IntroSpectre: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities

Moein Ghaniyoun[†], Kristin Barber[†], Yinqian Zhang[§], Radu Teodorescu[†]
[†]Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
{ghaniyoun.1, barber.245, teodorescu.1}@osu.edu
[§]Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
yinqianz@acm.org

***Abstract*—Transient execution vulnerabilities originate in the extensive speculation implemented in modern high-performance microprocessors. Identifying all possible vulnerabilities in complex designs is very challenging. One of the challenges stems from the lack of visibility into the transient micro-architectural state of the processor. Prior work has used covert channels to identify data leakage from transient state, which limits the systematic discovery of all potential leakage sources.**

This paper presents IntroSpectre, a pre-silicon framework for early discovery of transient execution vulnerabilities. IntroSpectre addresses the lack of visibility into the micro-architectural processor state by integrating into the register transfer level (RTL) design flow, gaining full access to the internal state of the processor. Full visibility into the processor state enables IntroSpectre to perform a systematic leakage analysis that includes all micro-architectural structures, allowing it to identify potential leakage that may not be reachable with known side channels. We implement IntroSpectre on an RTL simulator and use it to perform transient leakage analysis on the RISC-V BOOM processor. We identify multiple transient leakage scenarios, most of which had not been highlighted on this processor design before.

I. Introduction

Transient execution vulnerabilities, originally uncovered by the Meltdown [25] and Spectre [22] attacks have exposed fundamental security weaknesses in modern processor designs. The discovery of these vulnerabilities has lead to an explosion of transient execution attack variants [5], [7]–[9], [17], [27], [38]–[40], [48] over the last few years.

In response, a robust body of defenses has been proposed [2], [6], [14], [18]–[21], [26], [35]–[37], [42], [51], [53]–[55]. While prior work has made significant advances in addressing many of the attack variants, new transient execution vulnerabilities continue to be discovered [34], [47], [49].

A major challenge in discovering all the circumstances in which transient execution can lead to secret data leakage is posed by the nature of transient execution itself. By definition, transient microarchitectural state is rolled back when illegal data accesses, or mispeculated/invalid instructions are executed. This creates a major roadblock to fully understanding the negative side-effects of transient execution when only observing the architectural state of the system. As a result, prior work that has built tools for detection of transient execution vulnerabilities [30], [32], [52], has had to rely on covert channels to determine if secret leakage was possible on a given system. This significantly limits the visibility into potential leakage to only known side/covert channels, hindering a systematic analysis.

In this paper, we present IntroSpectre, a pre-silicon framework for early discovery of Meltdown-type transient execution vulnerabilities, which are rooted deeply in the microarchitectural data access speculation (as opposed to the control speculation abused by Spectre-type vulnerabilities).

In order to address the lack of visibility into the microarchitectural processor state, our framework integrates into the register transfer level (RTL) design flow, gaining full access to the internal state of the processor. Full visibility into the state space enables IntroSpectre to perform leakage analysis that includes all microarchitectural structures, allowing it to highlight *potential* leakage that may not be reachable with known side channels. The processor designer can therefore use IntroSpectre to discover and analyze potential transient execution leakage in the design before the processor is taped-out, as part of the traditional functional verification process.

The IntroSpectre framework consists of two main components: a Gadget Fuzzer and a Leakage Analyzer. The Gadget Fuzzer uses fuzzing to generate randomized code sequences that trigger various forms of speculative execution, as well as attempted data access across isolation boundaries. The Gadget Fuzzer builds test code sequences out of random selections of simple gadgets, which are predefined code snippets designed to cover the space of possible speculation primitives, isolation boundaries and access instructions. The role of these test code sequences is to set up secret data targets and generate a variety of random scenarios for accessing that data across isolation boundaries. The goal is to then observe if secret data is found in microarchitectural structures from which it could be extracted by an attacker. To that end, the fuzzing code rounds are run through an RTL simulator, which generates a detailed execution log. The Leakage Analyzer parses the execution log

This work was funded in part by Intel Corp. under the Side Channel Academic Program and by the Air Force Research Laboratory under the Assured and Trusted Microelectronics Solutions award FA8650-20-C-1719.

and searches for leakage of secret data deliberately inserted in each fuzzing round. A detailed execution model constructed by the gadget fuzzer is used throughout the framework to assist with the code generation and leakage analysis.

We implement INTROSPECTRE on top of Verilator [1], an open-source RTL simulator and use it to analyze the RISC-V BOOM processor [10]. We discover potential leakage arising from incorrect handling of permissions checks in several cross-boundary accesses, or aggressive line-fill buffer (LFB) fill policies. We also find that the hardware prefetcher may exacerbate some forms of leakage. INTROSPECTRE identifies potential leakage in the physical register file, line-fill buffer and write-back buffer of the tested processor. Overall we discover 13 distinct, not previously documented, Meltdown-type transient leakage vulnerabilities in the target processor.

This paper makes the following contributions:

- Presents INTROSPECTRE, the first RTL-level framework for detection of transient execution vulnerabilities.
- Presents a methodology for feedback-driven fuzzing of test gadgets that increases the effectiveness of the generated test code.
- Implements INTROSPECTRE on top of Verilator, an open-source RTL simulator.
- Demonstrates the effectiveness of INTROSPECTRE by detecting multiple Meltdown-type vulnerabilities and potential leakage scenarios in the RISC-V BOOM processor.

The rest of this paper is organized as follows: Section II provides background on transient execution attacks and related work on exploit detection and synthesis. Section III outlines the threat model. Sections IV–VI present the INTROSPECTRE framework design. Section VII details the implementation and experimental methodology. Section VIII presents a number of use case scenarios for transient leakage detection and Section IX concludes.

II. BACKGROUND AND RELATED WORK

A. Transient Execution Attacks

Transient execution attacks can be categorized along two primary axes: **(1) how transient effects are co-opted** and, **(2) the channel used to covertly communicate information**. The second axis has mostly explored use of the cache hierarchy (occupancy, replacement metadata, coherence state); but others have shown it practical to utilize port contention and AVX instruction latency as well [3], [7], [39]. Attacks can then be further placed into three classes, based on the first axis:

(a) **Metldown-type** [25]: leverage transiently executed instructions in the shadow of a fault. The canonical example and namesake for this category, transiently executed instructions past a pending exception on on an illegal memory load. This type of permission-bypassing manifests when protection checks are performed in parallel to actual data accesses. Other examples include attacks abusing transient execution past an illegal register read, or past lazily cleaned register state upon a context-switch [5], [40].

(b) **Spectre-type** [22]: leverage misprediction in the processor, such as the pattern history table, branch target buffer [57], return stack buffer [23], [27], or store-to-load forwarding [17], [28] to trigger the transient execution of instructions of interest to the attacker. Shared prediction mechanisms allow attackers to manipulate prediction outcomes across security boundaries, enabling the coercion of victim processes into executing down a selected transient path–ultimately leaking their own secrets through pre-determined disclosure gadgets.

(c) **Microarchitectural Data Sampling (MDS)-type** [9], [47]: leverage in-flight data that is stored in fill and other buffers, and which is forwarded without adequate permission checks. Examples include RIDL [48], which found that if load and store instructions are ambiguous, some processors may speculatively forward data from the store buffer to the load buffer; ZombieLoad [38] that demonstrated line fill buffers are accessible by all logical CPUs and make no distinction between processes or privilege levels when forwarding data; CrossTalk [34] which revealed the ability to leak data from staging buffers shared across CPU cores; and CacheOut [49] that showed it is possible to select which data is leaked.

There are several cross-cutting factors across these axes which play a role in determining the success of attacks, such as the ability to affect speculation primitives, speculative window size achievable, latency of the chosen disclosure gadget, timing reference resolution and retention time of the covert channel. Defenses attempt to reduce or eliminate the feasibility of these factors, in order to minimize the likelihood of attack success.

B. Attack Mitigations

Several approaches to defending against information leakage through transiently executed instructions have been proposed, both by industry and academia.

Industry responded to the initial disclosure of these exploits in 2018 with guidelines for application developers, which were essentially to insert serializing instructions directly after any vulnerable code sections to constrain speculation. Additionally, micro-code updates continue to be released by processor vendors as new vulnerabilities are disclosed, seeking to remedy sources of leakage in processor functionality where possible. For example, Intel provided a patch, along with system software support, to flush and disable shared branch predictor structures at runtime [19]. A comprehensive list of micro-code updates and security guidance can be found in Intel's Security Advisory portal [18]. Google engineers developed the *retpoline* [46] compiler-level steering scheme to isolate indirect branches. Kernel page table isolation (KPTI) [14] was integrated into the Linux kernel, which removed kernel page mappings from user-space and eliminated the ability to conduct Meltdown attacks.

Academia: the computer architecture community has advocated for modifications in processor design to enhance the security of transient execution. Solutions proposed have mostly followed two general protection schemes: **(1)** *invisible* *speculation* mechanisms–such as Invisispec [53], SafeSpec [20], Delay-on-Miss [36] and MuonTrap [2]–where microarchitectural state is hidden in dedicated shadow buffers while computing in a speculative shadow or undone following a misprediction, in the case of CleanupSpec [35]; **(2)** mechanisms *constraining* the use of speculative data from within the pipeline–such as SpecShield [6], STT [55], NDA [51], SDO [54] and DOLMA [26]–until data is no longer speculative. Other defenses require system software support, such as setting up cache partitions to isolate memory accesses across processes with DAWG [21]; or the annotation of secret data in ConTExT [37] and Context-Sensitive Fencing [42], which use taint-tracking to detect malicious gadgets and/or subvert the leaking of sensitive data.

