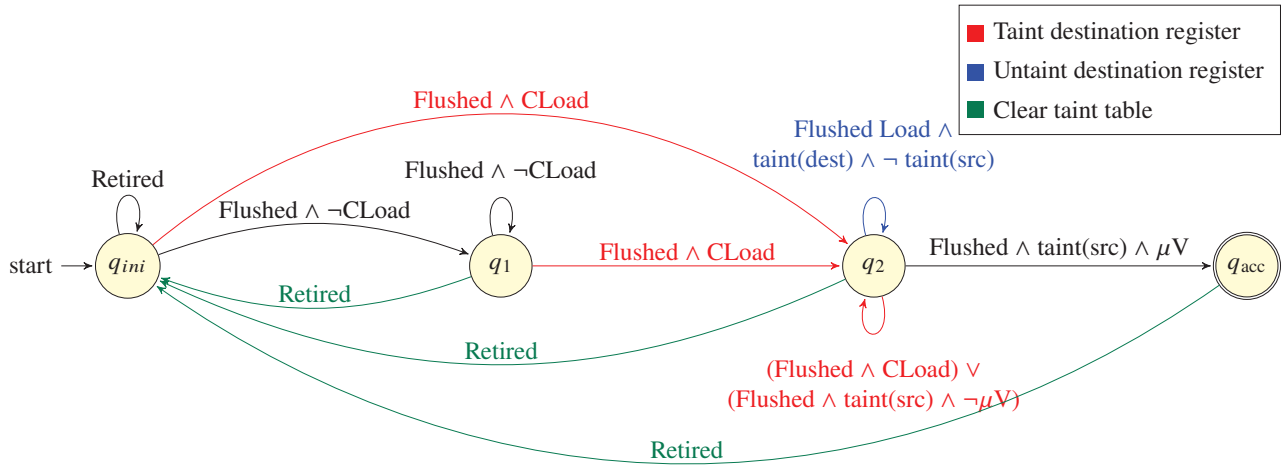

Fig. 3: The Finite State Machine representation for SpecCheck.

`taint(reg)` primitive to represent a lookup in the taint table, where `taint(reg)` indicates that the physical register `reg` is tainted and ¬`taint(reg)` indicates that physical register is not tainted (i.e., not present in the taint table).

In Figure 3, modifications to the taint table are denoted by different colors: red adds the destination register of the current instruction to the taint table, blue untaints the destination register by removing it from the taint table, and green resets the taint table. When an instruction reaches the accept state, once the mis-speculated window is resolved, there is an implied transition back to the initial state, clearing the taint table, without needing to consume an additional instruction.

D. Entering the SpecCheck States

SpecCheck identifies any mis-speculated code region as vulnerable if it adheres to each of the conditions listed in Section III-A. SpecCheck will return to the initial state upon resolution of the mis-speculation window. Specifically, SpecCheck will reject any mis-speculated code region that does not contain a load instruction that completes (breaking condition 2) or any microarchitecturally visible instruction with a data dependency on a completed load instruction (breaking condition 3). Upon breaking either of these conditions, the automaton will revert back to the initial state and reset the taint table without flagging that mis-speculation window as vulnerable.

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 condition (2)) and q_2 will be reached if this instruction is a load instruction that completes—CLoad—(*i.e.* satisfies condition (2)). A transition into the q_2 state triggers the addition of the destination register of the instruction into the taint table (*i.e.* tainting the register). In either state, condition (1) is satisfied.

If in state q_1, the state machine will maintain state until condition (2) is satisfied, whereupon the state machine will transition to q_2 and taint the destination register of the instruction by adding it to the taint table. Any instruction satisfying condition (2), i.e. a CLoad, while the state machine is in state q_2 will continue to taint the destination register 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 condition (3) by using a tainted register as a source. If the mis-speculation window resolves before condition (3) can be satisfied, the state machine will return to the initial state and reset the taint table. If at any point in the mis-speculation window a tainted register is overwritten and neither of the input operands are tainted, we untaint the overwritten register. The register taint propagation is done on the physical register rather than the architectural register, as only true data dependencies matter when it comes to transient execution attacks. Even if an architectural register is overwritten before a side-channel can be created, an instruction may still rely on an old physical register that is still able to leak information.

