Fallout: Reading Kernel Writes From User Space

Marina Minkin[1], Daniel Moghimi[2], Moritz Lipp[3], Michael Schwarz[3], Jo Van Bulck[4], Daniel Genkin[1], Daniel Gruss[3], Frank Piessens[4], Berk Sunar[2], Yuval Yarom[5]

[1]University of Michigan
[2]Worcester Polytechnic Institute
[3]Graz University of Technology
[4]imec-DistriNet, KU Leuven
[5]The University of Adelaide and Data61

Abstract

Recently, out-of-order execution, an important performance optimization in modern high-end processors, has been revealed to pose a significant security threat, allowing information leaks across security domains. In particular, the Meltdown attack leaks information from the operating system kernel to user space, completely eroding the security of the system. To address this and similar attacks, without incurring the performance costs of software countermeasures, Intel includes hardware-based defenses in its recent Coffee Lake R processors.

In this work, we show that the recent hardware defenses are not sufficient. Specifically, we present *Fallout*, a new transient execution attack that leaks information from a previously unexplored microarchitectural component called the *store buffer*. We show how unprivileged user processes can exploit Fallout to reconstruct privileged information recently written by the kernel. We further show how Fallout can be used to bypass kernel address space randomization.

Fallout affects all processor generations we have tested. However, we notice a worrying regression, where the newer Coffee Lake R processors are more vulnerable to Fallout than older generations.

1 Introduction

The architecture and security communities will remember 2018 as the year of Spectre [28] and Meltdown [33]. Speculative and out-of-order execution, which have been considered for decades to be harmless and valuable performance features, were discovered to have dangerous industry-wide security implications, affecting operating systems (OSs) [28, 33], browsers [1, 28], virtual machines [51], trusted execution environments (e.g., SGX) [49], AES hardware accelerators [47] and more.

Meltdown, in particular, is a severe hardware issue. In a Meltdown attack, an unprivileged attacker performs an explicit access violation to a privileged memory location containing the OS's kernel. The CPU responds with the value from that address, while marking the load operation as faulty. Perhaps most shockingly, the CPU then allows subsequent transient computation on the returned value. Finally, by the time that the CPU recognizes the violation and attempts to undo the damage caused by transient execution, the attacker already had sufficient cycles to leak the kernel data using a microarchitectural covert channel, such as via the processor's cache [8, 38].

Recognizing the danger posed by this hardware issue, the computer industry mobilized. Potentially incurring significant performance losses [12], all major OS deployed countermeasures based on the KAISER patch [14], which removes the mapping of kernel pages from the address space of user processes. At a high level, Kernel Page Table Isolation (KPTI) relies on the idea that even if the attacker can access the entire currently mapped address space, the attacker lacks the capabilities of accessing memory outside of the current address space, thus leaving the kernel safely out of reach.

Unfortunately, with Foreshadow [49] and Foreshadow-NG [51] it became clear that an attacker can transiently access even pages that are not mapped into the address space. The attacker then subsequently exploits a Meltdown-like technique

to leak privileged data, including enclave secrets safeguarded by Intel's Software Guard eXtensions (SGX) [49] or across virtual machines running on the same physical host [51].

In an attempt to claw back some of the performance loss, and to permanently eliminate Foreshadow and Meltdown related issues, Intel announced already back in 2018 strong, silicon-based Meltdown defenses in future processors enumerating Rogue Data Cache Load resilience (RDCL_NO) [22]. With the recent release of the 9th generation Coffee Lake R microarchitecture, such Meltdown-resistant processors are finally available on the mass consumer market. The RDCL_NO security feature promises to obviate the need for KPTI and other defenses, while improving overall performance [6]. However, while Intel claims that these fixes address Meltdown and Foreshadow, it remains unclear whether new generations of Intel processors are properly protected against Meltdown-type transient execution attacks. Thus, in this work we set out to investigate the following question:

Is kernel data safe in the new generation of processors? Can ad-hoc software mitigations be safely disabled on post-Meltdown Intel hardware?