C. Exploit Detection and Synthesis

Prior work has explored methods by which to synthesize unique transient execution exploits, as well as techniques to automatically detect vulnerabilities which allow for covert channel formation and transmission. Approaches taken so far include utilizing forms of fuzzing, formal analysis and information-flow tracking.

1) Fuzzing-based Approaches: Transynther [30] is aimed at discovering Meltdown-type attack variants by fuzzing seeds of known exploits, applying random mutations to various attributes associated with the faulty load and utilizing the cache as a covert channel in proving the existence of a leak. SpecFuzz [32] uses a dynamic testing methodology for discovering Spectre-type transient execution vulnerabilities. The tool simulates speculative execution in software by forcing the direct execution of all reachable code paths. Memory accesses that would otherwise be hidden are made visible to integrity checkers which can then identify potential disclosure gadgets. DifFuzz [31] fuzzes program inputs with the goal to discover side-channels. DifFuzz analyzes two copies of the same program, with different secret values but the same inputs. A cost metric is then computed based on the difference in side-channel measurements (resource usage, specifically, the number of instructions executed and memory footprint) between the two applications. SpeechMiner [52] leverages a fuzzing framework to identify Meltdown-type vulnerabilities in existing processors. Sequences of x86 instructions are constructed using templates and executed in a controlled environment. A cache-based covert channel is used to determine if vulnerabilities exist. ABsynthe [13] introduces a framework for finding contention-based side-channels by measuring the relative effects on timing between different instructions in real hardware and formulating an optimization problem based on the maximization of said effects.

2) Formal Specifications: Spectector [15] uses symbolic execution to automatically detect leaks from transient execution by verifying if speculatively executed instructions leak more than committed instructions. Checkmate [44] is a tool for exploit synthesis based on happens-before graphs, which encode microarchitectural events and event orderings. Relational model finding is applied to determine sets of edges which satisfy given constraints and represent the behavior of a specific exploit, from which actual exploit programs can be synthesized. UPEC [12] constructs 2-safety hardware properties by shaping formal definitions of security in the domain of transient execution attacks. The hardware properties are then checked to verify if the program is executed uniquely with respect to the secret.

3) Limitations of Existing Frameworks: Frameworks like SpeechMiner, Transynther, SpecFuzz or DifFuzz face the twin challenges of very limited microarchitectural implementation information and the lack of visibility into microarchitectural state. SpeechMiner, for instance, uses abstract implementation models that may be incomplete and could miss subtle behaviors. SpeechMiner also has to rely on covert channels to identify leakage. This means the source of the leakage has to be known or suspected, in order to be leaked by the covert channel. This significantly limits the possibility of discovering new leakage sources.

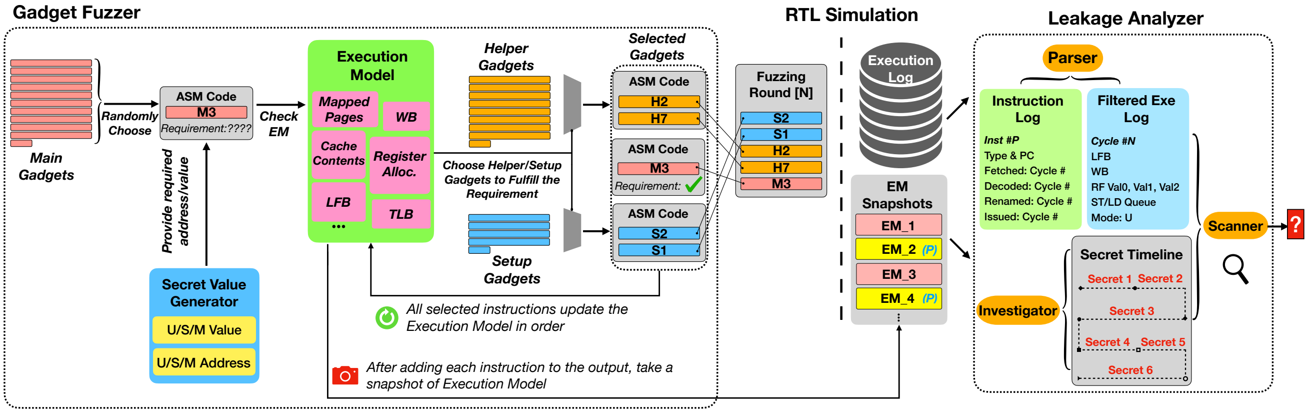
D. Hardware Verification

1) Information-flow Tracking: Information-flow tracking mechanisms for hardware designs seek to capture the routes of sensitive data, through architectural [41] or gate-level structures, in order to verify the absence of exposure to an attack surface. SecVerilog [56] introduces a type system extension to the Verilog HDL, where information flow policies for a design can be specified and checked statically at compile-time. GLIFT [43] proposes an architecture able to track these flows at run-time. These approaches are effective at identifying a broad range of information leakage scenarios. However, they generally require the redesign of the target processors, with increased complexity, which may not be practical for all applications.

2) Side Channel Analysis: SVF [11] proposes a quantitative method to evaluate side-channel information leakage of a design by measuring the correlation of side-channel information patterns to the actual execution patterns. CSV [58] builds on top of SVF by limiting the scope of side channels to only cache side-channel attacks and improves the accuracy of the resulting metric. These approaches are targeted to side channel characterization and do not directly address the sources of transient execution leakage.

3) Constrained Random Verification: Constrained Random Verification (CRV) is a design verification method used widely in industry [29], primarily to verify functional correctness. CRV uses automatic generation of random input vectors that follow a set of pre-defined constraints. As the stimuli generation is done automatically, CRV can be especially useful in uncovering corner cases in large and complex designs. CRV can be incorporated in hardware description languages [16], [33] such as SystemC providing faster verification for designs modeled in higher levels of abstraction. In general, CRV methods do not rely on feedback-driven input generation.

INTROSPECTRE uses approaches similar to CRV to randomize test gadget parameters. However, CRV does not generally include a sufficiently-expressive feedback mechanism to guide



All selected instructions update the Execution Model in order

After adding each instruction to the output, take a snapshot of Execution Model

Fig. 1: High-level illustration of the INTROSPECTRE workflow. Framework includes two main components: a Gadget Fuzzer and a Leakage Analyzer. INTROSPECTRE is designed to run in conjunction with an RTL simulator.

input selection. INTROSPECTRE relies on a detailed execution model to generate highly relevant test sequences, rather than rely on random selection. Also, unlike CRV, INTROSPECTRE relies on more abstract building blocks (instructions and gadgets) for test generation rather than unit-level/ensemble of units-level testing. This level of abstraction allows INTROSPECTRE to expose leakage that requires complex software-initialized states, such as OS isolation boundaries, secret-value prefetching in various microarchitectural structures, etc. Other recent work [45] has also similarly observed the benefits of fuzzing-like approaches to hardware verification.

III. THREAT MODEL

INTROSPECTRE targets transient execution vulnerabilities that can potentially leak data across isolation boundaries. We target Meltdown-type vulnerabilities, in which permission checks are lazily enforced relative to the data access. INTROSPECTRE is designed for pre-silicon verification and we assume access to the RTL implementation of the processor under test. We consider cases of potential leakage in which secret data that crosses isolation boundaries is found in microarchitectural structures while user-level code is executing.

IV. INTROSPECTRE FRAMEWORK DESIGN

Register-transfer level (RTL) representation is a low-level detailed implementation of a hardware design. An RTL design is sufficiently detailed to be synthesized and deployed into hardware. An RTL simulation uses the RTL representation of a design to produce a true and detailed simulation of the final hardware product. These simulations capture the precise timing and state of every single logic and memory element. They can therefore provide an accurate and very detailed representation of the runtime behavior of the hardware product. Unlike architectural simulations, which are approximations of a design's behavior, RTL simulations rely on the actual hardware implementation, and are therefore very accurate and detailed. RTL simulations can provide full visibility into the state of the processor and are therefore extensively used for functional debugging and testing in industry.

The INTROSPECTRE framework consists of two main components: **The Gadget Fuzzer** and **The Leakage Analyzer** (Figure 1). The gadget fuzzer (Section V) is responsible for generating stress test code sequences that use speculation primitives to attempt to access and leak privileged data. The test sequences are run through the RTL simulator, which generates a detailed execution log. The leakage analyzer (Section VI) is used to parse the RTL log to determine if secret leakage is possible. A detailed execution model (Section V-C) constructed by the gadget fuzzer is used throughout the framework to assist with the code generation and leakage analysis.