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

TABLE I: Conditions as defined in Section III-D that are guaranteed to be satisfied at each SpecCheck state

The Accept State. When the state machine reaches the accept state, q_{acc}, the program can be determined to have an exploitable

gadget within its source code. During the mis-speculation window, the program counter (PC) of the first mis-speculated instruction is stored, and will log this PC to an external file when reached. SpecCheck remains in this state until the mis-speculation window closes (i.e. a retired instruction is seen), where it reverts back to the initial state. Since code regions are likely to be identified multiple times, the gem5 implementation will only log a PC the first time this mis-speculated window reaches the final accept state, but will continue tracking the total number of vulnerable mis-speculation windows that are encountered.

E. Implementation

SpecCheck's primary goal is to serve as a software and hardware systematic verification tool, and the finite state machine lays a preliminary framework for a robust security debugger to help computer architects in evaluating security at the microarchitectural level. To evaluate the effectiveness and feasibility of SpecCheck's finite state machine as a design tool, we implemented SpecCheck in the cycle-accurate processor simulator called gem5 [4]. In this section, we describe the gem5 implementation specific details and provide an overview of how this tool can be used as an effective security debugging suite for secure hardware design.

1) gem5 Security Debugger: The gem5 processor simulator was chosen as the evaluation suite for SpecCheck as it is a widely used simulator for computer architecture researcher and it is an open-source development project with an active repository on GitHub [4]. 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. However, the gem5 simulator currently lacks sufficient support to evaluate microarchitectural designs against complex vulnerabilities that leverage microarchitectural states, such as Meltdown, Spectre, and Rowhammer [21]. As the nature of a Rowhammer attack is based on the physical properties of DRAM, we consider them to be out of scope for this work.

SpecCheck is 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: fetch, decode, rename, dispatch, issue, complete, and retire. As there is no explicit pipeline stage for *execute*, we denote an executed instruction as one that has successfully *completed*. SpecCheck is designed to be compatible with most architectures implemented in gem5, including x86, ARM, and RISC-V ISAs, as it is incorporated as a monitor in the underlying microarchitecture.

The gem5 SpecCheck implementation makes extensive use of the dynamic instruction (DynInsts) API [4] 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 architectural and physical destination and source registers, and the processor cycle of each pipeline stage (or a 0 if that stage is not reached). An example of a DynInst of a CLoad in gem5 can be seen in Figure 4 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—note that SpecCheck does not need to have this option enabled. The DynInst shown is a MOV instruction, specifying a source from the data segment indexed by the value stored in register *rcx*. The result of the MOV instruction is written to the destination register *rax*. The DynInst includes the corresponding micro-operation assigned to the macro-operation, in this case **ld**. Note that SpecCheck uses the corresponding physical registers (not shown on the figure) as opposed to the architectural registers for filling the taint table and tracking dependencies. 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 cycle value is 0. As this specific MOV instruction loads from the data segment, completes, and does not retire, it is classified as a CLoad in our implementation.

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 ROB. gem5 forces all instructions to go through the commit stage regardless if they are flushed from the pipeline or not, ensuring that SpecCheck captures all instructions in the instruction stream (flushed at any stage or retired). In addition, given that the DynInst API tracks the instruction through the whole pipeline and that the commit stage processes instructions in order, having the SpecCheck at the commit stage with the DynInst API ensures that all instructions are inspected in program order. 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.

SpecCheck maintains four key global structures: the current state, the register taint table (vector), a list of all unique mis-speculated code regions by fist mis-speculated PC (vector), and a list of all unique mis-speculation windows (PCs) that are marked vulnerable throughout the program's lifetime (vector). An instruction being processed in the commit stage calls the SpecCheck `advance_FSM()` function to inspect the instruction's state. This function will change the state of the FSM if the conditions outlined in Section III-A are met and update the taint table as necessary. Any time a unique mis-speculation 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 write the flagged PC to an external file, as well as the corresponding instructions that cause changes to SpecCheck's state. SpecCheck will also report the total number of unique mis-speculated windows inspected by SpecCheck, as well as how many of those were marked as vulnerable at the end of the simulation period within the statistics output file. In total SpecCheck adds about 400 lines of code to the gem5 simulator.