1.1 Our Contribution

Unfortunately, in this paper, we answer these questions in the negative. We present *Fallout*, a new attack on the hardware-based memory isolation mechanisms in Intel CPUs. Using Fallout, user-space programs can read data that has recently been written by the kernel, as well as derandomize Kernel Address Space Layout Randomization (KASLR). Similarly to previous transient execution attacks, Fallout does not require any privileges except for the ability to run code, and does not exploit any kernel vulnerabilities.

The Mechanism Behind Fallout. Fallout exploits an optimization that we call *Write Transient Forwarding* (WTF), which incorrectly passes values from memory writes to subsequent memory reads. In a nutshell, when the program writes a value to memory, the processor needs to first translate the virtual address of the destination to a physical address and then acquire exclusive access to the location. Rather than stalling the store instruction and subsequent computation, the processor records the value and the address in the *store buffer*, and continues executing the program. The store buffer then resolves the address, acquires the access to the memory location and stores the data.

When a value is in the store buffer, care should be taken that subsequent loads from the same address do not read stale values from memory. To solve this, the processor matches the addresses of all load instructions against addresses in the store buffer. In the case of a match, the processor *forwards* the matching value from the store buffer to the load instruction. To increase efficiency, the processor uses partial address matches to rule out the need for store-to-load forwarding. WTF kicks in when a load instruction partially matches a preceding store and the processor determines that the load is bound to fail. In such cases, instead of cleaning up the state of the processor, it marks the load as faulty, and *incorrectly* forwards the value of the partially matched store.

Exploiting the WTF optimization. Fallout exploits this behavior to leak, through a microarchitectural channel, the value that WTF incorrectly forwards. The attacker deliberately performs a faulty load, causing the CPU to transiently forward an incorrect value from the store buffer. We subsequently leak the value using a Flush+Reload [52] side channel. As the store buffer is a shared resource used by all software running on a CPU core, the incorrectly-forwarded value might not even belong to the attacker's process. Empirically demonstrating this, in this paper, we show how to exploit the WTF optimization to leak values recently written by the kernel from user space as well as how to derandomize the kernel's ASLR.

Fallout vs. Meltdown Like all Meltdown-type attacks, Fallout exploits transient execution past an exception. However, unlike previous Meltdown-type attacks, in Fallout the adversary does not read from the address of the protected value. Instead, the value leaks while the adversary loads from an unrelated memory address. As a result, the hardware countermeasures for Meltdown and Foreshadow in recent Intel processors do not protect against Fallout. Finally, we note a worrying regression in recent Intel processors, where, possibly due to the added hardware countermeasures, newer processors seem more vulnerable to Fallout than previous generations.

Security Analysis of Speculation Mechanisms and Coffee Lake Refresh. As a final contribution, we present the first analysis of various exception-creation and exception-suppression mechanisms used to mount Fallout across various Intel architectures. As we show, not all creation and suppres-

sion mechanisms are interchangeable, and the exact combination is, in fact, architecture dependent. Finally, we show that the hardware change in exception creation and suppression introduced by Intel in the latest Coffee Lake Refresh architecture make them more vulnerable to our attack.

1.2 Disclosure and Timeline

Following the practice of responsible disclosure, we have notified CPU vendors about our findings.

Intel. We notified Intel about our findings, including a preliminary writeup and proof-of-concept code, on January 31st, 2019. Intel had acknowledged the issue and requested an embargo on the results in this paper, ending May 14th, 2019. Intel has further classified this issue as Microarchitectural Store Buffer Data Sampling (MSBDS), assigning it `CVE-2018-12126` and a CVSS ranking of Medium. Finally, Intel had indicated that we are the first academic group to report this issue and that a similar issue was found internally as well.

AMD. We also notified AMD's security response team regarding our findings, including our writeup. AMD had investigated this issue of their architectures and indicated that AMD CPUs are not vulnerable to the attacks described in this paper.

ARM. We are in the process of identifying a trusted contact at ARM security in order to share our findings with them. We expect to achieve that within days of the submission deadline.

