SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities

Yuan Xiao
The Ohio State University
xiao.465@osu.edu

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

Radu Teodorescu
The Ohio State University
teodores@cse.ohio-state.edu

***Abstract*—SPEculative Execution side Channel Hardware (SPEECH) Vulnerabilities have enabled the notorious Meltdown, Spectre, and L1 terminal fault (L1TF) attacks. While a number of studies have reported different variants of SPEECH vulnerabilities, they are still not well understood. This is primarily due to the lack of information about microprocessor implementation details that impact the timing and order of various micro-architectural events. Moreover, to date, there is no systematic approach to quantitatively measure SPEECH vulnerabilities on commodity processors.**

This paper introduces SPEECHMINER, a software framework for exploring and measuring SPEECH vulnerabilities in an automated manner. SPEECHMINER empirically establishes the link between a novel two-phase fault handling model and the exploitability and speculation windows of SPEECH vulnerabilities. It enables testing of a comprehensive list of exception-triggering instructions under the same software framework, which leverages covert-channel techniques and differential tests to gain visibility into the micro-architectural state changes. We evaluated SPEECHMINER on 9 different processor types, examined 21 potential vulnerability variants, confirmed various known attacks, and identified several new variants.

I. INTRODUCTION

Speculative Execution Side Channel Hardware Vulnerabilities [27] are computer micro-architectural vulnerabilities in modern pipelined processors that, due to speculative and out-of-order execution, may execute instruction sequences that should not be executed if instructions are strictly executed in program order. Speculatively executed instructions may lead to information leakage as they lead to state changes in cache in the same way as retired instructions. Such vulnerabilities are the root causes of the well-known Meltdown [23], Spectre [17], Foreshadow [34], and RIDL [35].

Although these security attacks are high-profile due to their severe consequences, they are unlikely to be completely eliminated in modern high-performance processors, because transient execution (including speculative and out-of-order execution), implicit caching, and aggressive prefetching offer significant performance gains. While some of these vulnerabilities can be mitigated by microcode patches or hardware fixes [13], [2], [4], others have to be temporarily mitigated by software [27], [21], [3], [25]. Moreover, new variants of these vulnerabilities are constantly being discovered by hackers and security researchers. Prominent examples include LazyFP [32], Meltdown-RW [16], Fallout [29], ZombieLoad [30], *etc.*

A major challenge faced by researchers, software developers and hardware designers is the ignorance about the fundamental question of what determines the success or failure of an attack. Without a concrete general conclusion over the nature of these attacks, great efforts are put into figuring out unique mitigation for each newly-emerging variant. Evaluating variants is also difficult with only random attempts of seemingly relative implementation tricks, hoping for a successful exploitation. Three aspects of complexity lead to the difficulty for a general conclusion to be made. First, the attacks vary greatly from each other. They have different threat models and exploit different instructions. And not enough details are provided about their implementation. Second, the micro-architectural states during execution is unobservable and unpredictable. The aggressive speculative and out-of-order processor workflow leads to great complication for the execution of even one instruction. Third, the design and implementation of computer micro-architectures are highly variable depending on processor generation and manufacturer. The same variant of a vulnerability may manifest on one processor family but not others. Therefore, given a commodity processor, there is no method that could affirmatively assert that a specific processor is free of all known vulnerabilities: The only result that can be demonstrated by security researchers is a successful attack on a particular CPU under a certain condition, but unsuccessful attempts do not offer a sense of security.

The goal of the paper is to (1) comprehensively understand the SPEculative Execution side Channel Hardware (SPEECH)[1] Vulnerabilities [27] in modern computer micro-architectures and (2) systematically and quantitatively evaluate SPEECH vulnerabilities on commodity processors, including providing deterministic evidence for inexploitable variants. As the Spectre-type vulnerabilities [5] are due to illegal poisoning of branch prediction rather than hardware implementation flaws in the prediction units themselves, we put emphasis on the more complex Meltdown-type SPEECH vulnerabilities caused by fault handling. All Meltdown-type vulnerabilities [5] require such faults that may or may not trigger an explicit exception. In this paper, we intend to get a comprehensive understanding of the general Meltdown-type vulnerabilities. Thus, we focus

[1] Speculative Execution Side Channel Hardware Vulnerability is the term preferred by Intel and Microsoft. We omit "S" in the acronym to reflect the debate of whether a side channel or a covert channel is used in such attacks.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23105
www.ndss-symposium.org

on the core x86 ISA but leave hardware extensions such as SGX and VMX to future work.

However, it is very challenging to precisely determine the internal micro-architectural implementation of the processors, which is not made public in sufficient detail by the processor vendors. Moreover, while some of the micro-architectural design choices may be available, implementation details such as timing and order of events are not documented. To achieve the first research goal, we propose a novel two-phase model to describe the execution of x86 instructions with respect to the handling of faults. The model abstracts away complex implementation details and focuses on the software-observable and measurable events that are relevant to SPEECH vulnerabilities. More specifically, the two-phase model describes the exploitation of a SPEECH vulnerability as the outcome of two race conditions: A race condition between data fetching and processor fault handling and a race condition between covert channel transmission and speculative instruction squashing. The *exploitability* and *speculation window* can be determined by these two race conditions.

To achieve the second research goal, we designed a software framework, dubbed SPEECHMINER, to systematically and quantitatively measure the exploitability and the speculation windows of a variety of SPEECH vulnerabilities. However, building such an analytical framework is technically challenging. SPEECHMINER brings forward the following solutions to this non-trivial task:

To gain visibility into the micro-architecture, SPEECHMINER employs covert-channel techniques to indirectly infer micro-architectural state changes. To establish the link between the two-phase model and the SPEECH vulnerabilities, SPEECHMINER incorporated several carefully designed experiments to infer the internal implementations of the tested computer micro-architecture. To enable quantitative analysis, SPEECHMINER dynamically adjusts the tested instruction sequences and utilizes differential tests to quantify the exploitability and speculation windows. Finally, to enable systematic analysis, instead of exhausting all micro-architectural uses of speculative execution, we enumerate all architecturally-observable exceptions by referencing the software development manuals from the vendors (though in a manual way) to instantiate the test cases of SPEECHMINER.

We run SPEECHMINER on 9 different types of processors to examine 21 different variants of SPEECH vulnerabilities. our experiments not only confirmed previously demonstrated attacks, but also identified a few new exploitable variants on the tested Intel and AMD machines, which have been reported to the vendors. Moreover, SPEECHMINER enabled us to perform quantitative measurements of the exploitability and speculation windows of these vulnerabilities. The significance of the quantitative analysis is that it provides security assurance for the negative results—a processor not vulnerable in one of our exploitability tests is assured to be immune from the corresponding attack.

Moreover, our study yields some very interesting discoveries. For instance, it explains why zero values are sometimes returned by the Meltdown-US attacks; it suggests that the speculation window of any faulting instructions can be controlled and tuned by the attacker; it rules out the possibility of advanced attacks by nesting multiple speculative instructions; it explains why Meltdown-US attacks can leak data not present in the L1 cache, but L1TF attacks cannot.

In summary, this work makes the following contributions:

- It proposes a novel two-phase model to describe x86 fault handling and its relationship to the exploitability and speculation windows of SPEECH vulnerabilities.
- It designs and implements SPEECHMINER framework to automatically and systematically explore and measure SPEECH vulnerabilities.
- It enables quantitative measurements of the exploitability of SPEECH vulnerabilities, providing security assurance to negative test results.
- It explains the root causes of some observations made by prior studies and clarifies common misunderstandings.
- It performs automated tests of 21 vulnerability variants on 9 processor types; it confirms existing vulnerabilities and uncovers a few new variants of SPEECH vulnerabilities.

II. MODELING SPEECH VULNERABILITIES

A. Documented Instruction Execution Model

Although the exact internal implementation of a commodity processor is proprietary to each processor vendor, some design details are made available in Intel and AMD software developer manuals, white papers, patent applications, as well as technical blogs written by computer architects and hardware engineers. The execution model we build in this paper abstracts away aspects that are not relevant to SPEECH vulnerabilities. We show an overview of typical out-of-order execution engine in Fig. 1. As the figure shows, instruction execution follows five main stages: instruction fetch, decode, issue, execute (including memory access) and retire.

1) Fetching, Decoding, Execution and Retirement: We first model the five stages for an instruction in the execution engine.

Instruction fetching. The front end of the processor fetches instructions from the L1 instruction cache. Since instruction addresses are virtual, they must first be translated into physical addresses before a fetch request can be sent to memory. Virtual to physical address translations are cached in the instruction TLB (ITLB). If a translation is not present in the ITLB, a look-up request will be sent to the second-level TLB (STLB), paging structure caches, or the page tables in the memory.

An access permission check is performed simultaneously with the address translation. If the check fails (*e.g.* due to an illegal address), the front end will immediately raise an exception and start fetching instructions from the exception handler. The TLB entry may also be marked as invalid. The instruction at the illegal address will not be decoded or issued for execution [14, Chapter 2.3.2]. These instructions therefore cannot serve as the basis for a speculative execution attack. If the permission check succeeds, instructions are fetched from the L1 Instruction cache, lower-level caches, or the memory. They are then passed on to the decode stage.