Defining Micro-Visible Instructions. Microarchitecturally visible (μV) instructions are instructions that create externally observable effects. As defined in Section III-A, memory, floating

```
O3PipeView:fetch:1000000:0x150e:  MOV_R_M : ld rax, DS:[rcx]   // 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)
```

Fig. 4: An example of a flushed DynInst instruction on a simulated out of order x86 processor.

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 instruction [30]. The SpecCheck monitor examines each instruction's StaticInst pointer, static information about an instruction, to determine if an instruction is microarchitecturally visible or not by determining if the instruction is a load from memory, a branch (including returns), a call, or a floating point operation.

IV. Evaluation

The SpecCheck implementation in gem5 is evaluated using a variety of programs spanning multiple Spectre variants, Spectre V1 and V2 [1], [2], [22]. In addition, we tested SpecCheck results against 13 programs from the SPEC CPU 2017 benchmark suite [5] to show simulated time overhead and number of vulnerable windows. SpecCheck was evaluated using an out of order x86 CPU across all experiments We additionally evaluate SpecCheck on a series of known benign microbenchmarks (i.e. simple arithmetic or I/O programs) and mitigated Spectre attacks using fences.

Each proof of concept and microbenchmark is evaluated using gem5's system call emulation mode configured to use the `Deriv03` CPU with `LTAGE` branch predictor [35], 8GB of memory, 32kB, 8 way set-associative L1 instruction/data caches, and a 256kB 16 way set-associative L2 cache. Each program is statically compiled, and microbenchmarks were compiled without optimizations. SpecCheck does not require programs to be statically compiled to work correctly, but it does make the analysis of its results easier to reason about. Each program is simulated for a maximum of 150,000,000 instructions, longer than necessary to leak each secret byte in both Spectre proof-of-concepts.

A. Identifying Known Side-Channel Gadgets

To showcase the correctness of the SpecCheck debugging module, the finite state machine is run on known Spectre gadgets, such as the one shown in Figure 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 regions of code 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 two main Spectre variants, v1 (PHT) [2] and v2 (BTB) [1]: variants that remain largely unmitigated [49]. Each Spectre variant was compiled statically with x86. SpecCheck was able to identify all expected victim gadgets in each of the Spectre variants, ensuring that the debugging module is capable of providing an accurate depiction of the vulnerabilities in the transient execution paths. SpecCheck also discovers numerous, potentially malicious mis-speculated windows in the C standard library as well as other vulnerable code regions within each PoC, distinct from the chosen exploit gadget. These findings align with prior work [22].

```
1  # Branch conditional
2  0x401ae1 <+12>: mov    0xde3c9(%rip),%eax
3  0x401ae7 <+18>: mov    %eax,%eax
4  0x401ae9 <+20>: cmp    %rax,-0x8(%rbp)
5  0x401aed <+24>: jae    0x401d72 <victim+77>
6
7  # Branch target
8  0x401aef <+26>: lea    0xde3da(%rip),%rdx
9  0x401af6 <+33>: mov    -0x8(%rbp),%rax
10 0x401afa <+37>: add    %rdx,%rax
11 0x401afd <+40>: movzbl (%rax),%eax
12 0x401b00 <+43>: movzbl %al,%eax
13 0x401b03 <+46>: shl    $0x9,%eax
14 0x401b06 <+49>: cltq
15 0x401b08 <+51>: lea    0xe1b01(%rip),%rdx
16 0x401b0f <+58>: movzbl (%rax,%rdx,1),%edx
17 0x401b13 <+62>: movzbl 0xe05f6(%rip),%eax
18 0x401b1a <+69>: and    %edx,%eax
19 0x401b1c <+71>: mov    %al,0xe05ee(%rip)
20
21 # End of branch / not taken
22 0x401b22 <+77>: leave
```