IBM. Finally, we also notified IBM security about the finding reported in this work. IBM had responded that none of their CPUs is affected, including System-V and PowerPC.

The RIDL Attack. In a concurrent independent work[1], the RIDL attack [50] analyzes additional buffers present inside Intel CPUs, with specific attention to the Line Fill Buffer (LFB) and load ports. There, they show that faulty loads from the LFB or Load Ports leak information across various security domains. We note however that Fallout is different from (and complementary to) RIDL. This is since the two attacks exploit different microarchitectural elements (LFB and load ports for RIDL and Store Buffer and WTF optimization for Fallout). In particular, RIDL can be used to recover values recently placed in the LFB while Fallout allows the attacker to recover the value of a specific attacker-chosen writes in the store buffer.

[1] Both teams made contact on May 7th, provided each other with an overview of their findings, and coordinated public disclosure as well as communication with Intel. For a complete timeline describing the flow of information related to this disclosure, see `mdsattacks.com`.

2 Background

In this section, we provide the background required to understand our attack, including a description of caches and cache attacks, transient execution attacks, and Intel Transactional Synchronization Extensions.

2.1 Caches and Cache Attacks

Caches are an essential part of modern processors. They are small and fast memories where the CPU stores copies of data from the main memory to hide the main memory access latency. Modern CPUs have a variety of different caches and buffers for various purposes. The main cache hierarchy is the instruction and data cache hierarchy consisting of multiple levels, which vary in size and latency. The L1 is the smallest and fastest cache. The L3 cache, also called the last-level cache (LLC), is typically the largest and slowest.

Cache Organization. Modern caches are typically set-associative, i.e., a cache line is stored in a fixed set, as determined by part of its virtual or physical address. Addresses that map to the same set are called *congruent*. On modern processors, the last-level cache is typically physically indexed and shared across cores. It is also often inclusive of L1 and L2, which means that all data stored in L1 and L2 is also stored in the last-level cache. The cache hierarchy exposes the latency difference between the main memory access (cache miss) and the cache access (cache hit), i.e., exactly the latency difference that caches introduce. This can be used in side channels on a non-colluding victim or in covert channels where sender and receiver collude to transmit information.

Cache Attacks. Different cache attack techniques have been proposed in the past, such as Prime+Probe [40, 41] and Flush+Reload [52]. Flush+Reload attacks and its variants [16, 17, 32, 54] work on shared memory at a cache-line granularity. The attacker repeatedly flushes a cache line and measures how long it takes to reload it. The reload time will always be high unless another process has reloaded the cache line back into the cache. In contrast, Prime+Probe attacks work without shared memory, and only at a cache-set granularity. The attacker repeatedly accesses a set of congruent memory addresses, filling an

entire cache set with its own cache lines, and measures how long that takes. As this is repeated in a loop, the cache set is always filled with the attacker's cache lines. Hence the access time will always be rather low. However, if another process accesses a memory location in the same cache set, it will evict one of the attacker's cache lines and the access time will increase.

Cache attacks have been used to break cryptographic implementations [9, 10, 34, 40, 41, 52, 53], infer user input [17, 32, 42], and break system-level security [15, 20]. Both Prime+Probe and Flush+Reload have also been used in high-performance covert channels [16, 34, 38], also as a building block of transient execution attacks such as Meltdown [33], Spectre [28], and Foreshadow [49, 51] that we detail below.

2.2 Superscalar Processors

To achieve their high performance, modern processors are often *superscalar*, that is, they perform multiple operations in parallel. In current implementations, e.g., in modern Intel processors (refer Fig. 1), execution of a program is divided between two main parts. The *frontend* is responsible for processing the machine-code instructions of the program, decoding them to a stream of *micro-ops* (μOPs) that are sent to the *Execution Engine* for execution.

Out-of-order Execution. The execution engine consists of multiple execution units, which can execute various μOPs. To allow superscalar execution, the execution engine follows a variant of Tomasulo's algorithm [48], which executes μOPs when the data they depend on is available, rather than following strict program order. Once executed, the μOPs arrive at the *reorder buffer* whose purpose is to *retire* μOPs in program order, ensuring that architecturally-visible effects of μOPs execute in the order the programmer specified.