Instruction decoding. The instruction decoder is responsible for interpreting the instruction, identifying operands and, in the case of x86, translating the complex (CISC) instructions

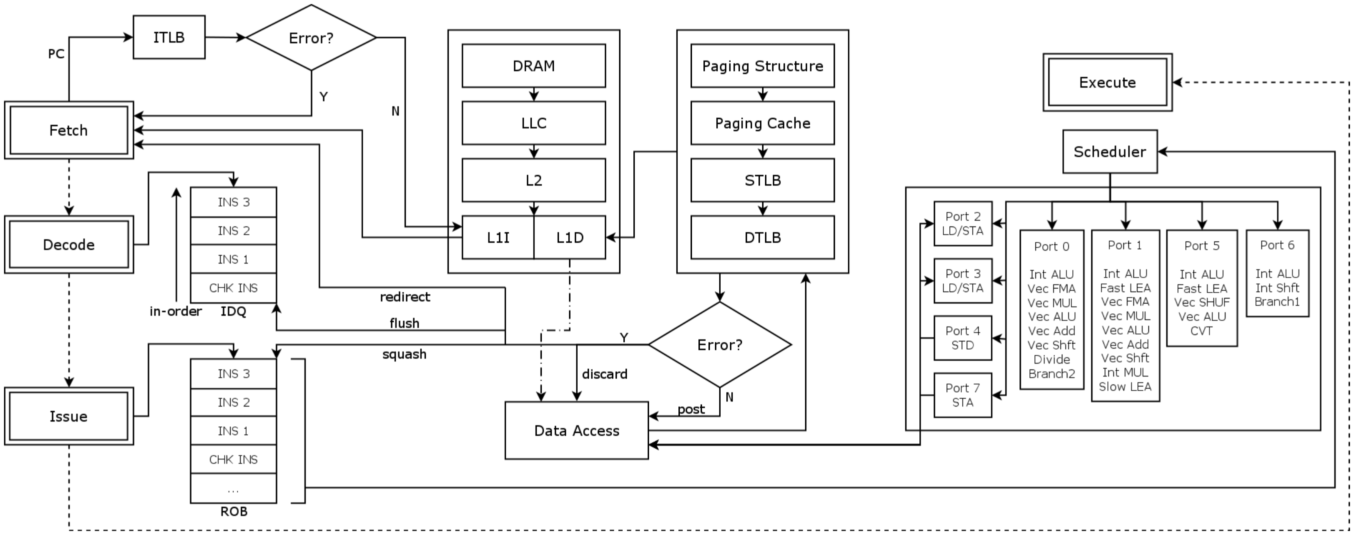


Fig. 1: Instruction execution model of x86 (illustrated using Skylake processors).

into a simpler internal representation called micro-operations (*a.k.a.*, μops). μops are not visible to the programmer and follow the reduced instruction set (RISC) design. This means they use two input operands and one output, all arithmetic and logic operations are performed on register operands and the only instructions that access memory are Loads and Stores. Decoded instructions are added to the Instruction Decode Queue (IDQ) in program order. This marks the end of the front-end of the processor and the last step in which instructions are processed in program order.

Instruction issuing. From the IDQ, μops are issued in FIFO order to the back end of the pipeline. Once issued, they are no longer constrained by program order and can execute out-of-order, as soon as their operands are available. While μops can execute out-of-order, they are required to commit their results to the visible processor state (architectural state) in program order, to preserve correctness. A hardware structure called a reorder buffer (ROB) is used to keep track of the μops program order, while they are in the back-end of the pipeline. The ROB is a table that records all μops in execution and their associated status (*e.g.* operands pending, in-execution, completed, etc.) When issued, μops are added to the ROB, in FIFO order, as long as there are available slots.

Instruction execution and retirement. All μops operands are renamed and their dependencies tracked with the help of hardware structures called reservation stations. Once issued, μops are eligible for execution provided that their operands are ready and execution resources are available. μops are executed out-of-order and in parallel. However, if a μop has data dependency on its preceding μops, it has to wait until the dependency is resolved before being scheduled for execution. When all conditions are met μops are dispatched to the appropriate execution units through hardware structures called ports. Multi-cycle operations can occupy execution units, possibly stalling other μops demanding the same resources.

When μops finish execution they write back their results to so-called physical registers that are not part of the architectural state and are not visible to the program. Results are also forwarded to dependent μops through dedicated bypass data paths allowing dependent μops to be scheduled for execution in the same cycle.

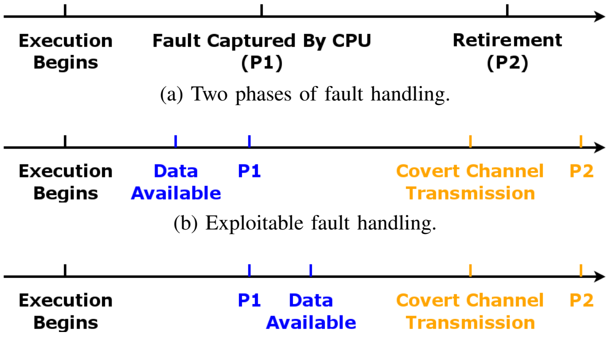
While μops can execute out-of-order, they are required to commit their results to architectural state visible to the program (including architectural registers and memory) in program order. The ROB is used to enforce this requirement by committing and retiring μops in FIFO order. As μops of an instruction reach the head of the ROB, if they have all finished execution, they can commit their results and retire, at which point they are removed from the ROB.

Transient execution relies on hardware to prevent transient instructions from changing the architectural state visible to the program, until instructions are determined to be correct. When mis-speculation is detected (*e.g.* a branch is mis-predicted or an exception is triggered), all mis-speculated instructions have to be squashed. Precise handling of transient state requires that all instructions that precede the first mis-speculated instruction must commit, and all other mis-speculated instructions must be squashed.

2) Memory Accesses and Address Translation: The μops that perform memory accesses are executed in specific execution units. In 32-bit mode, the logical address is first translated into linear address by referencing the segment descriptor; in 64-bit mode, the logical address is the same as the linear address. Then given the linear address of the data in memory, TLB is first consulted to look for the physical address. If the corresponding entry is not available in the data TLB (DTLB), the secondary TLB (STLB) is searched. Similarly, the paging structure cache and page tables in memory are looked up if an STLB miss is encountered. When walking the page tables, the corresponding page directory entries are loaded into the paging structure cache. When the page table entry (PTE) for the 4KB page is eventually located, it will be inserted into STLB and DTLB. Given a physical address from the DTLB, the data fetching starts from the L1 cache which is the fastest in the memory subsystem. Should there be a L1 miss, the L2 cache, the last-level cache (LLC), and the DRAM are checked one by one until there is a hit. Upon a hit, the data will be pushed to all levels of cache and L1 will pass it to the execution units. Processor internal buffers such as Data Cache Unit (DCU), Line Fill Buffer (LFB), Load Buffer (LB) and Store Buffer (SB), as components of L1 cache, are not paid enough attention to by the security community until recent disclosure of transient execution attacks leveraging them [35], [29], [30].

B. Detecting and Handling Mis-Speculation

Although high-level information on instruction execution is documented in the manuals of vendors, the internal fault handling during speculative execution (including out-of-order execution) remains unclear and is complex due to aggressive out-of-order implementation. We propose a two-phase model to understand the internal micro-architectural implementation of fault handling. Our model abstracts the unnecessary, maybe unclear, hardware implementation details into logical events that are directly related to SPEECH. The two phases to be explained not only clarify the exact fault handling scheme but also correspond to the two race conditions faced by an attack. This offers us an opportunity to study them separately and understand them systematically. The model will be later validated through experiments described in Sec. IV.



(a) Two phases of fault handling.

(b) Exploitable fault handling.

(c) Non-exploitable fault handling.

Fig. 2: The two-phase model for fault handling.

The security check starts at the same time as data fetching, but proceeds asynchronously. If the check passes, the fetched data can be finally committed. Otherwise, the processor will handle the fault (*e.g.*, by raising an exception) and clean up the pipeline by discarding the execution results and squashing μops not supposed to execute. As shown in Fig. 2a, the exceptions are handled in two phases: In the first phase (dubbed P1), defined as when the processor detects an error in the μop, the exception is passed to the corresponding execution unit immediately, which reacts to it by stopping the execution of the μop. If the μop performs a data fetching, *two* cases may happen: If the data is not yet retrieved, the fetching will be suspended and a dummy value (*e.g.*, zero) is returned as the data, as in Fig. 2c. If the data loading has already finished at that time, it will not be affected. The data fetching is immediately forwarded to μops in the ROB that are waiting for it. This is demonstrated in Fig. 2b. It in fact describes one of the two race conditions for a SPEECH attack to succeed, between speculative data fetching and processor fault handling.

The second phase of exception handling (dubbed P2) happens when the faulting μop reaches the head of ROB and is ready to retire. The processor checks any pending exceptions with the μops of the retiring instruction and, if detected, the entire execution engine is cleansed in the following way. First, all following μops in the ROB are squashed: already executed μops will never retire and their execution results are discarded; μops that are not yet executed will not be executed any more. Second, the IDQ will stop issuing more decoded μops to ROB. IDQ will be flushed for optimal future performance. Third, information about the exception is saved in relevant registers. Lastly, the front end will be redirected to exception handler. After all preparation is done by the processor, the exception handler of the OS will take over the control of the CPU. This leads to the other race condition for the attack. The covert channel transmission instructions executed speculatively should conclude before P2. Otherwise, they are going to be squashed and never get a chance to transfer the secret to the attacker.

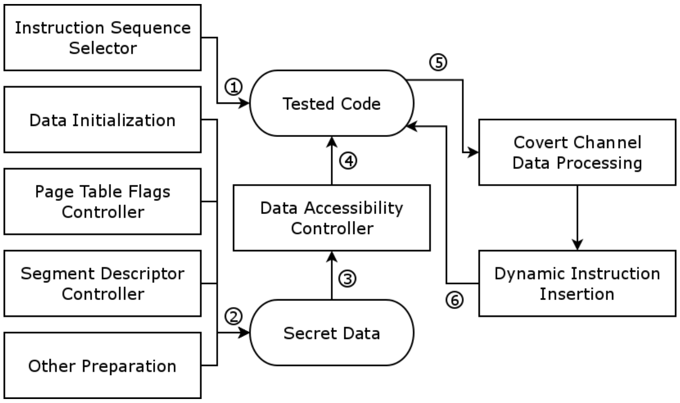

Instruction Sequence Selector
Data Initialization
Page Table Flags Controller
Segment Descriptor Controller
Other Preparation
①
②
Tested Code
Secret Data
③
④
Data Accessibility Controller
⑤
⑥
Covert Channel Data Processing
Dynamic Instruction Insertion

Fig. 3: Architecture and workflow of SPEECHMINER.

III. SPEECHMINER FRAMEWORK

To explore SPEECH vulnerabilities of a processor in an automated (or semi-automated) manner, we designed and implemented a software framework, dubbed SPEECHMINER, in which a sequence of x86 instructions is constructed using templates and executed in a controlled environment. The SPEECHMINER framework provides the users with interfaces to select the instruction sequences for testing and to analyze the results of the tests. The user can use scripting languages to automate large-scale tests using these interfaces.

As the same instruction sequence may exhibit non-deterministic behaviors at the micro-architectural level when executed with different micro-architectural conditions (*e.g.*, cache and TLB conditions, memory bus status, *etc*.), the SPEECHMINER framework is designed to provide control and abstraction of these external conditions. Moreover, the framework is designed to work in both kernel mode and user mode, allowing tests of instruction sequences that operate in both modes; it also supports both 32-bit and 64-bit architectures.