V. THE GADGET FUZZER

The primary role of the gadget fuzzer is to generate relevant test sequences that exercise a wide range of possible transient execution leakage scenarios. The gadget fuzzer borrows some of the principles of fuzzing-based test case generation, including feedback-based selection. In order to keep the test cases focused on transient execution vulnerabilities we use a set of predefined code gadgets designed to cover the space of possible speculation primitives, isolation boundaries and memory access instructions. The gadget fuzzer consists of three main components that work jointly to generate targeted test sequences: *Stress-Test Gadgets*, *Execution Model Generator* and *Secret Value Generator* (Figure 1). The gadgets are short code snippets that are combined by the fuzzer to generate leakage test sequences. An execution model is constructed in parallel with the test sequences and provides feedback to the fuzzer to ensure that the resulting test meets certain functionality requirements. This helps prune the very large space of possible test sequences by selecting the ones most likely to lead to useful outcomes.

Table I lists the gadgets we use in our INTROSPECTRE implementation on the RISC-V BOOM processor.

	Main Gadgets	Description	Permutations
M1	Meltdown-US	Retrieve a value from supervisor memory while executing in user mode.	8
M2	Meltdown-SU	Retrieve a value from a user page while executing in supervisor mode when SUM bit of `sstatus` CSR is clear.	8
M3	Meltdown-JP	Jump to a user address and execute the stale value.	16
M4	PrimeLFB	Prime line fill buffer (LFB) entries with known values from Secret Value Generator.	8
M5	STtoLD Forwarding	Generate store and load instructions with overlapping addresses.	256
M6	FuzzPermissionBits	Test different combinations of permission bits for a user page. Each page table entry (PTE) has 8 permission bits.	256
M7	ContExeWritePort	Create contention on execution units with the same write port.	1
M8	ContExeUnit	Create contention on unpipelined execution units.	1
M9	RandomException	Randomly choose an excepting instruction and execute it with a bound-to-flush method.	10
M10	TorturousLdSt	Randomly generate loads and stores back to back from/to addresses that the processor has already interacted with.	16
M11	AMO-Insts	Randomly execute one atomic memory operation (AMO) instruction.	14
M12	Load-WB-LFB	Generates loads from values currently in write-back buffer or line fill buffer.	64
M13	Meltdown-UM	Retrieve a value from machine-mode protected memory (PMP) while executing in supervisor/user mode.	8
M14	ExecuteSupervisor	Jump to a supervisor memory location and start executing instructions.	2
M15	ExecuteUser	Jump to an inaccessible user memory location and start executing instructions.	2
	Helper Gadgets	**Description**	**Permutations**
H1	LoadImmUser	Use Secret Value Generator to generate a user memory address.	1
H2	LoadImmSupervisor	Use Secret Value Generator to generate a supervisor memory address.	1
H3	LoadImmMachine	Use Secret Value Generator to generate a machine memory address.	1
H4	BringToMapping	Create a mapping for a user page with full permissions.	8
H5	BringToDCache	Load a memory location to the data cache through bound-to-flush load.	8
H6	BringToInstCache	Load a memory location to the instruction cache through bound-to-flush jump.	2
H7	Start/FinishDummyBranch	Create dummy branches where all instructions in between are going to be squashed.	8
H8	SpecWindow	Open speculative windows of different sizes.	4
H9	DummyException	Raise an exception to change the execution privilege in order to execute a setup gadget.	1
H10	Long/ShortDelay	Insert variable delays in before execution of main gadgets.	4
H11	FillUserPage	Fill a user page with data values that correlate with the page's address.	8
	Setup Gadgets	**Description**	**Permutations**
S1	ChangePagePermissions	Modify user pages permissions bits as needed for the main gadgets.	1
S2	CSRModifications	Modify supervisor/machine CSRs for the main gadgets.	1
S3	Fill/FlushSupervisorMem	Fill/Flush supervisor memory pages (4KB) with values generated by Secret Value Generator.	1
S4	Fill/FlushMachineMem	Fill/Flush machine-only memory pages (4KB) with values generated by Secret Value Generator.	1

TABLE I: INTROSPECTRE gadget types with a brief description of their intended functionality. Permutations indicates the number of distinct variants available for each gadget.

A. Stress-Test Gadgets

We use simple gadgets as the building blocks for the stress test code sequences. We use gadgets as an input to the Fuzzer, rather than individual instructions in order to keep the space of possible test outputs focused. However, to increase test entropy, these gadgets are randomly selected and assembled in random order by a Fuzzer module. In addition, each gadget has multiple parameters that are randomly set by the Fuzzer. INTROSPECTRE uses three types of gadgets to construct test sequences: Main Gadgets, Helper Gadgets and Setup Gadgets.

Main Gadgets represent the core of the leakage test sequences. They include speculation primitives and data access instructions. Many gadgets are based on known attacks, while others are added to ensure all documented speculation primitives are covered. In the current implementation of INTROSPECTRE we focus on Meltdown-type vulnerabilities, in which secrets are leaked across various isolation boundaries. The speculation primitives we target are primarily exception-causing instructions.

While several of the main gadgets are generated based on known transient execution attacks, others are intended to exercise different speculation primitives and isolation boundaries, even if no clear leakage channel can be defined a priori. This is achieved by gadgets such as *FuzzPermissionBits* ($M6$ in Table I) in which the fuzzer randomly changes the permission bits of target pages, while performing different loads and stores to these pages. This approach can help reveal novel transient execution vulnerabilities by exercising uncommon accesses and isolation primitives. INTROSPECTRE can identify potential leakage of "secret" data in microarchitectural structures, without having to include a leakage channel in the test sequence. This allows the framework to highlight *potential* leakage that may not yet have a known leakage channel.

Gadgets are designed to be composable and allow for data communication, where the output of one gadget becomes the input of another. In addition, multiple gadgets can be composed to execute together within the same speculation window. The execution model is used to ensure gadgets communicate through memory or registers, and to estimate the effect of the gadgets' execution on the architectural and microarchitectural state of the system.

The main gadgets represent the core of the speculation primitive and access instructions. In order to create a variety of conditions under which secret leakage can be observed, additional helper gadgets are needed. For example, for a simple *Meltdown-US* gadget ($M1$ in Table I) the intended behavior is to load a secret value from supervisor memory while the code is running in user mode. The main gadget itself consists of a single load instruction. However, certain microarchitectural conditions are required for each main gadget to execute as intended. For the *Meltdown-US* gadget, a known secret value stored in a known supervisor memory location is needed.

Helper Gadgets are used to establish the predefined conditions needed by the main gadgets to work as intended. For example, for Meltdown-US another requirement is that the target supervisor address should be present in L1 data cache. A helper gadget, *BringToDCache* ($H5$), is constructed to prefetch the required memory location in the cache. However, the required memory address will not be available in L1 data cache immediately after executing this helper gadget. The fuzzer uses the Long/Short Delay ($H10$) helper gadget to make sure the data is cached in the L1 data cache before the *Meltdown-US* ($M1$) gadget is executed.

Setup Gadgets are responsible for setting up the needed architectural/microarchitectural state for the main gadgets in supervisor/machine mode. Unlike helper gadgets, which run in user mode, setup gadgets are intended to prime the system with state that can only be changed in supervisor/machine mode. For example, we implement a simple *ChangePagePermissions* ($S1$ in Table I) gadget that assigns permissions to a target page. This gadget is preceded by a gadget that deliberately raises an exception in order to elevate the execution privilege to supervisor mode. The fuzzer inserts setup gadgets like *ChangePagePermissions* in the exception handler code where they will be executed with supervisor privileges. After the exception is handled and the setup code executes, control is returned to the main gadget in the test sequence.

In Listing 1 we show an example fuzzing round that includes all three types of gadgets combined to replicate the Meltdown-US behavior. First, a supervisor page is populated with secrets corresponding to its address by executing *S3* setup gadget in supervisor mode. Next, a random address is chosen from this page using *H2* helper gadget and then *H5* gadget is executed to bring the supervisor secret into the L1 data cache and also update the TLB with the new mapping. *H5* helper gadget utilizes a long-latency chain of dependent divide instructions to delay branch resolution providing enough time for the subsequent load to perform address translation and secret prefetching before being flushed from the pipeline. Now, we have the supervisor secret in the LFB and the next step is to use *H10* to wait for the secret to propagate into the L1 data cache. Finally, the *M1* main gadget retrieves the data from the cache. In order to suppress the page fault exception raised by this illegal access, the load instruction (*M1*) can be placed behind a mispredicted branch (*H7* helper gadget).

The decision to include setup gadgets and helper gadgets is based on the state of the execution model. For instance, after the main gadget *M1* is selected, the fuzzer checks *M1*'s list of requirements against the current state of the execution model. If $kernel\_addr$ is not present in the L1 Data Cache model, the fuzzer selects one of the prefetching gadgets (in this example *H5*) to execute before *M1*.