Fig. 5: Disassembled victim code for a Spectre variant 1 attack gadget shown in Figure 1

Spectre V1. We first analyze the Spectre v1 exploit to verify that SpecCheck can successfully identify the potential vulnerability induced by the gadget shown in Figure 1 by analyzing a known proof of concept exploit [2]. The disassembled binary for the Spectre v1 gadget is shown in Figure 5. SpecCheck successfully identifies the start of the mis-speculation window at PC `0x401aef` (a `lea` instruction), the speculated target of the conditional branch at PC `0x401aed`. SpecCheck also identifies the CLoad of malicious data from PC `0x401af6`, which dereferences the pointer to the malicious address thus loading the secret byte, and continues to propagate

```
0000000000401b0f <victim>:
...
# Train BP
...
0x401b4b mov    0x1267ae(%rip),%rax #branch target
0x401b52 mov    (%rax),%rbx
0x401b55 call   *%rbx
...

# mis-speculated target
0000000000401ad5 <gadget>:
0x401ad5 nop
0x401ad9 push   %rbp
0x401ada mov    %rsp,%rbp
0x401add mov    %rdi,-0x8(%rbp)     #save secret ptr
0x401ae1 mov    -0x8(%rbp),%rax
0x401ae5 movzbl (%rax),%eax         #load secret byte
0x401ae8 movsbl %al,%eax
0x401aeb shl    $0xa,%eax           #shift secret
0x401aee cltq
0x401af0 lea    0xe6809(%rip),%rdx #transmitter
0x401af7 movzbl (%rax,%rdx,1),%eax #transmit secret
0x401afb movzbl %al,%eax
0x401afe pop    %rbp
0x401aff ret
```

Fig. 6: Disassembled gadget code for the Spectre BTB (variant 2) POC [1].

register taint until the transmitting instruction at `0x401b0f`, successfully demonstrating SpecCheck's ability to track speculatively loaded data through various instructions until reaching a transmission gadget.

Spectre V2. SpecCheck is also able to identify information leaks that occur during branch target injection (BTB) attacks, i.e. Spectre v2. This analysis uses the Spectre v2 POC code provided by prior work, which was selected for its easy portability to the gem5 simulator [1]. Spectre v2 relies on poisoning the target of indirect jumps to induce speculation which SpecCheck is able to identify [22]. Figure 6 shows the exploitable gadget in the Spectre v2 POC: the victim function contains the indirect jump instruction at PC `0x401b55`, which allows a mistrained branch prediction unit to jump to the malicious gadget at PC `0x401ad5`. This gadget, whose argument contains a pointer to the secret data, loads the secret byte from memory (PC `0x401ae5`) and encodes the secret in a covert channel (PC `0x401af7`). We demonstrate that SpecCheck is able to classify each component of the vulnerability in this gadget by successfully identifying the beginning of the mis-speculation window (`0x401ad5`), the malicious load from memory (`0x401ae5`), and the transmitting gadget (`0x401af7`).

Mitigated Spectre v1. SpecCheck was also evaluated on a mitigated Spectre v1 proof of concept using a load fence as shown in Figure 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 retire before executing subsequent instructions, thus mitigating any possibility for 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.

B. Microbenchmark Evaluation

To verify SpecCheck accuracy, the tool was also evaluated on a series of microbenchmarks that do not induce the required conditions. These microbenchmarks include a program to store and load simple variables to/from memory, perform arithmetic operations such as addition (as shown in Figure 7), and utilize the `printf()` function to print a string. SpecCheck identified no potential leaks in either the memory operations microbenchmarks or the arithmetic ones, 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 for benign programs that use speculative execution as intended.

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

Fig. 7: Example of an arithmetic microbenchmark

C. SpecCheck 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. These statistics were measured in overall seconds when running the SPEC CPU 2017 benchmark suite [5].