A. Architecture of SPEECHMINER

Fig. 3 illustrates the architecture and workflows of the SPEECHMINER framework. Rectangles represent software components and ellipses represent code/data. SPEECHMINER consists of the following components. The *Instruction Sequence Selector* of SPEECHMINER selects the instruction sequence for testing a certain variant of SPEECH vulnerability. Then a dummy secret data is initialized in the memory and the required page table flags, segment descriptor or other settings are configured. Next, SPEECHMINER starts to run experiments. In each round of the test, the *Data Accessibility Controller* module first sets the desired execution environments such as the status of caches and TLBs. Then the experiment is conducted and raw output data is collected via covert channels. The *Covert Channel Data Processing* module analyzes the raw data, generating either the final analysis results or instructing the *Dynamic Instruction Insertion module* to modify the tested instruction sequence for the next round of experiments.

Instruction Sequence Selector. In each test, one instruction sequence is selected by the Instruction Sequence Selector.

Secret Data Initialization. In many of the tests, the secret value to be extracted is stored in the memory. The SPEECHMINER framework simulates the targeted secret using a 64-bit integer variable (in 64-bit mode) and initializes it to be a specified value (*e.g.*, 0x42000 as used in following code examples). A single MOVQ instruction could load this secret value from the memory into a register. The size of the secret is reduced to 32 bits in the 32-bit mode tests.

Page Table Flag Controller. For cases that require modification of page table entries (PTE), we implemented a kernel module that allows setting or clearing specific PTE flags.

Segment Descriptor Controller. Segmentation is still implemented in all modern x86 processors. When running the tool in the 32-bit mode, segmentation is enabled. Segment Descriptor Controller is provided to generate required segment descriptors in the Local Descriptor Table to trigger exceptions by violating segmentation-related rules.

Other Preparation. Besides these common preparation components, in some cases, SPEECHMINER also needs to take care of some special needs such as configuring memory protection keys [20], enabling SMAP [19], *etc*.

Data Accessibility Controller. To ensure a deterministic execution environment, the SPEECHMINER framework needs to control the status of the cache copies of specific memory blocks and the TLB entries of a specific memory page. The Data Accessibility Controller utilizes preloading and flushing techniques to control TLB and cache entries. Some technical challenges may arise, however.

The preloading of TLB entries may trigger exceptions (which is expected in our design). To preload the TLB entry of a kernel page, the Data Accessibility Controller preloads its TLB entry in a kernel module. To preload TLB entries with `Reserved` flag set or `Present` flag cleared, directly loading the corresponding page can preload entries in the paging structure caches [11, Chapter 4.10]. While valid TLB entries may not be created, invalid entries may be created. Flushing TLB entries are performed in the supervisor model via a kernel module. One method is to leverage the INVLPG instruction which flushes one single TLB entry and the related paging structure cache entries. The other is to reload the CR3 register which will flush all TLB entries and the whole paging structure cache.

Preloading and flushing the cache entries of a data block are performed on its shadow virtual addresses. For each data block that needs fine-grained control of its cache status (*i.e.*, on which cache level a copy is presented), two different virtual address mappings are provided for the same memory page that stores the data block: One mapping is used by the test that triggers exceptions, while the other is used as the shadow virtual address that does not block accesses. To force data in L1, it is directly preloaded via the shadow address. To make the data block in L2 (but not in L1), it needs to be preloaded first and then evicted from L1 using an eviction set [24]. As the LLC cache is shared among all physical cores, after flushing the data to memory (using the CLFLUSH instruction), preloading it from another physical core ensures that the data resides in LLC but not in the L1 or L2 caches (of the tested core). Moreover, as preloading or flushing cache entries also preloads the TLB entry of the page, additional procedures must be taken if this side effect is undesired.

Covert Channel Data Processing. The covert channel signals collected during the test are processed and analyzed. If needed, it provides feedback to the Dynamic Instruction Insertion module to repeat the test with adjusted the instruction sequences to be tested.

Dynamic Instruction Insertion. The module is implemented by altering the code at runtime. It dynamically adjusts a given code sequence according to the need of the experiments (*e.g.* by inserting a certain number of ADD/SUB instructions).

Handling or Suppressing Exceptions. As the tested instruction sequences may trigger exceptions, the framework must handle or suppress exceptions properly. When executed in user mode, exceptions are dealt within signal handlers to ensure compatibility on all hardware; when executed in supervisor mode, the exceptions are suppressed using Retpoline [33], as is done by Stecklina *et al.* [32].

B. Instruction Sequences

To trigger different types of faults, the tested instruction sequences may have distinct structures. Nevertheless, we managed to build a uniform modular template for all tested instruction sequences. Specifically, a template consists of three components: a Windowing Gadget, a Speculation Primitive, and a Disclosure Gadget[2].

Speculation Primitives. A Speculation Primitive consists of one or two instructions that will trigger a fault when executed.

Windowing Gadgets. A Windowing Gadget consists of a sequence of instructions that precedes the Speculation Primitive. It serves two purposes: to enlarge the speculation window and to eliminate side-effects of instruction issuing. These two purposes can be satisfied by delaying the retirement of the Speculation Primitive, which can be achieved by three means: (1) Delaying the retirement of the instructions of the Windowing Gadget. This is because an instruction can retire only when it finishes its execution and all prior instructions have already retired. (2) Making the Speculation Primitive dependent on the execution result of the Windowing Gadget.

[2] The terminologies follow the suggestions from Intel and Microsoft [27].

Thus, the instructions of the Speculation Primitives cannot be executed out-of-order before the dependency is resolved. (3) Occupying the execution units or registers that are also required by the Speculation Primitive. Therefore, typical techniques used by the Windowing Gadgets include accessing non-cached memory blocks, loading memory with a chain of dependency, performing integer ALU operations with a chain of dependency [27].

Disclosure Gadget. A Disclosure Gadget is a sequence of instructions that are speculatively executed, utilizing covert-channel techniques (in collaboration with the Disclosure Primitive to be explained shortly) to measure the speculation windows or the latency of data fetching, *etc*.

```
    movq (%rbx, %rcx, 1), %rbx
```

Listing 1: Example of a Type-I Disclosure Gadget.

- Type-I Disclosure Gadgets only have a single memory load instruction (see Listing 1, all assembly code in this paper follows AT&T syntax.), which we call the *covert-channel sender*. A FLUSH-RELOAD covert channel memory buffer is allocated, which consists of 256 logically consecutive 4KB pages. Each page is considered as one slot of the buffer. FLUSH-RELOAD is performed at the first integer-sized block of each page. Two forms of MOV instructions may be seen in the listings of this paper; whether or not an offset is used by MOV is determined by the values of the related registers.
- Type-II Disclosure Gadgets insert a sequence of ADD/-SUB instructions before the covert-channel sender (see Listing 2). All these instructions have data dependencies on each other, so that they are executed in program order. The execution latency of an ADD or SUB instruction is exactly one cycle, so the total execution cycles can be estimated. An ADD and a SUB instruction are inserted in an alternating pattern so that the resulting value of their operand—the memory address used for the covert channel—does not change significantly, which simplifies the design of the covert-channel receiver. By changing the number of ADD/-SUB instructions in the Windowing Gadget, the framework controls the latency of the execution of the covert-channel sender. Still, the last MOV can optionally include an offset, as shown in Listing 1.

```
    [add $1, %rbx]
    [sub $1, %rbx]
    ...
    movq (%rbx), %rbx
```

Listing 2: Example of a Type-II Disclosure Gadget.

Besides the three components of the instruction sequence template, the SPEECHMINER framework also incorporates a **Disclosure Primitive** to receive signals sent by the Disclosure Gadget. It leverages the FLUSH-RELOAD techniques to determine whether or not certain memory blocks have been accessed by the covert-channel sender of the Disclosure Gadget. As covert-channel communication is subject to noise, the test must be repeated multiple times for the Disclosure Primitive to assert whether or not the covert-channel sender was speculatively executed. SPEECHMINER only requires a binary output from the Disclosure Primitive: whether or not the signal has been received. The results will then be collected and analyzed by the Covert Channel Data Processing module.

C. Speculation Primitives

In this paper, we focus on Speculation Primitives that involve faults. While branch misprediction can also be explained and analyzed in the same two-phase model (see Sec. IV-F), they are vulnerable by design.

To comprehensively measure all possible faults and study their exploitability, we base our tests on the exception list excerpted from the Intel Software Developer Manual [11]. However, not all exceptions are directly related, as they do not serve the first role—they are not guarding secrets. We therefore define two templates of Speculation Primitives:

```
// %RBX: address of a read-only page
    mov $0x42000, (%rbx)
    mov (%rbx), %rbx
```

Listing 3: An example of a two-instruction template.

The first template contains one single Load instruction that triggers exceptions. The second template involves two instructions, with the first causing exceptions by writing to memory or performing checks (*e.g.*, BOUND) and the second speculatively loading data that is influenced by the first (see Listing 3). Note that the constant value 0x42000 used in the example is only for illustration purposes, which can be replaced by other values. This is also true in all the following code snippets. If an exception is not applicable to either of these two templates, it is excluded from the analysis. We comprehensively categorize all such exceptions by their protection mechanism, with a comprehensive list given in Appendix A.

IV. UNDERSTANDING SPEECH VULNERABILITIES

SPEECH vulnerabilities are caused by speculative execution. However, being able to speculatively execute instructions itself does not qualify a vulnerability. The root cause of SPEECH vulnerabilities is that some inaccessible secret data could be accessed by speculatively executed instructions before the processor captures the fault. Moreover, once the secret data is fetched by the speculative instructions, what can be done with it (*e.g.*, leaking the secret using covert channels) is determined by the speculation window—the time period (in CPU cycles) of instructions executed speculatively before the faulting instruction is squashed. Our two-phase fault handling model very well separates the two race conditions:

- *Race Condition I:* data fetching vs. processor fault handling.
- *Race Condition II:* covert channel transmission vs. speculative instruction squashing.

Race Condition I determines *exploitability* and Race Condition II determines the *speculation window*. The two-phase model enables a comprehensive understanding of key factors that determine these two aspects.

In this section, we *first* verify the two-phase model by examining the effects of P1 and P2 using the SPEECHMINER framework. We *then* leverage SPEECHMINER to perform a systematic analysis on the two race conditions. We will show that the analysis enabled by SPEECHMINER helps us explain known attack phenomena and clarify common misunderstandings. Several tests were designed for these goals. The SPEECHMINER framework allows running each of these tests to examine different types of Speculation Primitives. For the clarification and simplicity of discussion, we illustrate these tests using a Speculation Primitive used in the Meltdown-US attack [23]. But notice that the actual instruction sequences may differ for distinct Speculation Primitives.