Speculative Execution. The stream of μOPs that the frontend generates does not necessarily correspond to the sequence of instructions in the program. A major cause of deviation is *branch prediction*. When the frontend reaches a branch instruction, it often does not yet know where execution will proceed. Instead of waiting, the frontend attempts to predict the outcome of the branch and proceed from there. In the case that the prediction is correct, the generated μOPs match the program and can be processed. Otherwise, at some later stage, the processor notices the *misprediction*. The frontend is then *steered* to the correct instruction, and μOPs generated as part of the misprediction are dropped by the reorder buffer without committing any of their results to the architectural state of the processor. Following Canella et al. [5], we refer to μOPs that are not retired as *transient*. Similarly, following Glew et al. [11], we use refer to μOPs other than the one waiting for retirement as *speculative*. We note that speculative μOPs do not necessarily result from speculative execution. The are called speculative because the execution engine cannot determine whether they are transient or not.

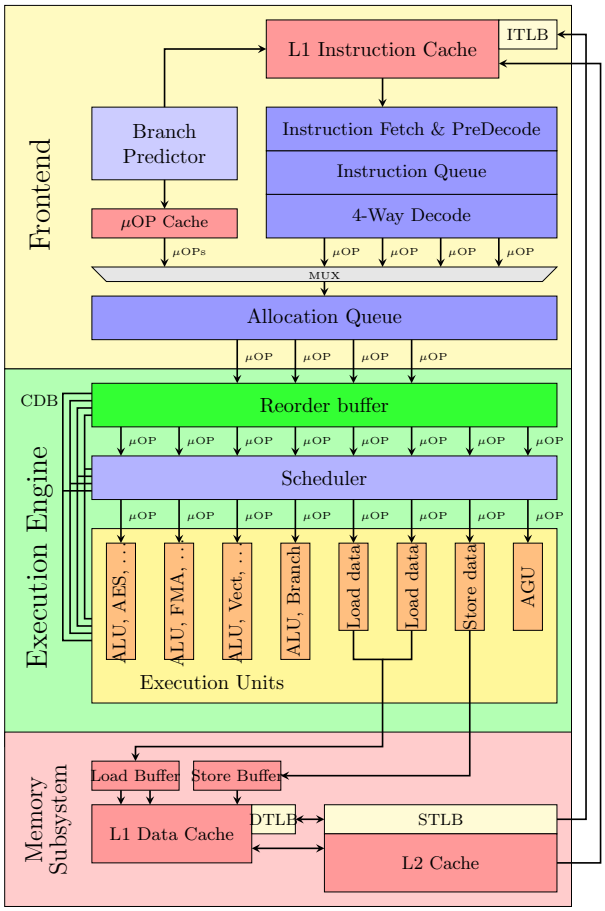

Figure 1: Simplified illustration of a single core of the Intel's Skylake microarchitecture (as presented in [33]). Instructions are decoded into μOPs and executed out-of-order in the execution engine by individual execution units.

2.3 The Memory Subsystem

In this work, we are mainly interested in how memory load and store operations are implemented. The main two issues we deal with are how to resolve the physical addresses used by these instructions and how to ensure that out-of-order execution does not break dependencies between these instructions.

2.4 Transient Execution Attacks

While transient execution does not influence the architectural state of the processor, it can change the microarchitectural state. Transient execution attacks abuse transient execution to execute a few instructions transiently and modify the microarchitectural state. The change in the microarchitectural state is then observed using a covert-channel attack. Spectre-type [28] attacks exploit different prediction mechanisms, while Meltdown-type [33, 49] attacks exploit transient execution following a CPU exception.