Each benchmark is simulated for a maximum of 150 million instructions or until the program completes and exits normally. Across all 13 benchmarks, SpecCheck incurs only a 4.05% overhead on average in simulation time, ranging between 1.8% in the best case (mcf) up to 9.8% in the worst case (cactuBSSN). 11 of the 13 benchmarks have less than 5% in simulation time overhead. Considering the increase in time to simulate processors, the overhead incurred by SpecCheck is reasonable and we do not expect the simulation overhead to become a hindrance to the adoption of the tool.

D. False Positives

To provide a conservative security guarantee, SpecCheck aims to avoid all instances of false negatives—where a malicious gadget is not identified as potentially malicious—and instead leans to have a larger false positive rate. To identify false positive and false negative rates, we inspected the unique mis-speculated windows (by their initial PC value) that we were expecting to be marked as vulnerable. We present these results in an accuracy table for each Spectre variant evaluated with SpecCheck, as shown in table II. Each entry in the table represents the number

Variant	Unique MW	True Positives	False Negatives	False Positives (FP)	FP rate
PHT (V1)	443	1	0	64	14.5%
BTB (V2)	441	1	0	60	13.6%

TABLE II: SpecCheck accuracy metrics for all unique mis-speculation windows (MW) across each Spectre variant.

Benchmark	Vulnerable Windows	Flushed Windows	Rate
cactuBSSN	804	3675	22%
exchange2	705	3324	21%
imagick	445	1908	23%
omnetpp	903	5607	16%
leela	107	791	14%
mcf	160	544	29%
x264	135	617	22%
nab	245	924	27%
xalancbmk	286	2526	11%
lbm	94	587	16%
bwaves	115	545	21%
deepsjeng	74	643	12%
xz	75	453	17%
Mean	319	1703	19%

TABLE III: Number of vulnerable and flushed windows identified by SpecCheck in the SPEC CPU 2017 Benchmark suite.

of unique mis-speculated PCs evaluated by SpecCheck. True positives are those mis-speculated windows (MW) that we knew corresponded to the attack gadget that were correctly identified as vulnerable. False negatives are MW that were incorrectly identified as benign. False positives are those MW that are incorrectly identified as vulnerable but after a manual inspection we did not find a correlation between the attack gadget and the PCs.

The 0% false negative rate is indicative of SpecCheck's strong accuracy 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 13-15%, 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. Of 13 benchmarks from SPEC CPU 2017 suite [6], SpecCheck reported 19% of vulnerable windows, with a minimum of 11% (xalancbmk) and a maximum of 29% (mcf) (Table III shows all the results).

V. Related Work

The only other piece of work that uses a FSM in a similar context is Spectify, which uses the FSM to construct a hardware defense mechanism against transient execution attacks [32]. Unlike SpecCheck, the Spectify FSM is solely limited to cache based side-channel attacks. SpecCheck aims to be agnostic to the underlying transmission gadget used during speculation, as various other, non-cache side channels exist [15], [45].

Another similar approach to SpecCheck is found in the work titled "Axiomatic Hardware-Software Contracts for Security" [30]. In this prior work, Mosier *et.al.* introduce the concept of Leakage Containment Models (LCMs) which 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 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.

CheckMate [41] 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 SpecCheck approach differs from the work in CheckMate primarily in the fact that SpecCheck is built in a tool that is ubiquitously used in computer architecture research, bypassing the obstacle of having to describe both the hardware and the attack behavior in a new environment. Second, SpecCheck analyzes software patterns in combination with pipeline state separated from the attack description. These are important differences, as SpecCheck is primarily focused on identifying vulnerabilities that arise during the software/hardware co-design process where CheckMate is exclusively used to verify a microarchitectural design implementation.

Fadiheh et al. [12] present work on a formalized approach characterizing transient execution vulnerabilities by extending prior work [13] to include out-of-order execution. Similar to CheckMate, this work is exclusively concerned with microarchitectural structures, differing from SpecCheck's analysis of software with pipeline state. SpecCheck aims to build upon the formalization from prior work by using a state machine to better fit the debugging framework offered by commonly used tools in the research community to provide a more holistic view of the system that includes both the software and hardware states.