A. *Confirming Speculative Instruction Squash*

It is known that speculatively executed instructions will be squashed when the processor handles the faults. We empirically verify that *issued but not yet executed μops will not be executed after the squashing*. This fact will be the basis of the following experiments. The experiments were conducted on an Intel i7-7700HQ (KabyLake) machine with Ubuntu 16.04 (Linux 4.4.0-137) as the operating system. Each experiment is repeated for 5 times for reliability.

```
// %RBX: address of uncached covert channel buffer
// %RDX: address of another uncached memory buffer
// *(%RDX) = %RBX
// %RCX: illegal address whose data is 0x42000
// ------------------------------------------------
// Windowing Gadget
     sub %rbx, %rcx
     movq (%rdx), %rbx
// ------------------------------------------------
// Speculation Primitive
     movq (%rcx, %rbx, 1), %rcx
// ------------------------------------------------
// Disclosure Gadget
     [add $1, %rbx]
     [sub $1, %rbx]
     ...
     movq (%rbx), %rbx
```

Listing 4: The effects of speculative instruction squash.

Instruction sequences. As shown in Listing 4, the instructions of the Disclosure Gadget are independent of the data read by the Speculation Primitive. However, because all instructions in the Speculation Primitive and the Disclosure Gadget are dependent on the data fetched in the Windowing Gadget, the Speculation Primitive and the Disclosure Gadget start at the same time. The slow memory fetching also allows enough time for following μops to be issued.

Experiments and expected observations. In this test, the framework tunes the number of ADD/SUB instructions inserted in the Disclosure Gadget. If all issued instructions are eventually executed, we would expect to receive the covert-channel signal regardless of the number of inserted ADD/SUB instructions. Otherwise, the signal should disappear when the number of ADD/SUB instructions increases to a certain threshold.

Results. In the experiments, we observed that when the inserted instructions exceed a threshold, the covert-channel receiver no longer receives any signal from the covert channel. As the number is much smaller than the size of ROB [37], it is not caused by failed issuing due to ROB limits.

Conclusion: Issued but not yet executed μops will be squashed when the exception is handled.

B. *Understanding Effects of* P1

Three sets of experiments were performed to understand the effects of P1 on the current execution unit, other execution units, and the entire execution engine, respectively. The experiments were performed in the same settings as Sec. IV-A.

1) P1 *on Current Execution Unit:* This test is designed to determine how P1 affects the execution unit being used by the Speculation Primitive.

```
// %RBX: address of uncached covert channel buffer
// %RDX: address of another uncached memory buffer
// *(%RDX) = %RBX
// %RCX: illegal address whose data is 0x42000
// ------------------------------------------------
// Windowing Gadget
     movq (%rdx), %rdx
// ------------------------------------------------
// Speculation Primitive
     movq (%rcx), %rcx
// ------------------------------------------------
// Disclosure Gadget
     movq (%rbx, %rcx, 1), %rbx
```

Listing 5: The effects of P1 on the current execution unit.

Instruction sequences. As shown in Listing 5, the instruction sequence consists of a Windowing gadget, a Speculation Primitive, and a Disclosure Primitive. The Speculation Primitive is a simple slow memory load to ensure that the retirement latency of the Speculation Primitive remains constant by postponing it to a late enough fixed time. Thus, influence of P2 is excluded from this experiment.

Experiments and expected observations. The tests were repeated four times, with the secret data placed in L1D cache, L2 cache, LLC, and memory, respectively. The TLB entry of the secret's address is always flushed to ensure a fixed P1 latency. In these experiments, we would hope to see whether the changes of the data fetching latency affects the covert-channel signal received by the Disclosure Primitive. If so, whether P1 happens before the data is fetched affects the return values of current execution unit.

Results. We observed that only when the secret data is stored in the L1 cache could the correct covert-channel signal be received. When the secret data is store in L2, LLC, or the memory, a zero signal is received. This observation validates our theory in II-B. The execution unit terminates after catching the exception; a dummy value of zero is returned as the result of the execution. We will show that P2 is not relevant in this experiment in Sec. IV-C.

Conclusion: P1 terminates the current execution unit. If the latency of P1 is greater than the data fetching latency, the correct value can be propagated to the speculative instructions; otherwise, a zero value will be returned.

2) P1 *on Other Execution Units:* As P1 terminates the current execution unit, it is directly related to the exploitability. However, it is not yet clear the exploitability is also affected by P1 of other faulting instructions. If so, attacks may be enhanced by combining two or more Speculation Primitives.

Instruction sequences. Different from other tests, two Speculation Primitives are included in this test to determine whether

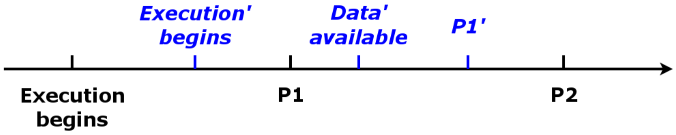


Fig. 4: Illustration of P1 effects on other execution units.

P1 of the first Speculation Primitive also influences the execution unit used by the second Speculation Primitive. We consider two cases, depending on whether the two Speculation Primitives access the same memory address.

First, the two Speculation Primitives access the same memory address. The instruction sequence is designed as shown in Listing 6. A sequence of inter-dependent ADD/-SUB instructions are inserted between the two Speculation Primitives to control the delay of the execution of the second Speculation Primitive. Since the first inserted ADD instruction and the first Speculation Primitive both have data dependency on the Windowing Gadget, they start at the same time. But the second Speculation Primitive has to wait until all the inserted ADD/SUB instructions finish. The Disclosure Gadget is used to monitor the data fetched by the second Speculation Primitive.

Second, the two Speculation Primitives access different memory addresses. The instruction sequence used is the same as the first case, except that the memory addresses accessed by the two Speculation Primitives are different.

```
// %RBX: address of uncached covert channel buffer
// %RDX: address of another uncached memory buffer
// *(%RDX) = %RBX
// %RCX: illegal address #1
// %RAX: illegal address #2 whose data is 0x42000
// ----------------------------------------------
// Windowing Gadget
   sub %rbx, %rax
   sub %rbx, %rcx
   movq (%rdx), %rbx
// ----------------------------------------------
// Speculation Primitive #1
   movq (%rcx, %rbx, 1), %rcx
// ----------------------------------------------
// special inserted instructions for the experiment
   add %rbx, %rax
   [add $1, %rax]
   [sub $1, %rax]
   ...
// ----------------------------------------------
// Speculation Primitive #2
   movq (%rax), %rax
// ----------------------------------------------
// Disclosure Gadget
   movq (%rbx, %rax, 1), %rbx
```

Listing 6: The effects of P1 on the other execution units

Experiments and expected observations. In both tests, the relevant data are stored in the L1D cache. As shown in Fig. 4, the events with black labels describe the first Speculation Primitive, and the ones with blue labels describe the second. By delaying the execution of the second Speculation Primitive, P1 of the first Speculation Primitive can happen before the second Speculation Primitive fetches the secret data. Therefore, in the experiment, we gradually inserted more instructions between the two Speculation Primitive to delay the start of the second Speculation Primitive.

Results. In both experiments, regardless of whether the two Speculation Primitives accesses the same memory addresses or not, by gradually inserting instructions between them, we never witnessed that the received covert-channel signal changes from the correct value to zero. Instead, we only observed that after some threshold, the signal disappears. This suggests that P1 of the first Speculation Primitive does not influence the other execution units (by zeroing their results) but its P2 does (by squashing their execution). Therefore, P1 of the first Speculation Primitive does not affect other execution units.

Conclusion: Performing transient execution attacks with two or more Speculation Primitive does not increase the exploitability.

3) P1 *on Execution Engine:* To confirm that P1 has nothing to do with the speculation window, we show that P1 does not influence other components of the execution engine, *e.g.*, by squashing speculative μops in ROB or altering code fetching in the front end.

Particularly, we define *speculation window* as the maximal number of CPU cycles from the beginning of the speculative execution till all speculatively executed instructions are squashed. SPEECHMINER enables us to indirectly measure the speculation window in the following test.

Instruction sequences. The design is close to Listing 4. The only difference is that after Line 9, a memory load instruction (*i.e.*, movq (%rax, %rbx, 1), %rax) is added.

Experiments and expected observations. The strategy of the test is to change the latency of P1 while fixing the latency of P2. If the measured speculation window does not change according to P1 latency, P1 has no effect on the entire execution engine.

To change the latency of P1, we control the TLB status of the page storing the secret data—by preloading or flushing the TLB entry. In this way, P1 of the Speculation Primitive changes accordingly.

To fix P2 latency, one additional memory load instruction is added in the windowing gadget, which begins to execute at the same time as the Speculation Primitive and the Disclosure Gadget. The goal of this instruction is to delay the retirement of all subsequent instructions, so that the retirement of the Speculation Primitive waits on the retirement of this memory load instruction. In this way, the P2 latency is not determined by the the Speculation Primitive itself, which changes according to TLB presence, but by the retirement of the memory load instruction. To achieve this goal, the data to be loaded by this instruction is placed in L2 cache and the TLB of the corresponding page is flushed.

Results. When changing the P1 latency, we did not observe any changes in the speculation window by counting the maximal ADD/SUB instruction numbers in the Disclosure Gadget that still allows the last covert channel access instruction to execute.

Conclusion: P1 does not affect the entire execution engine; altering P1 does not change the speculation window.

C. *Understanding Effects of* P2

The following test aims to confirm that P2 squashes all speculative instructions and P2 can be manipulated.

```
// %RBX: address of uncached covert channel buffer
// %RCX: illegal address whose data is 0x42000
// ——————————————————————————————
// Windowing Gadget
    movapd \%xmm0, \%xmm1
    addpd \%xmm1, \%xmm0
    [cpuid]
    mulpd \%xmm1, \%xmm0
    ...
    movapd \%xmm0, \%xmm1
    addpd \%xmm1, \%xmm0
    mulpd \%xmm1, \%xmm0
// ——————————————————————————————
// Speculation Primitive
    movq (%rcx), %rcx
// ——————————————————————————————
// Disclosure Gadget
    [add $1, %rcx]
    [sub $1, %rcx]
    ...
    movq (%rbx, %rcx, 1), %rbx
```

Listing 7: Tuning P2 latency.