Spectre Attacks. The first Spectre attacks focused on the CPU's Pattern History Table (PHT), Branch History Buffer (BHB), and Branch Target Buffer (BTB) as microarchitectural data structures causing mispredictions [28]. Both transient loads and stores [27] are possible, leading to a variety of attacks, including reading and writing from out-of-bound memory locations, transferring control-flow to arbitrary addresses via mispredicted indirect jumps [28] or returns [29, 36]. In all Spectre attacks, the attacker mistrains the processor by performing a certain type of branches, influencing the corresponding microarchitectural predictor. Subsequently, the victim runs with incorrect predictions and thereby leaks data. While Spectre attacks can only leak architecturally accessible data, the mistraining works across privilege boundaries, e.g., the kernel-to-user boundary, or SGX. Another type of Spectre attacks is based on unsuccessful load-to-store forwarding [19]. Spectre attacks can even be mounted in remote scenarios, i.e., from JavaScript [28] or just by sending requests to a vulnerable system [43].

Meltdown Attacks. Meltdown-type attacks do not exploit misprediction. Instead, they exploit deferred handling of permission checks. Before the permission check is performed and the attacker process triggers a processor exception architecturally, the data is already handed to the subsequent instructions that are also transiently executed. The first Meltdown attack [33] exploits the deferred permission check for the user/supervisor bit in the page tables, allowing to leak arbitrary memory mapped in the kernel address space. Other Meltdown attacks similarly exploit the deferred check of present or reserved bits in page table entries [49, 51], the writable bit in the page table entry [27], or the permission check when reading system registers [4, 21].

Countermeasures. Recognizing the danger posed by transient execution attacks, a wide range of defenses have been proposed to defend against them. However, to date, it is unclear which defenses actually increase the security level and which are trivially bypassable [5, 39]. One defense where the consensus across academia and industry is that it protects against Meltdown, if correctly implemented, is KAISER [14]. KAISER is the idea of duplicating the page table hierarchies for every process, once with the kernel space mappings present and once without. When running in user space the mapping without the kernel space is used. The idea of KAISER has been integrated into all major operating systems, e.g., in Linux as KPTI [35], in Windows as KVA Shadow [24], and in Apple's xnu kernel as double map [31]. While KAISER costs performance, the use of PCID and ASID on modern processors reduced the overheads for real-world workloads to almost zero [12]. More recent processors ship with hardware patches and hence have the KAISER patch disabled by default [6].

2.5 Exception Creation

As explained in Section 2.4, in a Meltdown-type attack the attacker exploits the deferred enforcement of permissions (i.e., deferred exception handling) present in Intel CPUs in order to obtain privileged information. In the original Meltdown attack [33], the attacker exploits the delayed enforcement of the User / Supervisor bit in the CPU's hardware in order to read privileged information and subsequently leak it through a covert channel. Next, in Foreshadow [49] and Foreshadow-NG [51], the attacker exploits the fault cases of a page marked as non-present and therefore cannot be accessed.

2.6 Exception Suppression

One problem common to Meltdown-type attacks is that the instructions they exploit cause exceptions, which by default terminate the program. Four main approaches have been suggested for handling this termination. In the fork-and-crash approach, a forked process executes the attack, and its parent resumes after the process terminates. Exception handling sets

up a signal handler to catch the exception and resume execution. A third option suppresses the exception by wrapping the attack code in a mispredicted branch or call, which speculatively executes the attack. Finally, the exception can be suppressed by wrapping it in a hardware transaction. The last approach is the most effective [33] and most widely applicable [49, 51]. Given its applicability, in Section 2.7 below, we provide additional details about exception suppressing using hardware transactions. We refer interested readers to Lipp et al. [33] for further information on the other approaches.

2.7 Transactional Memory

Intel's Transactional Synchronization Extensions (TSX) is an instruction set extension to the x86-64 architecture that supports hardware transactions. In a nutshell, a transaction is a sequence of instructions that are either executed atomically or not executed at all. Atomic execution implies that concurrent threads cannot observe intermediate updates from the transaction and the thread executing the transaction cannot observe any changes from other threads.