Another piece of work that was recently published, IntroSpectre, enables the verification of existence of Meltdown-type attacks at the Register Transfer Level (RTL) [14]. IntroSpectre uses fuzzing to test a variety of paths for specific gadgets to identify if a Meltdown-type attack (leveraging data access exceptions) is possible. In contrast, SpecCheck is designed for generality rather than to test one specific type of attack and does

not need fuzzing since a user can simply run any attack they are looking to evaluate against the processor simulator using SpecCheck to identify vulnerable sequences of code.

Finally, Speculative Taint Tracking (STT) presents one of the many hardware protection scheme for mitigating Spectre attacks by tracking all speculatively accessed data [53]. 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 meant to be a defense mechanism, the tool leverages the proposed FSM to determine if a leak is possible. The SpecCheck tool can be used to systematically verify if defense mechanisms like STT in fact do block transient execution attacks.

Various other mechanisms for defending against Spectre have been proposed in recent work [10], [38], [47], [52]. However, like many of the examples above, these pieces of work present active defense mechanisms for transient execution attacks, and are perfect candidates for security evaluation using SpecCheck.

A. Future Work

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 the 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 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) implementation. The SpecCheck tool will be extended to reason about other types of microarchitectures, such as in-order processors and processors with different pipeline designs. 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. Furthermore we will continue evaluating the accuracy of SpecCheck using different branch predictors included in gem5 (BiModeBP, LocalBP, and TournamentBP) as well as varying memory system implementations to provide a wider range of debugging opportunities for secure hardware design.

VI. Conclusion

SpecCheck introduces a novel framework for performing a systematic security verification of an application running on a specific microarchitecture. SpecCheck identifies regions of code as being vulnerable (or not) to transient execution attacks through a register-based finite state machine. The SpecCheck implementation in the gem5 simulator for 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 finite 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 conservative conclusion, successfully identifying vulnerable regions of code in two main Spectre variants. SpecCheck leverages the pipeline state information to pinpoint exact regions in a program's execution flow that may be vulnerable and provides support for further analysis of the speculative execution implementation by tracking all areas of mis-speculation. This analysis feature allows users to identify regions of code 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.

Acknowledgment

We would like to thank the anonymous reviewers who have helped in shaping the manuscript to its current form. We would also like to thank Ampere Computing for the generous donation of the equipment which was used in the experiments mentioned in this article.

Appendix

A. Abstract

This artifact presents the implementation of SpecCheck, a gem5 extension for identifying potentially malicious regions of transient execution. We implement the SpecCheck finite state machine, described in our paper, for an out of order x86 CPU, and verify its strong security guarantee by identifying known malicious transient execution in 2 Spectre variants. By logging all maliciously classified mis-speculation windows to an external file, we show that our artifact successfully marks proof of concept gadgets. As a fork of the official gem5 repository, SpecCheck requires minimal software dependencies and can be easily built on Ubuntu.

B. Artifact check-list

- **Program: gem5**
- **Compilation: SCons 3.0 (gem5) and GCC/G++ 11 (POCs)**
- **Data set: Spectre v1 and v2 POCs**
- **Run-time environment: Ubuntu 20.04 LTS (Focal Fossa)**
- **Metrics: Identifying known transient execution gadgets, false positive rates**
- **Output: Program Counters marked as vulnerable, number of vulnerable mis-speculated windows**
- **Experiments: emulation of the two Spectre variant attacks on gem5 with SpecCheck enabled**
- **How much disk space required (approximately)?: 13GB (after building)**
- **How much time is needed to prepare workflow (approximately)?: roughly 30 min to build from scratch**
- **How much time is needed to complete experiments (approximately)?: upwards of 10 minutes per POC**
- **Publicly available?: Yes**
- **Archived (provide DOI)?: 10.5281/zenodo.8326829**

C. Description

1) How to access: SpecCheck artifacts are located at the following github repository on the SpecCheckPACT branch:

https://github.com/zmckevitt/SpecCheck/.

The archived Zenodo repository is located here:

https://zenodo.org/record/8326829

with DOI 10.5281/zenodo.8326829.