Instruction sequences. The instruction sequence is shown in Listing 7. As the retirement of the Speculation Primitive only happens after that of the Window Gadget, the strategy is to manipulate the latter and look for any changes in the spelucation window.. The Windowing Gadget consists of 25 repeated sequences of three SSE2 instructions: MOVAPD, ADDPD and MULPD—thus 75 instructions in total. These floating point instructions are slow but can be executed in parallel with the Speculation Primitive and the Disclosure Gadget. Each of these floating point instructions has data dependency on its predecessor.

To fine tune the retirement of the Speculation Primitive, a CPUID instruction is inserted in the Windowing Gadget. As the CPUID instruction serializes the execution of the instructions before and after it (*i.e.*, no instruction is issued before CPUID retires), only the floating point instructions after CPUID are effective in the Windowing Gadget. Therefore, the retirement of the Speculation Primitive is further delayed if there is CPUID is inserted earlier in the floating point instructions; and vice versa. The Disclosure Gadget is of type-II. All instructions are dependent on the Speculation Primitive.

Experiments and expected observations. For each position in the Windowing Gadget where CPUID is inserted, the framework automatically alter the number of ADD/SUB instructions to identify the speculation window as in Sec. IV-B3. In this way, the correlation between P2 and the speculation window can be observed.

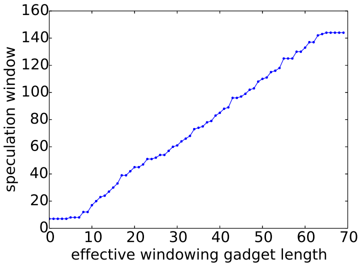


Fig. 5: Size of the effective Windowing Gadget vs. speculation windows.

Results. In Fig. 5, the x-axis is the effective size of Windowing Gadget (tuned by moving the position of CPUID) and the y-axis is the speculation window. By gradually moving the position of CPUID from the beginning of the Windowing Gadget to the end, which changes the effective length of Windowing Gadget, the speculation window also grows accordingly. Prior studies [23] have reported the speculation window of certain attack variants. However, our experiment suggests that it is not meaningful to report the size of the speculation window as it can be changed in manners described above. Despite that, the speculation window is still limited by the size of the ROB. In Fig. 5, the maximum speculation window is about 140 cycles, reflecting the ROB size of greater than 140 instructions (as each ADD/SUB instruction takes 1 cycle). And this is already big enough for covert channels to transmit data through one memory operation at a time.

Conclusion: The speculation window of any Speculation Primitive can be altered by delaying its P2.

D. *Investigating Race Conditions*

A successful exploitation of SPEECH depends on the outcome of the race conditions: (i) Data fetching latency must be lower than P1 latency, and (ii) speculative covert channel transmission should be faster than P2. As such, we leverage SPEECHMINER to investigate the following questions:

- Is it possible to quantitatively measure the race conditions?
- Is it possible to control the outcomes of the race conditions?

1) Revisiting Race Condition II: Sec. IV-C already demonstrated that we are able to quantitatively measure Race Condition II by evaluating the speculation window. Moreover, Fig. 5 suggests that by altering the retirement of the Windowing Gadget, the attacker is able to delay P2 of the Speculation Primitive.

Conclusion: The attacker can always win Race Condition II by delaying P2 of the Speculation Primitive.

2) Measuring SPEECH *Exploitability:* As the outcome of Race Condition II can be controlled, a successful attack depends solely on the outcome of Race Condition I. Therefore, by measuring the outcome of Race Condition I, SPEECHMINER enables automated tests of the *exploitability* of all possible exception-based variants on various processors. It leverages its ability to enumerate possible combinations of execution conditions such as cache and TLB presence to determine the exploitability under the most optimal condition.

We performed tests on 9 machines (2 laptops, 5 desktops and 1 cloud VM). All tested machines run Ubuntu 16.04 with Linux kernel 4.4.0-137 (or 4.4.0-141 for compatibility issues on newer hardware) and KPTI is turned off. All the Intel microcode versions are rolled back to version 20171117 (except for Coffee Lake, which does not have older microcode, and the cloud VM, which we cannot control). The AMD microcode version is 3.20180515.1. The test can be extended to evaluating patched microcode and countermeasures, such as KPTI [21] and PTE inversion [22]. We would like to open source SPEECHMINER to enable other researchers to perform tests in other settings.

Instruction sequences. As shown in Listing 8, the Windowing Gadget has a memory load instruction that retires slowly due to long latency of memory access, but it does not have dependency on the previous instructions, nor does any subsequent instructions depend on it. It is used to ensure that the retirement of the Speculation Primitive (the P2 latency) is sufficiently delayed.

```
// %RBX: address of uncached covert channel buffer
// %RDX: address of another uncached memory buffer
// *(%RDX) = %RBX
// %RCX: illegal address whose data is 0x42000
// ————————————————————————————————
// windowing gadget
    movq (%rdx), %rdx
// ————————————————————————————————
// speculation primitive
    movq (%rcx), %rcx // could be any illegal inst.
// ————————————————————————————————
// disclosure gadget
    movq (%rbx, %rcx, 1), %rbx
```

Listing 8: Exploitability test with P1 measurement.

Experiments and expected observations. Each tested instruction sequence was executed under a variety of conditions, with varying data access latency (cached in L1D, L2, or LLC) and address translation latency (whether or not TLB entries are created for the corresponding pages). SpeechMiner is able to enumerate all possible combinations to achieve the optimal condition. In each test, if the Disclosure Primitive receives the correct signal from the covert channel, the vulnerability is exploitable. Otherwise, if a zero signal is received, the vulnerability is not exploitable as the P1 latency is shorter than data available latency. However, if no signal can be received, it suggests speculation is not allowed by the variant.

Results. The results are shown in Table I and a reference of the tested variant names to their description can be found in Appendix A. Some variants are unable to be tested due to lack of hardware support or OS support on certain machines and they are marked as N/A. On the tested Intel machines, Meltdown-US (accessing supervisor memory page from user space) and Meltdown-RW (writing to a read-only memory page) are exploitable while the AMD machine shows no speculation. When testing Meltdown-Present (`Present` flag cleared) and Meltdown-Reserved (`Reserved` flag set), signal handler cannot be used since the whole OS will crash. Thus, only the machines also equipped with Intel TSX [15] are tested and reported. Loading restricted registers (CR4 and MSR) on all tested machines show that no speculative load is allowed. Meltdown-MPK (bypassing the restriction of memory protection keys) could only be tested on Amazon EC2 E5 instance and it was found exploitable. Meltdown-FP (accessing a lazy-context-switch float pointer register) is also found exploitable on Intel machines. Meltdown-BR (accessing an array with an over-range index) is found exploitable on all tested machines although the BOUND instruction raises an exception when it discovers the violation. In 32-bit mode, segmentation is enabled and thus relevant variants could be tested. Most of them are not exploitable or does not allow further speculation at all, since paging checks has to wait until segmentation translation produces a linear address while segmentation check begins at the same time as the translation. However, still some of them are found exploitable on either Intel or AMD platform.

In our experiments, a few new variants were found by the SpeechMiner framework. *We have reported these new variants to Intel and AMD already.*

- *Supervisor mode access violating SMAP (Intel & AMD).* Supervisor Mode Access Prevention (SMAP) forbids code in the kernel mode from accessing to user-space addresses. However, as shown in Table I, on some of our tested machines, SMAP can be bypassed using speculative execution when the secret data is in the L1 cache. When the data is in lower-level caches, zero signal is captured.
- *Supervisor mode access bypassing MPK (Intel).* When a user space page is set to be inaccessible using Memory Protection Key (MPK), its accesses from kernel code is also forbidden, which triggers a page fault exception. However, as demonstrated on the tested cloud server (see Table I), if the secret data is already in the L1 cache, it can be leaked through speculatively executed Disclosure Gadget.
- *Memory writes to read-only data segments (Intel).* Segmentation is used in the 32-bit mode. However, memory writes to a read-only data segments can be speculatively used by following instructions. In this case, the caching and TLB status does not affect the exploitability. This vulnerability is similar to the `Meltdown-RW` [16], but its security implication is different: `Meltdown-RW` takes advantage of store-to-load forwarding. As the store buffer is indexed by the linear address (not logical address), the forwarding is speculative as the logical-to-physical translation is not yet finished. Therefore, it is not surprising that the permission check (using flags in the PTE) happens after the store-to-load forwarding. However, in contrast, the segmentation check is performed during the translation from logical addresses to linear addresses. Thus, when the store buffer queues the store instruction to the given linear address [9], the segmentation access privilege check should already be done. However, our test suggests that this is not the case as a following load could directly use the store value. This validates the conclusion of the recent Fallout attack [29] that store buffers predict aggressively using only the lowest bits of addresses.
- *Reading from a logical address over the limit of segment (AMD).* When tested in 32-bit mode, a load to a logical address beyond the segment limit is forbidden, which triggers an exception. However, we have found that the segmentation check can be bypassed by speculative execution. The vulnerability is exploitable when the TLB entry of the page is present and the data is cached in the L1 cache. The same vulnerability cannot be found in Intel processors.

> **Conclusion:** SpeechMiner enables automated tests of Speech vulnerabilities on various processors. It detects several new variants of transient execution attacks.