B. Secret Value Generator

The IntroSpectre fuzzer uses a secret value generator to produce memory addresses and populate them with "secret" data values that can be later observed in the execution log. Having memory pages with different privileges populated with known "secret" data helps the leakage analyzer identify potential leakage by searching the log for those values. These secret values are generated as a function of the address where they will be stored. If the leakage analyzer finds one of the secret values in the execution log, it can identify the memory location where the leaked data originated. Using the execution model the analyzer can then identify which instruction was the potential source of the leakage and flag it for further analysis.

```
// Setup Gadget
  //S3 (Populate kernel page with secrets)
  memset(KernelPage_X, Secret_X, 4096);

// Helper Gadget
  //H2 (Choose a random address)
  kernel_addr = random(KernelPage_X, Kernel_Page_X + 4096)

  //H5 (Prefetch secret data into L1D$/TLB)
  x = y / z  // dependent divide insts to delay branch resolution
  z = y / x
  y = y / z
  if (y < N) // mispredicted branch on 'y' value
      load(kernel_addr)

  //H10 (Wait for the data to arrive in L1D$)
  NOP * 8

  //H7 (Mispredicted branch to hide exception.)
  x = x / z
  if (x < N)
      //Main Gadget (M1)
      load(kernel_addr) // load supervisor secret
```

Listing 1: Meltdown-US gadgets.

C. Execution Model

The IntroSpectre fuzzer includes an execution model designed to assist gadget selection, instruction generation and leakage analysis. One of the key design challenges for our framework was how to prune the very large space of possible test code that the fuzzer can generate. To this end, we designed a novel approach to guide the fuzzing code generation that relies on an execution model to predict the behavior of the fuzzed code. The execution model guides the intelligent selection of gadgets and gadget parameters to make the code generated for each fuzzing round more likely to exercise useful paths.

The execution model provides an estimate of the microarchitectural state of the processor gradually constructed by the fuzzer as new gadgets are selected for a given fuzzing round. For each instruction or group of instructions that the fuzzer adds to a fuzzing round, the execution model records the expected effects on the microarchitectural states of the processor.

The execution model works similarly to a simple microarchitectural simulator. As Figure 1 shows, the model keeps track of the state of multiple architectural and microarchitectural elements, including mapped pages, register allocation, cache and TLB contents, etc. For each instruction added to the test code, the execution model is updated by estimating the effects of that instruction. For example if the fuzzer adds a

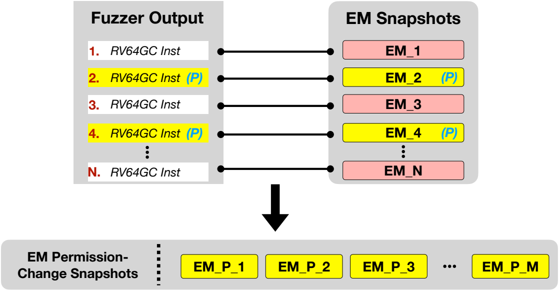


Fig. 2: Execution model snapshots record expected microarchitectural state after each instruction.

Load instruction, its address is also added to a list of cached addresses, along with all other addresses residing in the same cache line. The contents of the TLB and Line Fill Buffer are also updated.

This information is used by the fuzzer to determine if certain requirements for the main gadgets are met. Based on this analysis the fuzzer may choose to add setup or helper gadgets to the code. Let's consider a scenario in which a main gadget M relies on certain values to be in the cache in order to perform as designed. After adding M to the test code, the fuzzer checks the execution model to determine whether the targeted address is on the list of cached addresses. If it is not, the fuzzer chooses a helper gadget that will attempt to prefetch the needed data into the cache. This feedback helps the fuzzer generate test code that is more likely to reveal interesting leakage cases with fewer fuzzing rounds.

1) Interface with the Leakage Analyzer: The Execution Model is used by the Leakage Analyzer to assist with identifying secret leakage. For example, the execution model captures the secret values that need to be looked up in the RTL execution log. If a secret value is found, the model assists with tracing that value back to the source instruction. The execution model state for a fuzzing round is stored as a series of snapshots of the microarchitectural state of the system after each instruction, **EM_1 - EM_N** in Figure 2.

The more relevant snapshots are tagged with special labels for easier identification. For instance the execution model uses permission change labels *(P)* to tag instructions that follow permission changes to user pages. The Leakage Analyzer generates permission change snapshots (**EM_P_1 - EM_P_M**) to determine the sections of the RTL execution log during which illegal accesses may cross isolation boundaries and the log should be monitored for secret leakage.

D. Test Code Generation

The INTROSPECTRE fuzzer generates randomized test code sequences that can be executed on an RTL representation of a processor. The code generation process is illustrated in Figure 3. In the first step, the fuzzer randomly chooses one of the main gadgets ($M3$). The main gadgets include a specification of the microarchitectural state expected by the gadget, if any. An execution model is generated for gadget $M3$ and the requirements of the gadget are checked against it. Note that the gadget's requirements could be satisfied by the execution of other earlier gadgets in the test code. If the check fails, the fuzzer chooses helper/setup gadgets designed to satisfy the missing requirements ($H2$ and $H5$). The execution model is augmented to include the new gadgets. The process is repeated N times and a new randomly chosen main gadget is added to the code sequence in each iteration. The idea behind including multiple main gadgets in the same code sequence in random order is to increase the entropy of the fuzzing code by creating complex interactions between gadgets. The value of N controls the number main gadgets and therefore the complexity of each fuzzing round.

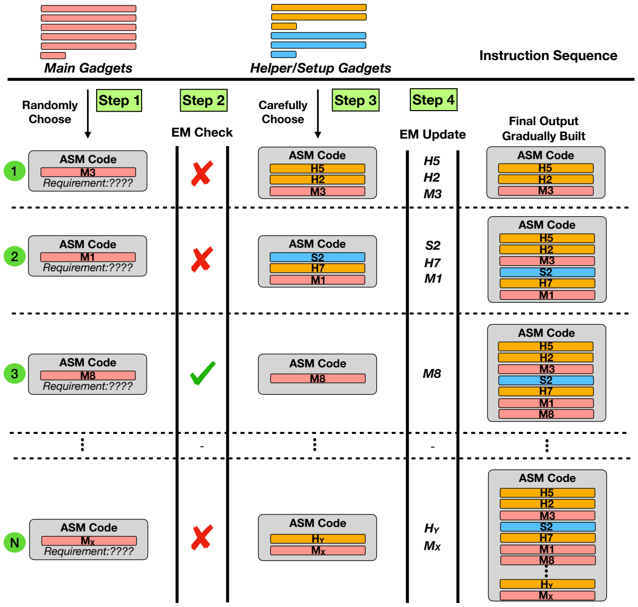


Fig. 3: Generating a fuzzing round from multiple gadgets.

VI. THE LEAKAGE ANALYZER

Each fuzzing round is executed on an RTL simulator augmented to generate a detailed, cycle-level execution log that includes the state of all architectural and microarchitectural storage structures. A Leakage Analyzer module is used to parse the RTL log to identify potential leakage. In addition to the RTL log, the Leakage Analyzer also uses the execution model corresponding to the same fuzzing round. The analysis process consists of three main steps. The first step is to analyze the list of EM snapshots to establish timelines during which values can be considered secrets. This helps exclude legal accesses as well as priming code in which secret values are being set up. The second step is to parse the RTL simulation log to extract all the cycles in which the processor ran in user mode. Finally, the relevant sections of the execution log are searched for all secret values specified in the execution model.

An **Investigator** module parses the execution model to identify secret values that were generated by the fuzzer, as well as establish liveness timelines during which the presence of these values in the RTL log can be considered potential

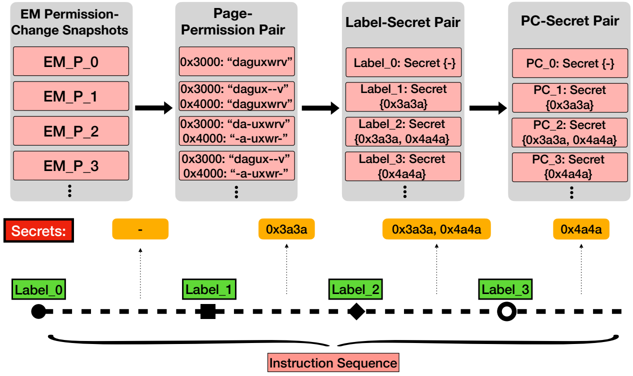


Fig. 4: The Investigator module identifies secrets and their liveness in each fuzzing round.

leakage. The Investigator examines the mapped pages dictionary in each permission-change snapshot. This dictionary records the addresses of mapped user pages along with their permission bits. The investigator extracts these mappings from each EM snapshot (as shown in Figure 4) and creates a key-value pair for each EM permission-change snapshot. The key is the permission-change label used by the fuzzer to track a permission change event in the log. The value is the list of secrets that will be "live" in the next section of the log.

The list of secrets associated with each permission change label is inferred from the permission bits of mapped user pages, and the secret values stored in those pages. For example, in Figure 4 we show that after `Label_1`, the user permissions of page `0x3000` changes from **`xwrv`** to **`x--v`** which means the user loses read/write permissions. The Investigator adds all secret data values (**`0x3a3a`**) stored in page `0x3000` to the secret list for the section of the log spanning from `Label_1` to `Label_2`. A permissions change to page `0x4000` similarly adds a secret from that page (**`0x4a4a`**) after Label_2.

Finally, the Investigator maps the labels to program counter (PC) values in the test sequence binary, to generate (PC-Secret) pairs. Note that the secret timeline is only needed for values residing in user pages. Supervisor/machine memory values are considered secrets throughout the entire fuzzing round execution, while in user mode.