2) Hardware dependencies: The gem5 simulator can be built and run on both x86 and ARM systems, but SpecCheck was only tested with the x86 architecture as the POC attacks in consideration only work with the x86 ISA. Each POC must be built for the specific ISA that the simulator is built with, in the case of our experiments that was x86. Each Spectre proof of concept is provided as a precompiled, statically linked x86 binary in the `pocs/precompiled/` directory, with the source code for each variant in `proc/`. To manually compile proof of concepts for experimental use, statically compile each variant using the Makefile provided (requires x86). More information about each proof of concept is provided in section IV-A.

3) Software dependencies: This artifact is built upon the gem5 simulator [4] and can be easily installed on Ubuntu 20.04. Necessary dependencies include Scons and Python3. If you do not already have Scons, install it with:

```
$ sudo apt install scons -y
```

Each Spectre proof of concept is provided as a precompiled, statically linked binary. To build each POC from scratch, use GCC/G++ 11 (Ubuntu 20.04).

4) Data sets: SpecCheck was evaluated using 2 Spectre variants:

- Spectre v1 (Pattern History Table) [22] (POC source [2])
- Spectre v2 (Branch Target Buffer) [22] (POC source [1])

Each proof of concept was modified to work within the gem5 simulator. Each POC source is included in the `pocs/` directory, and precompiled x86 binaries of each POC are located in `pocs/precompiled/`. Each POC can be built from source using `make` from within the `pocs/` directory. Each precompiled binary was compiled with GCC/G++ 9.3.0 for Ubuntu 20.04 with an x86 target. Note that each POC *must* be built for an x86 target as they use x86 specific timing instructions to recover secrets from the side-channel.

Each POC contains a known transient execution gadget to be identified by SpecCheck, and each gadget is described in Section IV-A of the paper. The program counter of each gadget to be identified by SpecCheck can be found in Table IV. Each gadget PC can be verified by object dumping the binary files (if on an x86 host) and comparing each PC to the gadgets shown in Section IV-A:

```
$ objdump -d pocs/spectre_v*_x86 | less
```

5) Models: SpecCheck makes use of gem5's x86 Out Of Order (O3) CPU model where binaries are run with Syscall Emulation (SE) mode.

D. Installation

SpecCheck is an extension of gem5 and can be easily installed on modern Ubuntu after installing the necessary prerequisites described in section C3. First, clone the repository:

```
$ git clone https://github.com/zmckevitt/SpecCheck.git
$ cd SpecCheck/
```

While inside the `SpecCheck/` directory, set your gem5 path for use in experiment scripts:

```
$ export GEM5_PATH=$(pwd)
```

Lastly, build x86 gem5:

```
$ scons build/X86/gem5.opt -j$(nproc)
```

E. Experiment workflow

Experiments can be simulated with or without SpecCheck enabled. To run an experiment with SpecCheck enabled, use the `sc.sh` script:

```
$ bash sc.sh <stats file> <path to bin>
```

Where the user provided statistics file is written to `m5out`. To run a gem5 experiment without SpecCheck enabled:

```
$ bash run.sh <stats file> <path to bin>
```

Note that all experiments in this artifact will have SpecCheck enabled. After each experiment finishes, all respective statistics files can be found in the `SpecCheck/m5out/` directory. For a specified `outfile`, `SpecCheck/m5out/outfile` contains standard gem5 statistics as well as 4 additional SpecCheck statistics:

- system.cpu.commit.speccheckFlushedWindows
- system.cpu.commit.speccheckUniqFlushedWindows
- system.cpu.commit.speccheckVulnWindows
- system.cpu.commit.speccheckUniqVulnWindows

SpecCheck also creates a log of all Program Counters flagged as vulnerable in `gem5/m5out/outfile.scout`.