Extended study on state-of-the-art mitigation. KPTI [21] is a software solution designed to prevent the exploitation of the Meltdown-US vulnerability, but not removing the vulnerability from hardware. Since the test cases of SpeechMiner are designed by explicitly setting the bits in PTE, KPTI should not place any influence on it. On the other hand, the existing microcode patches are for Spectre variants and L1TF only [28] and thus is expected not to affect the tests. We performed the tests on Laptop 1 with the latest microcode patch and found

Variant	Laptop 1 KabyLake	Laptop 2 KabyLake	Desktop 1 Haswell-EP	Desktop 2 SandyBridge	Desktop 3 Westmere-EP	Desktop 4 CoffeeLake	Desktop 5 KabyLake	Desktop 6 AMD EPYC	Cloud 1 Skylake-SP
PTE (Present)	Y	N/A	N/A	N/A	N/A	Y	Y	N/A	Y
PTE (Reserved)	Y	N/A	N/A	N/A	N/A	Y	Y	N/A	Y
PTE (US)	Y	Y	Y	Y	Y	Y	Y	R	Y
Load CR4	R	R	R	R	R	R	R	R	R
Load MSR (0x1a2)	R	R	R	R	R	R	R	N/A	N/A
Protection Key (User)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Y
Protection Key (Kernel)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Y
SMAP violation	Y	Y	N/A	N/A	N/A	Y	Y	Y*	Y**
PTE (write w/ RW=0)	Y	Y*	Y	Y	Y	Y	Y	R	Y
Load xmm0 (CR0.TS)	Y	Y	Y	Y	Y	Y	Y	N/A	N/A
BOUND (32-bit)	Y	Y	Y	Y	Y	Y	Y	Y	Y
DS Over-Limit (32-bit)	N	N	N	N	N	N	N	Y	N
SS Over-Limit (32-bit)	N	N	N	N	N	N	N	Y	N/A
DS Not-Present (32-bit)	R	R	R	R	R	R	R	R	R
SS Not-Present (32-bit)	R	R	R	R	R	R	R	R	R
DS Execute-Only (32-bit)	R	R	R	R	R	R	R	R	R
CS Execute-Only (32-bit)	R	R	R	R	R	R	R	R	R
DS Read-Only (write, 32-bit)	Y	Y	Y	Y	Y	Y	Y	R	Y
SS Read-Only (32-bit)	R	R	R	R	R	R	R	R	R
DS Null (32-bit)	N	N	N	N	N	N	N	R	N
SS Null (32-bit)	R	R	R	R	R	R	R	R	R
SS $DPL \neq CPL$ (32-bit)	R	R	R	R	R	R	R	R	R

TABLE I: Exploitability evaluation on different machines. Y: exploitable. N: non-exploitable. R: no speculative execution. N/A: unable to test. Y*: both expected data and zero data are captured in all covert channel reloads. Y**: the only exploited case with success rate not 100% (about 60%). Laptop 1: i7-7820HQ (Kaby Lake). Laptop 2: i7-7700HQ (Kaby Lake). Desktop 1: Xeon E5-1607v3 (Haswell-EP). Desktop 2: i3-2120 (Sandy Bridge). Desktop 3: Xeon E5620 (Westmere-EP). Desktop 4: Xeon E-2124G (Coffee Lake). Desktop 5: i7-7700 (Kaby Lake). Cloud 1: Amazon EC2 C5 instance. Desktop 6: AMD EPYC 7251.

all vulnerabilities still present, including Meltdown-US.

3) Quantitatively Measuring P1 *Latency:* A more powerful measurement could be done with SPEECHMINER to quantitatively measure the relative latency of P1 compared to data fetching.

```
// %RBX: address of uncached covert channel buffer
// %RCX: illegal address whose data is 0x42000
// ——————————————————————————————————————————————
// Suppressing Primitive
     [movq (%rax), %rax] // legal access
     [movq (%rax), %rax] // legal access
     ...
     movq (%rax), %rax // suppressing w/ exception
// ——————————————————————————————————————————————
// Speculation Primitive
     movq (%rcx), %rcx // could be any illegal inst.
// ——————————————————————————————————————————————
// Disclosure Gadget
     [add $1, %rcx]
     [sub $1, %rcx]
     ...
     movq (%rbx, %rcx, 1), %rcx
```

Listing 9: Quantitative measurement of P1 latency

Instruction sequences. The construction of the instruction sequence is different from other tests as a *Suppressing Primitive* is needed in the test. The Suppressing Primitive precedes all other components in order to conceal the effect of executing the instruction sequence. This is achieved by ensuring that the instructions executed in all other components never retire. In the example of Listing 9, the Suppressing Primitive is simply an illegal memory load from address 0, but it can also be implemented using conditional branches, indirect jumps, or retpoline. The Suppressing Primitive creates a fixed speculation window for the rest of the instruction sequence to execute. Because it is desired that this speculation window is greater than the one created by the Speculation Primitive, the Suppressing Primitive includes a few memory loads before the faulting instruction and leverages the pointer chasing technique [1] to further enlarge its speculation window. The speculation window cannot be longer than the size of the ROB.

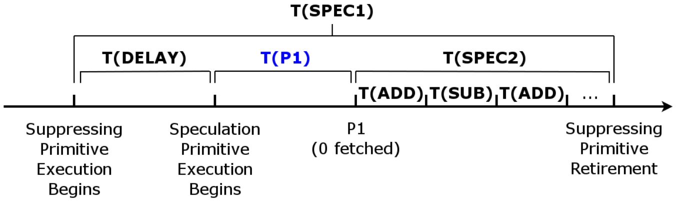

T(SPEC1)
T(DELAY)
T(P1)
T(SPEC2)
T(ADD)
T(SUB)
T(ADD)
...
Suppressing Primitive Execution Begins
Speculation Primitive Execution Begins
P1 (0 fetched)
Suppressing Primitive Retirement

Fig. 6: Illustration of quantitatively measurement of P1 latency.

A type-II Disclosure Gadget is used; all its instructions depend on the Speculation Primitive. Thus, the Disclosure Gadget only begins execution after the data is returned from the Speculation Primitive, regardless of its correctness.

Experiments and expected observations. By flushing the secret data to memory, the framework ensures that a zero data is returned by the Speculation Primitive. Let T_{P1} be the number of cycles to retrieve the zero data by the Speculation Primitive, T_{spec1} be the speculation window of the Suppressing Primitive, and T_{delay} be the latency for the Speculation Primitive to begin execution, which are both fixed. The number of inserted instructions in the Disclosure Gadget is tuned to determine T_{spec2}, which is the speculation window of the Speculation Primitive. As the Disclosure Gadget starts after the Speculation Primitive has retrieved the data, we have $T_{delay} + T_{P1} + T_{spec2} = T_{spec1}$ (shown in Fig. 6).

To estimate T_{P1}, a control test is run. In the control test, the Speculation Primitive accesses a legal data, which does not trigger exceptions. But still, the execution will be reverted due to the Suppressing Primitive. In this case, the number of cycles to retrieve the legal data is T_{data}. With all other conditions unchanged, we have $T_{delay}+T_{data}+T_{spec2'} = T_{spec1}$. Without the need of calculating the exact values of T_{spec1} and T_{delay}, the relationship between T_{data} and T_{P1} can be estimated in a differential manner: $T_{data} - T_{P1} = T_{spec2} - T_{spec2'}$.

Results. We ran the tests on Meltdown-US as an example and found that $T_{data}-T_{P1} = 0$, which means P1 and L1 cache data fetching arrives at the same cycle. This also explains why most attacks cannot work when the data is in higher-level caches (unless the effects of prefetching is exploited, see Sec. IV-E).

We also performed the tests on negative results in Table I. For example, when accessing data beyond the segment limit, data in Table I suggest the vulnerability is not exploitable. By running the test to quantitatively measure T_{P1}, we found out that $T_{data} - T_{P1} = 0$, which means P1 come right before the data is available. However, when data is available in the L2 cache, we find $T_{data} - T_{P1} = -12$, which suggests P1 come 12 cycles earlier than the data is available. Due to SPEECHMINER's ability to quantitatively measure a negative relative P1 latency, we are able to affirmatively claim the inexploitability of certain variants on given hardware.

Conclusion: SPEECHMINER enables quantitative measurement of Race Condition I, providing security assurance for the negative results of the exploitability tests.

4) Controlling Race Condition I: The ability to quantitatively measure the relative P1 latency also enables the exploration of the controllability of Race Condition I. Given a certain variant of SPEECH vulnerabilities, we can leverage SPEECHMINER to alter one factor (*e.g.*, TLB entry status) while keeping all others unchanged. Then, SPEECHMINER is able to determine how the relative P1 changes according to the tested factor. For example, we found that the absence of TLB entry leads to a decrease of the relative P1 latency by over 100 cycles when testing the variant violating segmentation limit. However, we did not find such an effect when testing it on Meltdown-US attacks. We leave a comprehensive examination of all variants and all possible factors to future work.

E. Speculation Primitive as Prefetcher

Given the study on the race conditions for exploitation in ONE round of attack, an attack could be guided to create an optimal attack scenario. However, not all of the resources are within the control of the attacker. Thus, the following question is whether the attacker is able to change those conditions that controls/influences the race condition. How a sophisticated attacker can manipulate victim is beyond the scope of this paper, so we only study whether ONE round of attack itself benefits the race conditions of next round of attack. Take Meltdown-US and Meltdown-P (the base of L1TF) as an example, the only influencing condition towards the exploitability is data fetching latency as analyzed in Sec. IV-D4. We are unable to test buffers for now, so we focus on caching.

Some Speculation Primitive may only lead to exploitable vulnerability when the data fetching latency is small—the data is already cached in the L1 cache—but others may succeed even when the data is completely uncached. We speculate the root cause is that some Speculation Primitive, though failed to extract secret from L2, LLC, or memory, could work as a prefetcher to preload secret data into L1 caches so as to facilitate future attacks of the same kind. We empirically validate this hypothesis.

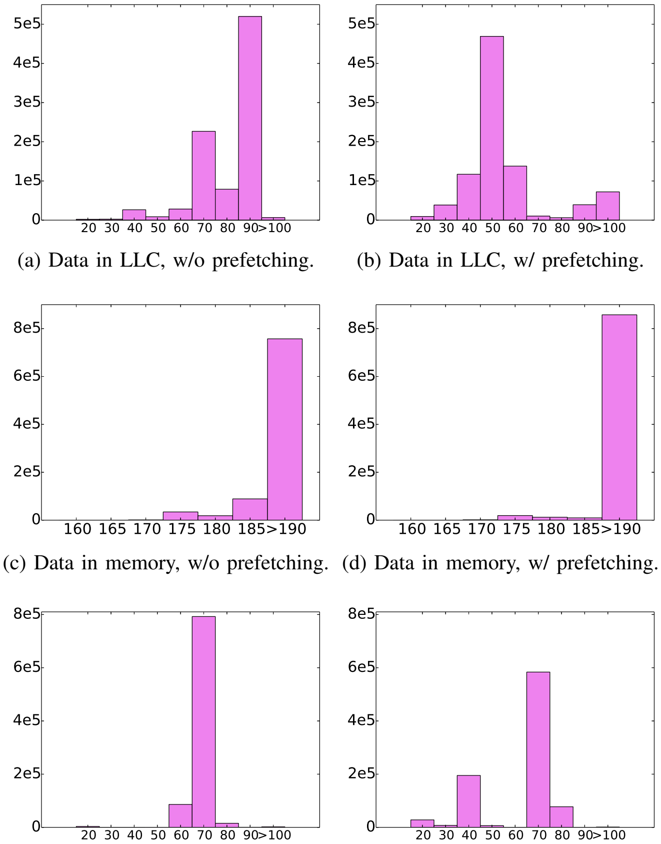
Experiments. In this test, Windowing Gadget and Disclosure Gadget are unnecessary. The experiment is conducted in three steps: First, the secret data to be accessed by the Speculation Primitive is preloaded (from the shadow virtual address) into the LLC or memory. Second, the tested Speculation Primitive is executed N times (with the exceptions suppressed by the SPEECHMINER framework). Third, the data is reloaded from the shadow address and the latency is measured. Two versions of this experiment were tested: (1) $N = 0$; (2) $N = 1000$.