A **Parser** module processes the raw RTL simulation log, which includes the content of all the microarchitectural structures at cycle granularity. The Parser generates two separate files, as shown in Figure 5: The Filtered Execution Log is simply a pruned version of the main RTL log which excludes the machine/supervisor mode execution. The Instruction Log is a timing record for each dynamic instruction executed in the fuzzing round. It includes the cycle number in which each instruction is fetched, decoded, issued, etc. This information helps the Leakage Analyzer track instructions responsible for potential leakage.

A **Scanner** module uses the annotated execution model to search the Filtered Execution Log for leakage, as illustrated in Figure 6. The list of (PC-Secret) pairs is used to identify the sections of the execution log to scan for each secret value. For each user-space secret value, only the sections of the log executed while the value was secret are scanned. For supervisor/machine values the entire log is searched since those values should always be inaccessible from user space. If a secret value is found in a microarchitectural structure, the Leakage Analyzer traces that value back to the producing instruction and flags it as potential leakage. The list of potential leakage sources and the microarchitectural units where leakage was observed are included in the INTROSPECTRE report.

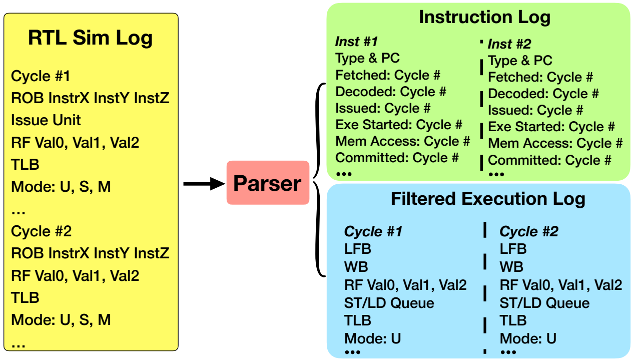


Fig. 5: The Parser module processes the main RTL log in preparation for the leakage analysis.

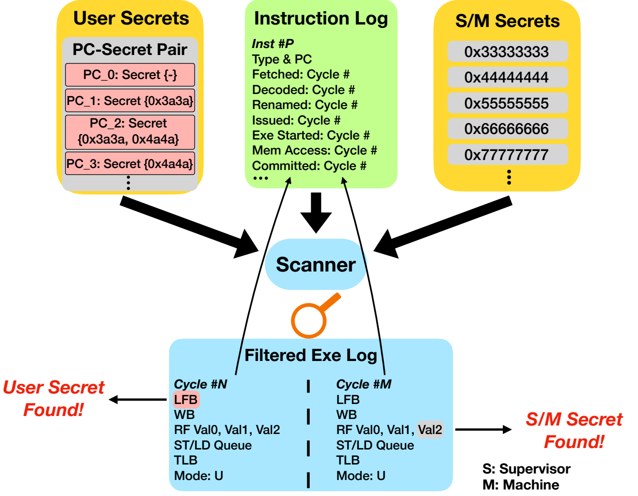


Fig. 6: The Scanner module searches the RTL log for secrets identified and tagged by the Investigator.

VII. IMPLEMENTATION AND EVALUATION METHODOLOGY

We evaluate INTROSPECTRE on a system-on-chip design generated through the Chipyard framework [4]. Chipyard includes Chisel, a high-level hardware generation language, as well as a compiler and toolchain to translate Chisel designs into the FIRRTL intermediate representation, which is then elaborated into Verilog. Particularly of interest to INTROSPECTRE is the `printf` synthesis feature in Chisel. Components defined at higher levels of abstraction can have their state logged, where these statements are automatically carried along through elaboration. This allows for microarchitectural state tracing at cycle granularity.

We implement IntroSpectre on top of Verilator, an open-source RTL simulator which converts Verilog code to a cycle-accurate behavioral model in C++ or SystemC. We use the RISC-V BOOM (Berkeley Out of Order Machine) (v2.2.3) as a target for transient leakage analysis. Detailed configuration parameters for the BOOM SoC are included in Table II.

Core Configuration	Parameter Value
# Core	1
Fetch/Decode Width	4/1
# ROB Entries	32
# Int Physical Regs	52
# FP Physical Regs	48
# LDq/STq Entries	8
Max Branch Count	4
# Fetch Buffer Entries	8
Branch Predictor	Gshare(HisLen=11, numSets=2048)
L1 Data Cache	nSets=64, nWays=4, nMHSR=4, nTLBEntries=8
L1 Inst. Cache	nSets=64, nWays=4, nMHSR=4, fetchBytes=2*4
Prefetching	Enabled: Next Line Prefetcher

TABLE II: BOOM core configuration parameters.

Test cases generated by IntroSpectre build on existing infrastructure in Chipyard for unit test creation, used in verifying functional correctness of individual assembly instructions. This infrastructure, `riscv-tests`, provides a testing environment enabling system support through a minimalist operating system kernel tasked with bootstrapping the processor, setting up virtual memory and exception handlers. This efficient testing environment makes running many test rounds practical with execution times on the order of minutes. All tests are run on a machine equipped with an Intel Xeon E5-2440 2.40GHz CPU, 32GB of RAM, running RHEL 7.9. As shown in Figure 1, IntroSpectre consists of three phases: 1) Gadget Fuzzer (Instruction sequence generation, EM snapshots, binary compilation), 2) RTL Simulation (Verilator simulator) and 3) Analyzer (Investigator, Parser and Scanner). Table III shows the average wall-clock time for each phase, as well as total time for an average fuzzing round.

IntroSpectre Module	Execution Time
Gadget Fuzzer	3.71s
RTL Simulation	206.53s
Analyzer	31.57s
Total	241.81s

TABLE III: Average wall-clock execution time for one fuzzing round.

The majority of test cases consider a threat model with user/supervisor privilege escalation/de-escalation with respect to data accesses. However, we also consider cases of machine privilege escalation, specifically, violating the security guarantees of the Keystone [24] trusted execution environment.

VIII. Leakage Case Studies

In this section, we discuss empirical findings of IntroSpectre which demonstrate its efficacy in discovering potential information disclosure. We categorize results according to the nature of the potential leakage source, or *leakage scenario*, and present various case studies which surface these leakage scenarios. We describe each case study in terms of the isolation boundary bypassed, conditions required to trigger the targeted speculation primitive and helper gadgets incorporated to satisfy said requirements.

Table IV lists all the transient leakage cases and other isolation boundary violations identified by IntroSpectre. We categorize our findings across three classes: **R-Type**: secret values in both physical register file (PRF) and line fill buffer (LFB), **L-Type**: secret values in LFB only, and **X-Type**: miscellaneous, control-flow oriented.

A. R-Type Leakage Case Studies

R-type leakage scenarios include fuzzing rounds where sensitive data values can be found in both the physical register file and line-fill buffer entries. We detail test cases assembled with IntroSpectre exhibiting these characteristics.

*1) **R1**, Supervisor-only Bypass:* The *R1* case study triggers lazy handling of a faulty load to bring supervisor data into the register file and/or LFB, allowing user-mode instructions to access supervisor-owned data. *R1* was found with IntroSpectre by combining the $M1$ main gadget and fulfilling its requirements with the assistance of the $S3$ setup gadget and the $H5$ helper gadget, which fill supervisor pages with secrets and prefetch the secret data into the L1 data cache, respectively. The behavior of *R1* is reminiscent of the original Meltdown exploit.

*2) **R2**, User-only Bypass:* The *R2* case study is similar to *R1*, but the isolation boundary crossed is instead supervisor-to-user. Normally, accessing a memory location mapped to a lower privilege while running in a higher privilege mode is not strictly illegal. However, by clearing the access bits in the `sstatus` CSR (RISC-V configuration register) with the $S2$ setup gadget, the supervisor's access to user pages is no longer permitted. Here again, similar to *R1*, despite raising a page fault exception, the memory access is performed and data is brought into the RF or LFB. $H11$ and $H5$ helper gadgets are used to fill user pages with secrets and bring the user data into the cache. The $M2$ main gadget then accesses user mapped memory while in supervisor mode.

*3) **R3**, Machine-only Bypass:* Describing the *R3* case study necessitates an explanation of the physical memory protection mechanisms in RISC-V and the Keystone trusted execution environment. Keystone is an open-source framework for designing trusted execution environments (TEE) with secure hardware enclaves on RISC-V systems. A Keystone system has a security monitor (SM) at its core which is trusted software that runs in the highest execution privilege in RISC-V (M), and forms the trusted computing base (TCB) in a Keystone system. The security monitor is implemented on top of Berkeley Bootloader (BBL) and is responsible for isolating memory regions using RISC-V physical memory protection. The security monitor is also in charge of managing secure hardware enclaves and remote attestation.