F. Evaluation and expected results

Begin the evaluation process by running each experiment and generating respective statistics files:

```
$ bash sc.sh v1.txt pocs/spectre_v1_x86
$ bash sc.sh v2.txt pocs/spectre_v2_x86
```

After running each experiment, determine the number of unique flushed/vulnerable windows identified by SpecCheck:

```
Reading 6 bytes starting at 0x4b7008:
reading 0x4b7008...success: 0x53='S'
reading 0x4b7009...success: 0x45='E'
reading 0x4b700a...success: 0x43='C'
reading 0x4b700b...success: 0x52='R'
reading 0x4b700c...success: 0x45='E'
reading 0x4b700d...success: 0x54='T'

Exiting @ tick 2450436000 because exiting with \
    last active thread context
```

Fig. 8: Spectre v1 and v2 gem5 output

```
$ grep "speccheckUniq" m5out/v*.txt
```

Next, ensure that the known gadgets are properly discovered in the `scout` file (note that `.scout` files exclude the `0x` hex prefix):

```
$ grep <gadget PC> m5out/v*.txt.scout
```

Results should be similar to those found in the table below.

POC	Uniq Flushed	Uniq Vulnerable	Gadget PC
Spectre v1	443	65	0x401aef
Spectre v2	441	61	0x401ad5

TABLE IV: Table of expected SpecCheck results

Each gadget leaks each byte of the string `"SECRET"`, and each leaked byte is printed to stdout upon successful recovery of the secret byte from the side channel.

Successful runs of each POC in gem5 are shown in figure 8:

References

[1] R. H. Anton Cao, "Spectre btb example code," https://github.com/Anton-Cao/spectrev2-poc, 2020.

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

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin *et al.*, "The coq proof assistant reference manual," *INRIA, version*, vol. 6, no. 11, 1999.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[5] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[6] ——, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[7] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[8] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[9] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 288–28 815.

[10] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[11] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, 2014.

[12] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. 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)*, 2020, pp. 1–6.

[13] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 994–999.

[14] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 874–887.

[15] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[16] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[17] R. Guanciale, M. Balliu, and M. Dam, "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Conference on Computer and Communications Security (CCS)*, 2020.

[18] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[19] Y. 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*. IEEE, 2014, pp. 19–24.

[20] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2013, pp. 99–106.

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[22] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[23] T. S. Lehman, A. D. Hilton, and B. C. Lee, "Poisonivy: Safe speculation for secure memory," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[25] J. Longo, E. D. 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.

[26] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "{DOLMA}: Securing speculation with the principle of transient {Non-Observability}," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1397–1414.

[27] E. Love, Y. Jin, and Y. Makris, "Enhancing security via provably trustworthy hardware intellectual property," in *2011 IEEE international symposium on hardware-oriented security and trust*. IEEE, 2011, pp. 12–17.

[28] ——, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2011.

[29] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[30] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 72–86.

[31] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 106–119.

[32] A. Pashrashid, A. Hajiabadi, and T. 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*, 2022, pp. 1–9.

[33] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," *International Symposium on Computer Architecture (ISCA)*, 2021.

[34] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*. Springer, 2019.

[35] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[36] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ACM Sigplan Notices*, 2004.

[37] M. Taram, A. Venkat, and D. 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 395–410. [Online]. Available: https://doi.org/10.1145/3297858.3304060

[38] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. 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*, 2009, pp. 109–120.

[39] S. Tizpaz-Niari, P. Černỳ, S. Sankaranarayanan, and A. Trivedi, "Efficient detection and quantification of timing leaks with neural networks," in *International Conference on Runtime Verification*. Springer, 2019, pp. 329–348.

[40] S. Tizpaz-Niari, P. Cerny, and A. Trivedi, "Data-driven debugging for functional side channels," *The Network and Distributed System Security (NDSS) Symposium*, 2022.

[41] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 947–960.

[42] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[43] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. 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 [48].

[44] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *Security and Privacy (S&P)*, May 2021.

[45] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 679–697.

[46] L. Wei, B. Luo, Y. Li, Y. Liu, and Q. 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*, 2018, pp. 393–406.

[47] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

[48] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [43].

[49] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[50] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–36, 2021.

[51] J. Yang and C. 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*, 2010, pp. 99–110.

[52] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[53] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. 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*, 2019, pp. 954–968.

[54] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2011, pp. 67–70.