Results. When the Meltdown-US variant is selected as the Speculation Primitive, the result of the experiment is shown in Fig. 7. In particular, Fig. 7a ($N = 0$) and Fig. 7b ($N = 1000$) show the latency of reloading the secret data that is already in the LLC. Clearly with speculative prefetching, the reload latency drops to the range that is close to L2 cache hit. In addition, if a Disclosure Primitive is used to monitor the covert channel while measuring the reload latency, 1/1000 of the time the correct signal can be received while other times a zero signal is. Because in our previous test we have confirmed that only when data is placed in the L1 cache could it be leaked through speculative execution, we conclude that if the data is already in the the L2 cache, the Speculation Primitive has a probability of prefetching it into the L1 cache. In contrast, in Fig. 7c and Fig. 7d, the reload latency distribution does not change with and without prefetching.

We conducted another experiment to validate our analysis. This time, the Speculation Primitive accesses a memory page with its `present` flag cleared. We repeated the experiments with the data originally stored in the LLC. The results are shown in Fig. 7e ($N = 0$) and Fig. 7f ($N = 1000$) . Terminal faults have different prefetching effects. After 1000 rounds of Speculation Primitive execution, although some data is preloaded to L2, the peak still keeps at around 70 cycles (LLC). Only zero signals can be received from the covert channel.

Conclusion: In Meltdown-US, if the data is already in the LLC, the Speculation Primitive may prefetch it to the L2 cache and with some probability the L1 cache; if the data is in the memory, the Speculation Primitive cannot prefetch it to the LLC. In Meltdown-P, prefetching with terminal faults is almost impossible to load the data to L1D.

Misunderstanding regarding Meltdown-US. Since the publication of the first Meltdown-US paper, there has been a debate whether the attack can be successful if the data is not cached (*i.e.*, stored in the memory). Our work concludes that with only *one round* of Meltdown-US attack, it is only possible to leak the data if it is already in the L1 cache. However, as in most demonstrated Meltdown-US attacks, *multiple rounds* of attack are performed, the secret data can be prefetched to the L1 cache if it is already in the LLC or L2. These findings coincide with our conclusions. Our study provide an explanation for the prior works. In fact, no one has demonstrated Meltdown-US to leak non-cached data—unless the data has been preloaded into the Line Fill Buffer by a thread running in parallel. We



(a) Data in LLC, w/o prefetching.

(b) Data in LLC, w/ prefetching.

(c) Data in memory, w/o prefetching.

(d) Data in memory, w/ prefetching.

(e) Terminal fault, data in LLC, w/o prefetching.

(f) Terminal fault, data in LLC, w/ prefetching.

Fig. 7: Reload latency distribution after using Speculation Primitive as a prefetcher.

have confirmed this fact with the Meltdown-US authors. On the other hand, our study also confirms that L1TF attacks can only succeed when data is already in the L1 cache, the mechanisms of which have not been thoroughly discussed in prior works.

F. *Misprediction Handling and Spectre-type Attacks*

Branch misprediction handling. As an extended study, the two-phase model also applies to Spectre-type misprediction handling. When branch instructions are executed in the back end to determine the branch target address, they leverage the branch prediction units (BPU) to make predictions before the execution completes to improve performance. BPU is a front-end component, making instruction fetching to be immediately redirected to the predicted branch target address. However, if the branch execution unit detects a mismatch between the predicted target address and the true target address, a misprediction is captured by the processor. We expect the misprediction handling is different from exception handling: in P1 of misprediction handling, everything required for the handling is performed. The true branch target address is passed to the front end right away once it is determined. All subsequent μops in ROB are squashed. The IDQ is also flushed. There is no point for the handling to wait until requirement, especially for performance consideration.

Test goals. The test aims to verify whether the speculative execution is terminated as soon as the prediction is found to be incorrect, *i.e.*, at P1 of the branch instruction, in contrast to at P2 of the exception handling.

```
// %RAX: address of uncached memory buffer
// %RCX: memory address whose data is an address
// to a covert channel buffer
// ------------------------------------------------
// windowing gadget
   [movq (%rax), %rax]
// ------------------------------------------------
// speculation primitive
   branch // poisoned to disclosure gadget
   ...
// ------------------------------------------------
// disclosure gadget
   movq (%rcx), %rcx
   [add $1, %rcx]
   [sub $1, %rcx]
   ...
   movq (%rcx), %rcx
```

Listing 10: Examing misprediction handling.

Experiments. It is already known that P2 can be delayed by deferring the retirement of the branch instruction. Thus, the key idea of the experiment is to determine whether the speculative window grows accordingly.

The experiment still follows the basic design of the instruction sequences, but the Speculation Primitive is now a branch instruction. Following the sample code in Listing 10, the branch instruction is poisoned to speculatively execute the Disclosure Gadget. The first speculative instruction in the Disclosure Gadget loads a value. The last instruction uses this value as the address of a covert channel buffer. Then SPEECHMINER gradually inserts as many ADD/SUB instructions between them as possible for the last instruction to still be executed. The execution of the last instruction is determined using a FLUSH-RELOAD covert channel Disclosure Primitive. The speculative window is measured by counting the maximum number of permitted ADD/SUB instructions.

The experiment requires a differential analysis. In the first run of the test, no Windowing Gadget is implemented. In order to slow down the retirement of the branch instruction, a slow memory load instruction is placed in the Windowing Gadget in the second run for comparison. Given different P2 latency, we observe whether the speculative window are also different.

Results. Compared to exception handling for Meltdown-type attacks, misprediction handling is known to be exploitable. The only characteristics of interest is the speculation window. During our experiments, it is found that the speculation window remains unchanged with or without the slow Windowing Gadget. Therefore, we conclude that the speculation window of a branch instruction is only determined by P1 but not P2. This is reasonable since branch prediction is designed to optimize performance. Squashing the instructions on the wrong path as soon as possible helps speeding up the execution of the instructions on the correct path.

Conclusion: Misprediction is handled at P1. Speculative execution stops as soon as misprediction is captured.

V. Discussion

Extending SpeechMiner. SpeechMiner can be extended to analyze other Speech variants systematically. For example, the recently disclosed RIDL and Zombieload attacks exploit processor internal buffers (*e.g.*, LFB). Specifically, the Data Accessibility Controller can be extended to control the status of these buffers. We will leave such implementation to future work. In addition, hardware extensions such as SGX, TSX, and VMX can also be tested by SpeechMiner. However, some difficulties may arise: First, it is impossible to identify a comprehensive list of faults for these extensions. Second, some faults may be handled silently, without triggering exceptions.

Limitation. There are still limitations to SpeechMiner and we also consider countering them in the future work. First, SpeechMiner requires some manual efforts to construct the tests from the exception lists collected from vendor manuals. For each exception type in the manual, we still need to determine whether it involves security protection (which is desired) and if so, whether it forms a one-instruction or a two-instruction Speculation Primitive, as demonstrated in Appendix A. Then based on the conditions to trigger the exception, we need to manually select a Speculation Primitive and develop scripts to automate the tests. Second, SpeechMiner is unable to perform tests on the variants that trigger micro-architecture events that cannot be unobserved by the Disclosure Gadget (using covert channels). Third, SpeechMiner cannot test exceptions not described in the manufacturer's manual.

VI. Related Work

Closest to our study is by Canella *et al.* [5] that aims to systematically categorize Speech attacks. In their work, these attacks are classified into two categories: Meltdown-type [23] and Spectre-type [17]. Meltdown-type attacks consider attacks from untrusted but confined programs; Spectre-type attacks assume a benign program tricked to speculatively execute unintended control flows. Both this work and Canella *et al.* [5] aim for comprehensive analyzing attack variants. While Canella *et al.* focuses on attack taxonomization, however, our work emphasizes on the understanding and modeling of fault handling mechanisms. Moreover, a core contribution of our work is the SpeechMiner framework that helps automate testing of such vulnerabilities on commodity processors. Moreover, our work provides insights into the inexploitability of certain vulnerabilities on a particular hardware, bringing a level of assurance to users of these machines.

Meltdown-type attacks. Meltdown-type attacks can be categorized according to where the secret is stole from.

- *Memory in separated address spaces.* The original Meltdown-US attack [23] leverages out-of-order execution to extract secret data from kernel-space memory. `L1TF-OS` extracts OS or SMM [11, Chapter 34] memory. `L1TF-VMM` accesses memory of another guest VM or the hypervisor from a non-privileged guest VM. The recently disclosed RIDL [35], Zombieload [30] and Fallout [29] are somewhat different from previous attacks since they leverage processor internal buffers as a source of leakage. They leak only in-flight data that are already in these buffers, but meanwhile relax the constraints of address matching.
- *Memory in the same address space.* The Foreshadow attack [34] (also called the L1TF attack [12]) steals secret data from an SGX [10] enclave from the process that creates the enclave. By clearing the present flag in the PTE of the enclave address, it induces a page fault and thus performs a Meltdown-like attack. `Meltdown-PK` [5] reads data speculatively from a memory page protected with Intel protection keys [11, Chapter 4.6.2]. `Meltdown-BR` [5] bypasses boundary check instructions: When an array in memory is accessed with an index over the bound, the boundary check instructions trigger a range exceeded exception (#BR). However, it does not prevent the out-of-order execution from accessing the address out of the boundary. Kiriansky *et al.* [16] exploits speculative writing instead of reading. Memory writes to read-only pages raise an exception but the results could still be speculatively used by following instructions.
- *Restricted registers.* LazyFP [32] exploits lazy FPU context switching to speculatively read register values used before context switches, even though such accesses trigger a device-not-available (#NM) exception. A Variant 3a disclosed by both ARM [4] and Intel [13] is also a Meltdown attack but it targets the privileged system registers such as MSR. However, during our tests, we did not find such exploitable vulnerabilities on tested machines.