	Secret Leakage Instances in the BOOM processor	**Gadget Combination - Guided Fuzzing**
R1	Supervisor-only bypass	S3, H2, H5$_{6}$, H10$_{3}$, H7$_{4}$, H8$_{3}$, $\mathbf{M1_{2}}$
R2	User-only bypass	H1, H4$_{8}$, H11$_{4}$, S2, H1, H5$_{3}$, H10$_{1}$, H7$_{2}$, H8$_{1}$, $\mathbf{M2_{5}}$
R3	Machine-only bypass	S4, H3, H5$_{7}$, H10$_{2}$, H7$_{1}$, H8$_{2}$, $\mathbf{M13_{8}}$
R4	Reading from invalid user pages regardless of permission bits	H1, H4$_{5}$, H11$_{1}$, H9, S1, $\mathbf{M6_{0\text{-}64}}$, $\mathbf{M10_{10}}$, H7$_{3}$, H8$_{2}$, $\mathbf{M5_{128\text{-}192}}$, H6$_{1}$, H7, $\mathbf{M10_{4}}$, M3$_{15}$, H10$_{3}$, M8, H1 , M11$_{3}$
R5	Reading from user pages without read permission	H1, H4$_{3}$, H11$_{8}$, H9, S1, $\mathbf{M6_{192\text{-}256}}$, H1, H4$_{3}$, $\mathbf{M5_{64\text{-}96}}$, H11$_{5}$, M4$_{2}$, M9$_{4}$, H7$_{1}$, $\mathbf{M10_{10}}$, H1, H4$_{4}$, H11$_{3}$, S1, $\mathbf{M6_{160\text{-}192}}$, M9$_{3}$, M5$_{192\text{-}224}$, H7$_{6}$, $\mathbf{M10_{8}}$
R6	Reading from user pages with access and dirty bits off	H1, H4$_{1}$, H11$_{3}$, H5$_{4}$, H10$_{2}$, M5$_{32\text{-}64}$, H9, S1, $\mathbf{M6_{32\text{-}96}}$, H7$_{3}$, $\mathbf{M10_{5}}$, $\mathbf{M5_{0\text{-}16}}$
R7	Reading from user pages with access bit off	H1, H4$_{2}$, H11$_{6}$, S1, $\mathbf{M6_{80\text{-}160}}$, M9$_{3}$, M5$_{32\text{-}96}$, H7$_{4}$, H8$_{2}$, M10$_{1}$, H1, S1, $\mathbf{M6_{64\text{-}256}}$, H7$_{3}$, $\mathbf{M5_{16\text{-}64}}$, $\mathbf{M10_{8}}$
R8	Reading from user pages with dirty bit off	H1, H4$_{4}$, H11$_{1}$, S1, $\mathbf{M6_{32\text{-}48}}$, H5$_{2}$, H7$_{1}$, $\mathbf{M10_{2}}$, $\mathbf{M10_{9}}$
L1	Leaking page table entries through LFB	H1, $\mathbf{H4_{6}}$, H11$_{4}$, M8, H9, S1, M6$_{128\text{-}160}$, H5$_{7}$, H7$_{4}$, $\mathbf{M10_{3}}$, H1, H4$_{8}$, M4$_{6}$, H9, S1, M6$_{64\text{-}160}$, H5$_{3}$, $\mathbf{M12_{0\text{-}48}}$
L2	Leaking secrets of a page without proper permissions in LFB by using prefetcher	H1, H4$_{3}$, H11$_{2}$, M4$_{1}$, H1, H6$_{2}$, H7$_{3}$, M3$_{7}$, M11$_{12}$, H9, S1, $\mathbf{M6_{192\text{-}256}}$, H7$_{1}$, $\mathbf{M5_{16\text{-}64}}$, H1, H5$_{8}$, H10$_{2}$, M12$_{32\text{-}64}$, H1, M4$_{3}$, H5$_{2}$, H7$_{3}$, $\mathbf{M10_{4}}$, H5$_{2}$, M11$_{5}$
L3	Leaking supervisor secrets after handling an exception through LFB	**S3**, H2, M9$_{5}$, H1, H4$_{6}$, M5$_{64\text{-}128}$, **H9**, S1, S3, M6$_{64\text{-}128}$, H7$_{6}$,H8$_{2}$, M4$_{1}$, $\mathbf{M12_{0\text{-}16}}$, H1, H11$_{3}$, M5$_{0\text{-}32}$
X1	Jump to an address and execute the stale value	H1, H4$_{3}$, H11$_{2}$, H7$_{1}$, M9$_{7}$, H1, H4$_{4}$, H7$_{3}$, H6$_{2}$, H10$_{4}$, $\mathbf{M3_{13}}$
X2	Speculatively execute supervisor-code/inaccessible-user-code while in user mode	S3, H2, H5$_{6}$, H7$_{2}$, H6$_{1}$, M3$_{1}$, H1, H4$_{3}$, H11$_{7}$, H7$_{1}$, H6$_{2}$, H10$_{3}$, $\mathbf{M15_{2}}$, M9$_{7}$, $\mathbf{M14_{1}}$
	Secret Leakage Instances in the BOOM processor	**Gadget Combination - Unguided Fuzzing**
Rnd1	Supervisor-only bypass (Secret only in LFB)	H1, H2, M9$_{5}$, S3, M11$_{3}$, M10$_{4}$, H7$_{3}$, $\mathbf{M1_{7}}$, M4$_{1}$, S1
Rnd2	Supervisor-only bypass (Secret only in LFB)	H6, M10$_{4}$, H10$_{4}$, S3, H2, M12$_{16\text{-}32}$, H9, M11$_{6}$**,** $\mathbf{H5_{3}}$, H10$_{2}$
Rnd3	Supervisor-only bypass (Secret only in LFB)	M15$_{1}$H2, H11$_{5}$, M8, M3$_{2}$, M2$_{6}$, S3, $\mathbf{M10_{8}}$, M5$_{224\text{-}256}$, H4$_{8}$

TABLE IV: Secret leakage scenarios and the gadget combinations that triggered them. The main gadget responsible for the leakage is highlighted in bold. The subscript number for each gadget represents the permutation ID.

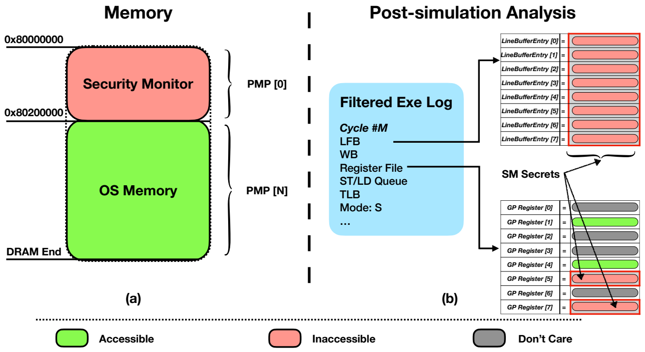


Fig. 7: a) Memory layout of security monitor-enabled bootloader. b) SM secrets show up in PRF and LFB in post-simulation analysis.

According to the RISC-V Privileged ISA [50], the RISC-V physical memory protection (PMP) unit provides a set of CSRs to specify physical memory access privileges (read, write, execute) for each physical memory region. Using PMP control and status registers, the security monitor divides the memory into two parts at boot time, as illustrated in Figure 7. The first PMP entry CSR configured by the security monitor sets its own address range with all permissions turned off (all bits set to 0). The last PMP entry CSR is configured to grant full permissions for the remainder of the memory space (bits set to 1). This setup allows the OS access to all of memory, with the exception of memory belonging to the security monitor.

The test sequence responsible for leakage scenario *R3* passes fuzzer selected instructions as the payload to the bootloader, which is enhanced with a security monitor, to be executed in supervisor mode. Enclave creation is not part of the test sequence because the goal is to bypass machine-only memory, where the entire security monitor address range already meets this requirement.

INTROSPECTRE primes the machine-only memory by storing different secret values throughout the security monitor address range using the $S4$ setup gadget. According to Keystone security assumptions, this region should not be accessible by processes in user or supervisor mode. However, INTROSPECTRE was able to produce test cases showing machine-mode secrets in the LFB, PRF and write-back buffer. Analyzing the test sequence, we determined this was accomplished by the $M13$ main gadget executing in supervisor mode. $M13$ accessed security monitor memory (protected by RISC-V PMP), which raised a Load Access Fault exception. However, the memory request was not squashed, and the secret value was eventually accessed–finding its way through to the LFB (if not cached) or PRF (if cached by the $H5$ helper gadget).

4) ***R4-R8****, Leaking Inaccessible Data:* Case studies *R4-R8* include scenarios where user pages are accessed by user-mode processes, but without proper access permissions. INTROSPECTRE first fills user pages with secret values using the $H11$ helper gadget and then changes the permission bits of those pages with the $M6$ main gadget. The permissions change requires higher privilege execution and the $S1$ setup gadget is used to accomplish this. Finally, main gadgets such as $M10$, $M12$, and $M5$ are used to perform various loads to different pages with different permission bits, in an effort to increase the odds of discovering potential leakage. The cause of leakage in this scenario is similar to *R1*, where the memory request is not canceled despite raising an exception.

As a concrete example, in the *R4* case study, secrets reside in an invalid user page and can be leaked through the LFB and PRF. Normally, when the valid bit of a page is clear the page should not be accessible to any privilege level, regardless of other permission bits in the PTE. However, we have observed that contents of an invalid user page is leaked to the LFB and register file even though the access instruction raises a page fault exception.

The same behavior can be found in case studies *R8*, *R7*, *R6* and *R5* where user pages have dirty bits cleared, access bits cleared, both access and dirty bits cleared, or lack read permissions, respectively. Accessing pages with the criteria outlined for case studies *R4-R8* raises a page fault exception, but the data is retrieved before the offending instructions are squashed. We have used the *FuzzPermissionBits* $M6$ main gadget to cover all possible combinations of user page permission bits.