Implementing TSX Transactions. Transactions are delimited by two instructions. The XBEGIN instruction starts a transaction and XEND terminates it. The XBEGIN instruction also specifies an abort location where execution continues if the transaction fails. Transaction implementation mostly relies on existing processor mechanisms. Instructions following XBEGIN are not retired and instead are kept in the reorder buffer until the XEND is executed. If the transaction is aborted, all pending instructions in the transaction are discarded, and the architectural state of the processor is reverted to the state before the XBEGIN. To revert memory state and to maintain atomicity, memory stores inside a transaction modify the L1 cache but are not evicted to lower memory layers, and memory lines read in a transaction remain in the last-level cache. TSX locks the affected lines to protect against concurrent modifications and reads of modified lines.

Transaction Aborts. If concurrent processes try to write to these locked lines, the transaction aborts and is rolled back. Similarly, if the processor runs out of cache space for the transaction data, the transaction aborts. This behavior of TSX transactions has been exploited for both side-channel attacks and defenses [7, 13, 45]. Transactions also abort in other scenarios. In particular, transactions abort when the processor receives an exception or if an instruction within the transaction causes a fault. Thus, when a Meltdown-type attack is enclosed in a TSX transaction, the faulting instruction causes a transaction abort, which effectively reverses the architectural state of the processor to the state prior the XBEGIN instruction, suppressing the fault. Yet, as Lipp et al. [33] observe, the microarchitectural state of the processor is not reverted when a transaction aborts, allowing the attacker to recover information from the aborted instructions.

3 The Write Transient Forwarding Optimization

In this section, we discuss the WTF optimization that is exploited with the Fallout attack. First, we will illustrate the basic idea of Fallout with a simple toy example before discussing the hardware mechanisms responsible for the attack.

3.1 A Toy Example

Listing 1 shows a simple code snippet which exploits the WTF optimization to read variables without directly accessing them. While this example does not have security implications on its own, it nonetheless shows the general concept behind Fallout, allowing user-level code to read information stored in the CPU's store buffer without directly accessing the address corresponding to that information.

Setup. First, 2 pages are allocated. The `victim_page` is a user space accessible page where the user can store and read data. However, by setting the protection level to `PROT_NONE` on the `attacker_page`, all access permissions to this page are revoked and the page is marked as *not-present*. Thus, any access to the `attacker_page` will yield an exception.

Next, we write the value 42 to the offset 7 of the `victim_page`. Rather than executing the write to memory immediately, the processor first notes the operation in the store buffer. We note that the code in Listing 1 never reads from the `victim_page` directly.

Reading Previous Stores. Instead of reading from the victim page at the specified offset, the code starts a TSX transaction (Line 8) and reads from the `attacker_page`. As the page is inaccessible, the memory access will fail and the TSX transaction aborts. However, the exception will be only handled by the reorder buffer when the memory access operation is retired. In the meantime, due to the

```
1  char* victim_page = mmap(..., PAGE_SIZE,
       ...);
2  char* attacker_page = mmap(..., PAGE_SIZE
       , ...);
3  mprotect(attacker_page, PAGE_SIZE,
       PROT_NONE);
4
5  offset = 7;
6  victim_page[offset] = 42;
7
8  if (tsx_begin() == 0) {
9    memory_access(lut + 4096 *
         attacker_page[offset]);
10   tsx_end();
11 }
12
13 for (i = 0; i < 256; i++) {
14   if (flush_reload(lut + i * 4096)) {
15     report(i);
16   }
17 }
```

Listing 1: Pseudocode of Fallout. Some mmap parameters were omitted for clarity

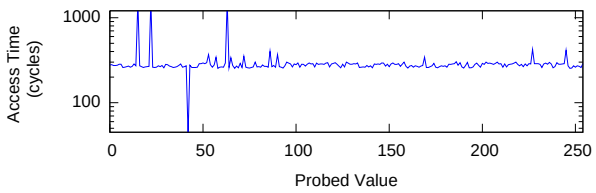

Figure 2: Access times to the probing array during the execution of Listing 1. The dip at 42 represents a correct recovery of the value from the store buffer.