Spectre-type attacks. Prior studies on Spectre-type attacks can be categorized by the exploited branch prediction units [5].

- *Prediction History Table (PHT).* The original Spectre attack [17] poisons the PHT to enable speculative reading of out-of-bound data. NetSpectre [31] extends this local attack to a remote settings. In contrast, Kiriansky *et al.* [16] demonstrated out-of-bound data writing using similar techniques. OKeeffe *et al.* [7] poisons PHT to attack SGX enclaves.
- *Branch Target Buffer (BTB).* Spectre v2 [17] targets BTB storing the branch targets of indirect branch instructions. SGXPectre [6] makes use of this variant to steal secret from SGX enclaves.
- *Return Stack Buffer (RSB).* Koruyeh *et al.* [18] and Maisuradze *et al.* [26] demonstrated the poisoning of RSB to trigger speculative side channels.
- *Store-to-Load Buffer.* Data dependency and data disambiguation related to the Store-to-Load Buffer (although not a prediction unit) were exploited by Horn [8] to perform Spectre-type attacks.

VII. Conclusion

This paper describes a software framework called SpeechMiner, which enables systematic investigation and quantitative measurement of a variety of Speech vulnerabilities on commodity processors. We have applied SpeechMiner to test the exploitability of 21 vulnerability variants on 9 processors, confirming prior disclosed vulnerabilities and also uncovering new ones. Moreover, our study explains the root causes of some observations made by prior studies and clarifies common misunderstandings, which paves the paths for future studies.

Acknowledgement

This work was funded in part by NSF through grants 718084, 1750809 and 1629392, and by Intel through the Side Channel Academic Programme (SCAP).

References

[1] "Pointer chasing," https://en.wikichip.org/wiki/pointer_chasing, 2018.

[2] AMD, "Amd processor security updates," https://www.amd.com/en/corporate/security-updates, 2018.

[3] ARM, "Arm speculation barrier header," https://github.com/ARM-software/speculation-barrier, 2018.

[4] ——, "Vulnerability of speculative processors to cache timing side-channel mechanism," https://developer.arm.com/support/security-update, 2018.

[5] 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," *arXiv preprint arXiv:1811.05441*, 2018.

[6] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[7] P.-L. A. F. K. C. P. J. L. H. Z. Dan O'Keeffe, Divya Muthukumaran and P. Pietzuch, "Spectre attack against sgx enclave," https://github.com/lsds/spectre-attack-sgx, 2018.

[8] J. Horn, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[9] Intel, "Method and apparatus for performing a store operation," US Patent, Intel Corporation, US6378062, 2002.

[10] ——, "Intel software guard extensions (intel sgx)," 2016. [Online]. Available: https://software.intel.com/en-us/sgx

[11] ——, "Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D," 2017, order Number: 325462-065US, December 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[12] ——, "Deep dive: Intel analysis of l1 terminal fault," https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, 2018.

[13] ——, "Intel analysis of speculative execution side channels," https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf, 2018.

[14] ——, "Intel 64 and IA-32 architectures optimization reference manual," 2019, order Number: 248966-041, April 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[15] ——, "Intel transactional synchronization extensions (intel tsx) overview," https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview, 2019.

[16] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[17] 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.

[18] 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)*, 2018.

[19] Linux, "Supervisor mode access prevention," 2012. [Online]. Available: https://lwn.net/Articles/517475/

[20] ——, "Memory protection keys," 2015. [Online]. Available: https://lwn.net/Articles/643797/

[21] ——, "Kaiser: hiding the kernel from user space," https://lwn.net/Articles/738975/, 2017.

[22] ——, "Meltdown strikes back: the l1 terminal fault vulnerability," https://lwn.net/Articles/762570/, 2018.

[23] 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.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 605–622.

[25] LLVM, "[patch] d41723: Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities disclosed today," http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html, 2018.

[26] 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*. ACM, 2018, pp. 2109–2122.

[27] Microsoft, "Mitigating speculative execution side channel hardware vulnerabilities," https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018.

[28] ——, "Summary of intel microcode updates," https://support.microsoft.com/en-us/help/4093836/summary-of-intel-microcode-updates, 2019.

[29] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

[30] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," *arXiv preprint arXiv:1905.05726*, 2019.

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

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

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

[34] 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 [36].

[35] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," *S&P (May 2019)*, 2019.

[36] 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 [34].

[37] H. Wong, "Measuring reorder buffer capacity," may 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

Appendix

A. *Categorized Exception List*

The categorized exception list provided below is summarized from the *Exception and Interrupt Reference* [11, Chapter 6.15]. The * and ** marks refer to one-instruction and two-instruction template respectively as explained in Sec. III-C. Notice that the original list does not include exceptions or protections from hardware extensions. In our categorized list and in the current work, we do not consider hardware extensions as well. Only some general description about the exceptions triggered by hardware extensions is included in the list below.

Page Table Based Data Protection

1) Page-Fault Exception (#PF)
   - p bit cleared *[_PTE(Present)_]
   - user mode accesses
     - access to supervisor-mode page *[_PTE(US)_]
     - write to read-only page **[_PTE(write w/ RW=0)_]
     - CR4.PKE = 1, access to user-space page forbidden access by MPK *[_Protection Key (User)_]
   - kernel mode accesses
     - CR4.SMAP = 1, access to user-space page *[_SMAP violation_]
     - CR4.PKE = 1, access to user-space page forbidden access by MPK *[_Protection Key (Kernel)_]
     - CR0.WP write to read-only page **
   - reserved bits not all cleared *[_PTE(Reserved)_]
   - An enclave access violates one of the specified access-control requirements.
2) Virtualization Exception (#VE) - EPT violations

Segmentation Based Data Protection

1) General Protection Exception (#GP) - Segment-related protection
   - Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments. *[_DS Over-Limit_]
   - Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment. **[_DS Execute-Only_]
   - Reading from an execute-only code segment. *[_CS Execute-Only_]
   - Writing to a code segment or a read-only data segment. **[_DS Read-Only_]
   - Loading the SS register with a segment selector for a read-only segment. **[_SS Read-Only_]
   - Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector. *[_DS Null_]
   - Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment. **[_SS_ $DPL \neq CPL$]
   - Transferring execution to a segment that is not executable.
   - Loading the CS register with a segment selector for a data segment or a null segment selector.
   - Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
   - Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handlers code segment DPL is greater than 0.
2) Data Segment Not Present (#NP) *[_DS Not-Present_]
3) Stack Fault Exception (#SS)
   - Limit violation when accessing ss register (eg. pop) *[_SS Over-Limit_]
   - Loading non-present stack into SS register. **[_SS Not-Present_]
   - Loading the SS register with the segment selector of an executable segment or a null segment selector. **[_SS Null_]

Program Instruction Based Data Protection

- BOUND Range Exceeded Exception (#BR) **[_BOUND_]
- Intel MPX

Other Protection

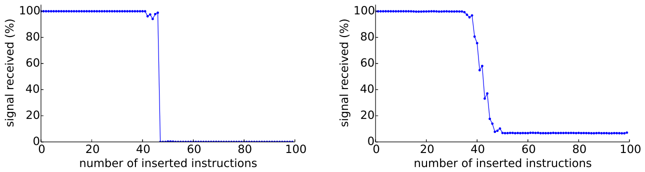
- Device Not Available Exception (#NM) - Lazy context save after context switch (CR0.TS) *[_Load xmm0 (CR0.TS)_]
- SMM memory access protection
- General Protection Exception (#GP)
  - Attempting to execute a privileged instruction when the CPL is not equal to 0. (MOV (load) control/debug registers, RDMSR) *[_Load CR4_] & *[_Load MSR (0x1a2)_]
  - Attempting to execute SGDT, SIDT, SLDT, SMSW, or STR when CR4.UMIP = 1 and the CPL is not equal to 0.
  - Attempting to execute a privileged (serializing) instruction when the CPL is not equal to 0 (LGDT, LLDT, LTR, LIDT, MOV [store] (control registers / debug registers), LMSW, CLTS, WRMSR).
  - Executing the INT n instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.

Arithmetic Protection

- Overflow Exception (#OF) - INTO instruction
- x87 FPU Floating-Point Error (#MF)
- SIMD Floating-Point Exception (#XM)
- Divide Error Exception (#DE)

Non Protection

- Debug Exception (#DB)
- Breakpoint Exception (#BP) - INT3 instruction
- Invalid Opcode Exception (#UD)
- Double Fault Exception (#DF)
- Invalid TSS Exception (#TS)
- General Protection Exception (#GP)
  - Accessing a gate that contains a null segment selector.
  - The segment selector in a call, interrupt, or trap gate does not point to a code segment.
  - The segment selector operand in the LLDT instruction is a local type (TI flag is set) or does not point to a segment descriptor of the LDT type.
  - The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
  - The target code-segment selector for a call, jump, or return is null.
  - Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
  - Instruction length limit exceeded.
  - Loading CR0 with PG=1 (paging enabled) and PE=0 (protection disabled). / Loading CR0 with NW=1 and CD=0.
  - Attempting to write a 1 into a reserved bit of CR4/MSR/MXCSR/(64-bit)CR3, CR4 or CR8.



(a) Exception-based suppression.

(b) Retpoline-based suppression

Fig. 8: Comparison between different suppression methods.

  - If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1.
  - Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
  - (64-bit) Non-canonical address / null address memory access.
  - Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.
  - An attempt is made to clear CR0.PG while IA-32e mode is enabled.

- Alignment Check Exception (#AC)
- Machine-Check Exception (#MC)
- Stack Fault Exception (#SS)
  - There is not enough stack space for allocating local variables when executing ENTER instruction.
  - (64-bit) Non-canonical address using SS register.

B. Choosing The Best Suppressing Primitive

We explore two implementations of the Suppressing Primitive: exception-based and retpoline-based. An exception-based Suppressing Primitive is illustrated in Listing 9. The other uses a retpoline to suppress exceptions [32]. We evaluated these methods using the following method: Using each method, we gradually insert ADD/SUB instructions to find the maximum speculation window. The covert-channel tests were repeated 100,000 times for each number of inserted instructions. The rate of receiving the covert-channel signals for each number is illustrated in Fig. 8a and Fig. 8b, respectively. It can be seen from the figures that the retpoline-based approach is less desirable as the rate of receiving the signal drops gradually when the inserted instructions increases, making it hard to determine the speculation window. Therefore, in our test, the exception-based approach is used.