B. L-Type Leakage Case Studies

L-type leakage scenarios include fuzzing rounds where secret values can be observed in line fill buffer entries but did not make their way through the register file. Note that the line fill buffer interfaces between caches, and, while not directly accessible by the user, it has been shown to be vulnerable to side channel leakage [38].

1) ***L1****, Leaking PTE through LFB:* In the *L1* case study, IntroSpectre captured page table entries in the LFB while running in user mode. Page table entries are part of supervisor memory, which should be protected from user mode instructions. On every TLB miss, which may be added to the test sequence with the $H4$ helper gadget, an internal cache request to the address of the root page table is performed to retrieve the correct mapping (if available). If this cache request misses in the L1D, the LFB gets filled with the entire line of PTEs in the next few cycles. The prefetcher can also exacerbate this situation by prefetching the next line and bringing it to the LFB, exposing additional PTEs.

2) ***L2****, Leaking from Prefetcher to LFB:* The *L2* case study reveals that hardware prefetchers can introduce potential sources of leakage. Normally, accessing page boundary-straddling addresses of a user page with proper permissions is legal. However, if inaccessible user pages (created by fuzzing page permission bits with main gadget *M6*) are located right after an accessible user page, we may have secrets of the inaccessible page leaked into the LFB. This can be triggered by accessing the boundary-straddling addresses of the accessible pages via torturous loads from the *M10* main gadget. As a simple example shown in Figure 8, if an access to address `0x5FF8` misses in the L1D, the prefetcher will request the next address, which falls within the boundary of the next (inaccessible) page. As a result, secret values from page `0x6000` will be brought into the LFB.

3) ***L3****, Exception Handler Leakage:* Case study *L3* demonstrates possible leakage on exception handler exit. When an exception is raised while executing in user mode (e.g executing *H9, H4* helper gadgets), execution privilege is changed to supervisor mode to handle this exception. Before executing the exception handler code, the processor stores the values of general purpose registers in supervisor memory (Trap Entry in Figure 9). After the handler has finished execution and before changing the execution privilege back to user mode, the processor reloads the previously stored register values from supervisor memory back to the registers (Pop Trap Frame in Figure 9). IntroSpectre produced test cases in which some of these loads from supervisor memory miss in the cache. As a result, the LFB gets filled with supervisor data residing in memory locations that fall in the same cache line as the stored register values. Figure 10 illustrates such a scenario. While entries *LFB[0–5]* contain the saved registers, *LFB[6]* and *LFB[7]* contain supervisor data. In other words, this secret data is brought into the LFB and remains there even after the execution privilege is reverted back to user mode. This effect is again amplified by the prefetcher, which fetches the next line into the LFB bringing supervisor data to *LFB[8–15]*.

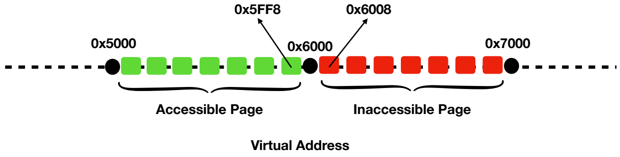
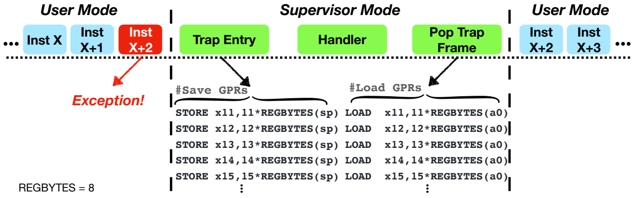


Fig. 8: Illustration of accesses straddling two memory pages with different permissions.



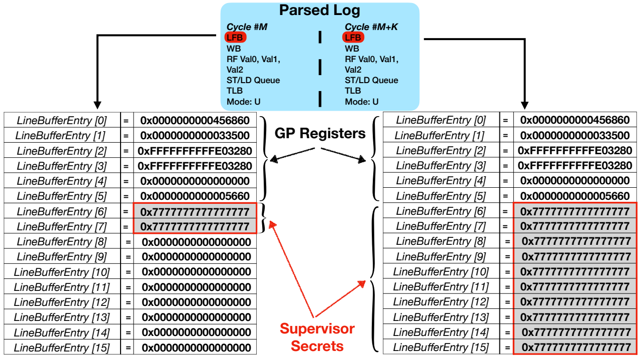
```
#Save GPRs                       #Load GPRs
STORE x11,11*REGBYTES(sp) LOAD  x11,11*REGBYTES(a0)
STORE x12,12*REGBYTES(sp) LOAD  x12,12*REGBYTES(a0)
STORE x13,13*REGBYTES(sp) LOAD  x13,13*REGBYTES(a0)
STORE x14,14*REGBYTES(sp) LOAD  x14,14*REGBYTES(a0)
STORE x15,15*REGBYTES(sp) LOAD  x15,15*REGBYTES(a0)
          ⋮                           ⋮
```

Fig. 9: Instructions executed to handle an exception.

C. X-Type Leakage Case Studies

X-type leakage scenarios exhibit speculative control-flow hijacking and the influence of indirect branch targets.

1) ***X1****, Execute Stale PC:* In this case study, IntroSpectre demonstrates that control logic can proceed with a stale PC value, if there is an outstanding store request to the same address as the PC. This behavior was revealed by the $M3$



LineBufferEntry [0]	=	0x0000000000456860
LineBufferEntry [1]	=	0x0000000000033500
LineBufferEntry [2]	=	0xFFFFFFFFFFE03280
LineBufferEntry [3]	=	0xFFFFFFFFFFE03280
LineBufferEntry [4]	=	0x0000000000000000
LineBufferEntry [5]	=	0x0000000000005660
LineBufferEntry [6]	=	0x7777777777777777
LineBufferEntry [7]	=	0x7777777777777777
LineBufferEntry [8]	=	0x0000000000000000
LineBufferEntry [9]	=	0x0000000000000000
LineBufferEntry [10]	=	0x0000000000000000
LineBufferEntry [11]	=	0x0000000000000000
LineBufferEntry [12]	=	0x0000000000000000
LineBufferEntry [13]	=	0x0000000000000000
LineBufferEntry [14]	=	0x0000000000000000
LineBufferEntry [15]	=	0x0000000000000000

LineBufferEntry [0]	=	0x0000000000456860
LineBufferEntry [1]	=	0x0000000000033500
LineBufferEntry [2]	=	0xFFFFFFFFFFE03280
LineBufferEntry [3]	=	0xFFFFFFFFFFE03280
LineBufferEntry [4]	=	0x0000000000000000
LineBufferEntry [5]	=	0x0000000000005660
LineBufferEntry [6]	=	0x7777777777777777
LineBufferEntry [7]	=	0x7777777777777777
LineBufferEntry [8]	=	0x7777777777777777
LineBufferEntry [9]	=	0x7777777777777777
LineBufferEntry [10]	=	0x7777777777777777
LineBufferEntry [11]	=	0x7777777777777777
LineBufferEntry [12]	=	0x7777777777777777
LineBufferEntry [13]	=	0x7777777777777777
LineBufferEntry [14]	=	0x7777777777777777
LineBufferEntry [15]	=	0x7777777777777777

Fig. 10: In the **L3**, Exception Handler Leakage, supervisor secrets end up in the LFB in cycle M. In cycle M+K, the prefetcher brings an entire cache line of supervisor data into the LFB.

Isolation Boundaries	Main Gadgets															Leakage Type Identified
	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	
U ——> S	✓		-	✓	✓		✓	✓	✓	✓	✓	✓		✓		R1, L1, L3
S ——> U		✓	-							✓	✓				✓	R2
U ——> U*			-	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	R4-R8, L2
U/S ——> M			-	✓			✓	✓		✓		✓	✓			R3

TABLE V: Coverage of leakage across multiple isolation boundaries - (U)ser, (S)uperviser, (M)achine - types of leakage found, and main gadgets used in the fuzzing rounds that revealed the leakage. Arrows represent the execution privilege of main gadget (left) and privilege level of accessed memory (right).

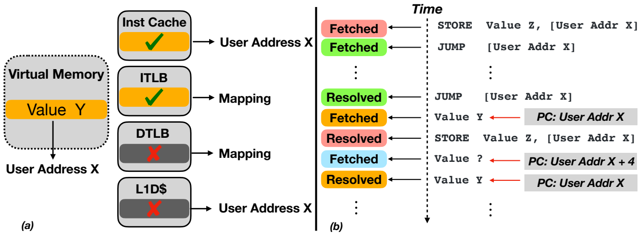


Fig. 11: a) Required setup for Meltdown-JP. b) Timeline of instruction execution.

main gadget. Figure 11 shows a "User Address X" located in a user page with full permissions. This address is primed with the assistance of the $H6$ helper gadget, but it is not present in the cache. Also, the virtual-to-physical mapping for this user page is available in the ITLB but not in the DTLB. With this setup, a jump instruction to "User Address X" that immediately follows a store to the same address, would resolve faster than the store. As a result, the PC is loaded with the stale "Value Y" and the control-flow of the program is changed. Importantly, the addresses of the jump and store are not disambiguated, so no conflict is detected and no exception will be raised.