WTF optimization, the CPU will transiently forward the value of the previous store at the same page offset. Thus, the memory access will pick-up the value of the store to the `victim_page`, in this example 42. Using a cache-based covert channel, the incorrectly forwarded value is transmitted. Finally, when the failure and transaction abort are handled, the architectural effects of the transiently executed code are reverted.

Recovering the Leaked Data. Using Flush+Reload, the attacker can recover the leaked value from the cache-based covert channel in Line 14. Fig. 2 displays the results of measured access times to the look-up-table (`lut`) on a Meltdown-resistant i9-9900K CPU. As the figure illustrates, the typical access time to an array element is above 200 cycles, with the exception of element 42, where the access time is well below 100 cycles. We note that this position matches the value written to `target_page`. Hence, the code can recover the value without directly reading it.

3.2 The Mechanism Behind Fallout

We now turn our attention to the *store buffer*, a microarchitectural component, which lies in the core of WTF and Fallout.

The Store Buffer Implementation. When the CPU writes data to memory, it needs to first resolve the virtual address to a physical address. Then it acquires exclusive access to the cache line of the target data. Rather than waiting, the processor stores the information to the store buffer.

Fig. 3 shows the structure of the store buffer according to Intel patents [2, 3]. Based on these patents, a store operation is implemented using two μOPs, store address (STA) and store data (SDA). Splitting the operation to two μOPs allows the processor to process the parts independently and asynchronously.

Asynchronous processing raises the issue of memory ordering. Specifically, operations that access the same memory locations must be performed at the order specified in the program and, in particular, load operations should get the value from preceding stores to the same address. Intel published some properties of the store buffer [23]. However, we are not aware of any public documentation of the algorithms used for resolving memory access conflicts. Intel's patents on the topic [2, 3, 30] suggest that the store buffer is virtually indexed, but each entry also includes parts of the physical address, such that mismatches on the partial addresses ensure the absence of dependencies, allowing loads to proceed without waiting for full address resolution.

Write Transient Forwarding. An algorithm for handling partial address matches appears in another Intel patent [18]. Remarkably, the patent explicitly states that:

> "if there is a hit at operation 302 [partial match using page offsets] and the physical address of the load or the store operations is not valid, the physical address check at operation 310 [full physical address match] may be considered as a hit"

That is, if address translation of a load μOP fails and the 12 least significant bits of the load address match those of a prior store, the processor assumes

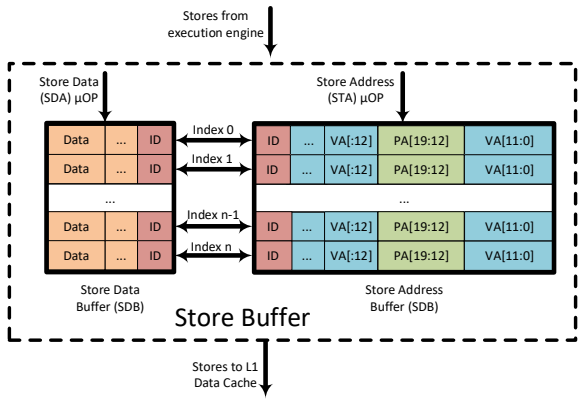

Figure 3: Structure of the store buffer on Intel CPUs.

that the physical addresses of the load and the store match and forwards the previously stored value to the load μOP. We note that the failed load is transient and will not retire, hence WTF has no architectural implications. However, as this work demonstrates, microarchitectural side effects of transient execution following the failed load may result in inadvertent information leaks. Given the surprising nature of this optimization and its security consequence, we refer to it as the Write Transient Forwarding (WTF) optimization.

Fault and Suppression Mechanisms. To better understand the WTF mechanism, we evaluate the toy example in Listing 1 with multiple combinations of causes of faults and fault-suppression mechanisms. We experimented with three Intel processors: a Coffee Lake R i9-9900K, a Kaby Lake i7-7600U, and a Skylake i7-6700. We summarize the results in Table 1.