*2) **X2**, Illegal Speculative Control-Flow:* The *X2* case study executes supervisor code speculatively from user mode. The requirement for this test is to have the supervisor address in the instruction cache. By performing a jump to this S-mode address, the processor assigns an ROB entry for the instruction and raises an instruction page fault exception as soon as the instruction is added to the ROB. Although the instruction never executes and subsequent instructions are also not assigned resources, it may still leak information about the instruction type in supervisor code.

D. Guided and Unguided Fuzzing

In order to demonstrate the importance of guided fuzzing, we also examined INTROSPECTRE with the Execution Model removed. In this experiment we generate 100 fuzzing rounds in which gadgets are randomly chosen from our pool of gadgets, with randomly assigned configuration parameters. Each fuzzing round includes 10 gadgets. We find that, out the 100 rounds, 3 rounds reveal the "Supervisor-only bypass" leakage instance (*Rnd1–Rnd3* in Table IV), however the leakage is only observed in the line fill buffer, with the secret value not reaching the register file. Overall, the random pick of gadgets is much less effective at identifying new leakage, with 1 leakage type out of 100 runs, compared to the 13 distinct leakage scenarios for roughly the same number of fuzzing rounds, with the INTROSPECTRE guided process.

E. Coverage Analysis and Discussion

We examine the coverage of INTROSPECTRE along four dimensions:

1) Coverage of Microarchitectural Structures: INTROSPECTRE can track all microarchitecturally accessible storage elements. In our implementation we record all structures that could possibly be a leakage source. From this perspective, INTROSPECTRE can guarantee full coverage of all microarchitectural storage elements.

2) Coverage of Isolation Boundaries: We enumerate all possible combinations of isolation boundaries that could be vulnerable to Meltdown-type attacks. Table V shows all possible accesses across isolation boundaries. We show the main gadgets that exercise those accesses and the instances in which possible leakage was identified. We can see that the INTROSPECTRE gadgets cover accesses across all possible isolation boundaries.

3) Gadgets Coverage: Gadgets implement kernels from known attacks, speculation primitives, memory access across isolation boundaries, etc. As a proof-of-concept, we implement a subset of the known Meltdown-like attacks that are relevant to the BOOM architecture. This set can be expanded to more attacks, other speculation primitives, etc. The set of gadgets cannot be guaranteed to be complete, as there are simply too many degrees of freedom.

To increase the probability of discovering new attacks or new variants, most gadgets are parametrized, adding another dimension to the fuzzing space. In most cases these parameters can be enumerated to ensure complete coverage of each parameter value. The permutation metric in Table I shows the number of variants for each gadget class. Examples of gadget parameters include speculation primitives, access instructions, secret layouts, access permissions, etc. For example, the *STtoLD-Forwarding* ($M5$ in Table I) main gadget has 256 variants (permutations), as illustrated in Figure 12. For M5 we choose between four types of Load instructions, four types of Store instructions, four memory access granularities, and residency state in the L1D and line fill buffer (LFB).

In summary INTROSPECTRE looks for leakage in all relevant storage elements, supports accesses across all isolation boundaries and utilizes gadgets targeting all known Meltdown-like attacks, enhanced with hyperparameters to increase coverage to new variants.

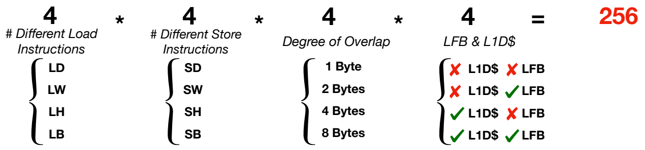


Fig. 12: Possible permutations for the M5 *SToLD-Forwarding* gadget.

F. False Positives and False Negatives

False negatives can be examined along two dimensions: (1) Will IntroSpectre always flag secret leakage revealed by the fuzzer? The answer is yes. There are *no false negatives, if a secret leakage is triggered by the fuzzer*; and (2) will IntroSpectre discover all known and unknown Meltdown-like attacks? In terms of known Meltdown-like attacks, all relevant and applicable known attacks are covered. Also, hyperparameters increase IntroSpectre coverage of potential new variants. However, IntroSpectre cannot guarantee all Meltdown attacks will be discovered.

False positives can similarly be examined along the following dimensions: (1) If IntroSpectre flags leakage in a structure, is it guaranteed to be a violation of an isolation boundary? The answer is yes. *There are no false positives for isolation boundary violations*. (2) Are all IntroSpectre-identified leakage cases exploitable? The answer is no, because this depends on whether a covert channel can be open to leak that secret. Additional expert analysis will be required to ascertain if the leakage is exploitable. IntroSpectre will therefore have *false positives for exploitable attacks*.

IX. Conclusion

This paper presented IntroSpectre, a design verification framework for identifying transient execution leakage. We show that integrating IntroSpectre into the RTL design flow enables a systematic analysis of the entire microarchitectural state of complex processor designs. We integrated IntroSpectre with an RTL simulator and used it to perform transient leakage analysis on the open-source RISC-V BOOM processor. We identified 13 potential transient leakage scenarios, most of which had not been highlighted on this processor design before.

Acknowledgment

The authors would like to thank Matthew Fernandez and Yuan Xiao from Intel for advice and help with this work. We would also like to thank the anonymous ISCA reviewers for constructive and detailed feedback which helped us improve this paper.

References

[1] "Verilator SystemC Environment," Tech. Rep., https://veripool.org/papers/verilator_systemperl_nascug.pdf, [Online; accessed 6-May-2021].

[2] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 132–144.

[3] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 870–887.

[4] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[5] ARM, "Vulnerability of speculative processors to cache timing side-channel mechanism," 2018, https://developer.arm.com/support/security-update, [Online; accessed 6-May-2021].

[6] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Spec-Shield: Shielding speculative data from microarchitectural covert channels," in *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 151–164.

[7] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[8] J. V. 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 *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[9] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[10] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "BOOM v2: An open-source out-of-order RISC-V core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017.

[11] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, vol. 40, no. 3, 2012, pp. 106–117.

[12] M. R. Fadiheh, J. Muller, 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 *57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[13] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Proceedings Network and Distributed System Security Symposium*, 2020.

[14] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: long live KASLR," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 161–176.

[15] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *IEEE Symposium on Security and Privacy*, 2020, pp. 160–178.

[16] F. Haedicke, H. M. Le, D. Grosse, and R. Drechsler, "Crave: An advanced constrained random verification environment for SystemC," in *International Symposium on System on Chip (SoC)*, 2012, pp. 1–7.

[17] J. Horn, "Speculative execution, variant 4: Speculative store bypass," 2018, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, [Online; accessed 6-May-2021].

[18] Intel, "Intel analysis of speculative execution side channels," Intel, 2018, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, [Online; accessed 6-May-2021].

[19] Intel, "Speculative execution side channel mitigations," Intel, 2018, https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, [Online; accessed 6-May-2021].

[20] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[21] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: a defense against cache timing attacks in speculative execution

processors," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 974–987.

[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 *IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[23] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018.

[24] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[25] 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, pp. 973–990.

[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.

[27] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS 2018 : The 25th ACM Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[28] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[29] A. Mehta, *Constrained Random Verification (CRV)*. Springer, 06 2018, pp. 65–74.

[30] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[31] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 176–187.

[32] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface." in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1481–1498.

[33] M. F. Oliveira, C. Kuznik, H. M. Le, D. Große, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen, "The system verification methodology for advanced tlm verification," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2012, pp. 313–322.

[34] H. Ragab, A. Milburn, R. Kaven, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *42nd IEEE Symposium on Security and Privacy (S&P'20)*, 2021.

[35] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "Undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[36] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Sjalander, "Efficient invisible speculative execution through selective delay and value prediction," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 723–735.

[37] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruß, "Context: A generic approach for mitigating Spectre," in *Network and Distributed System Security Symposium*, 2020.

[38] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[39] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network." *arXiv preprint arXiv:1807.10535*, 2018.

[40] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels." *arXiv preprint arXiv:1806.07480*, 2018.

[41] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, vol. 39, no. 11, 2004, pp. 85–96.

[42] 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*, 2019, pp. 395–410.

[43] 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*, vol. 44, no. 3, 2009, pp. 109–120.

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

[45] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software." *arXiv: Hardware Architecture arXiv:2102.02308*, 2021.

[46] P. Turner, "Retpoline: A software construct for preventing branch-target- injection," Google, 2018, https://support.google.com/faqs/answer/7625886, , [Online; accessed 6-May-2021].

[47] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[48] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.

[49] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on Intel CPUs via cache evictions." *arXiv preprint arXiv:2006.13353*, 2020.

[50] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified," RISC-V Foundation, Tech. Rep., June 2019, [accessed 6-May-2021]. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf

[51] 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.

[52] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities," *Network and Distributed System Security Symposium (NDSS)*, 2020.

[53] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[54] 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 *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 707—720.

[55] 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*, vol. 40, no. 3, 2019, pp. 954–968.

[56] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 43, no. 1, 2015, pp. 503–516.

[57] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans." in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[58] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: the promise and the pitfalls," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, p. 2.