We observe that unlike earlier generations, the Coffee Lake R processor exhibits a different behavior based on the fault suppression mechanism. Specifically, in the example in Listing 1 replacing the TSX fault suppression mechanism with branch misprediction does not trigger the WTF optimization, and the value does not leak. We suspect that the processor inhibits some forms of speculative execution within branch misprediction while allowing it in TSX transactions. Moreover, the Coffee Lake R processor does not seem to trigger the WTF optimization when a load fails due to a read from a kernel page. We note that transient reads from such pages is the main cause of the Meltdown bug. Thus, we conjecture that the differences in behavior between the processor generations are due to the recent mitigations for the Meltdown and Foreshadow attacks introduced in the Coffee Lake R architecture.

Coffee Lake R Regression. We also note a troubling *regression* in Intel's newest architecture. When accessing a page marked as non-present, we can only trigger the WTF optimization on the Coffee Lake Refresh processor.

3.3 Measuring the Store Buffer Size

We now turn our attention to measuring the size of the store buffer. Intel advertises that Skylake processors have 56 entries in the store buffer [37]. We could not find any publications specifying the size of the store buffer in newer processors, but as both Kaby Lake and Coffee Lake R are not major architectures, we assume that the size of the store buffers has not changed. As a final experiment in this section, we now attempt to use Fallout to confirm this assumption. To that aim, we perform a sequence of store operations, each to a different address. We then use a faulty load aiming to trigger a WTF optimization and retrieve the value stored in the first (oldest) store instruction. For each number of stores, we attempt 100 times at each of the 4096 page offsets, to a total of 409,600 per number of stores. Fig. 4 shows the likelihood of triggering the WTF optimization as a function of the number of stores for each of the processor and configurations we tried. We see that we can trigger the WTF optimization provided that the sequence has up to 55 stores. This number matches the known data for Skylake and confirms our assumption that it has not changed in the newer processors.

The figure further shows that merely enabling hyperthreading does not change the store buffer capacity available to the process. However, running code on the second hyperthread of a core halves the available capacity, even if the code does not perform any store. This confirms that the store buffers are statically partitioned between the hyperthreads [23], and also shows that partitioning takes effect only when both hyperthreads are active.

4 Using Fallout to Break Kernel Isolation

In this section, we show that Fallout can leak information from the OS kernel to unprivileged users. Our proof-of-concept implementation consists of two components. The first is a kernel module that writes to a

Fault Suppression	Transactional Memory (TSX)		Branch Misprediction	
Architecture	Pre Coffee Lake R	Coffee Lake R	Pre Coffee Lake R	Coffee Lake R
User not present	✗	✓	✗	✗
Kernel data	✓	✗	✓	✗
Kernel code	✓	✓	✓	✓
Unmapped kernel	✗	✗	✗	✗

Table 1: Evaluating different fault-inducing and fault-suppression mechanisms on Intel architectures before Coffee Lake R and on Coffee Lake R. ✓ indicates that our attack can successfully leak data, while ✗ indicates no leakage was observed. Finally, we denote the case of the Coffee Lake R regression with ✓, while changes following hardware countermeasures are marked with ✗.

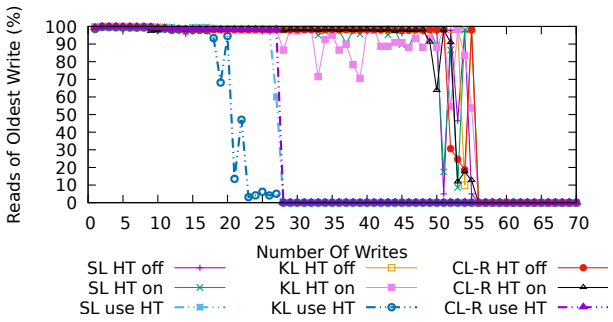

Figure 4: Measuring the size of the store buffer on Kaby Lake and Coffee Lake machines. In the experiment, we perform multiple writes to the store buffer and subsequently measure the probability of retrieving the value of the first (oldest) store. The results agree with 56 entries in the store buffer and with a static partitioning between hyperthreads.

predetermined virtual address in a kernel page. The second is a user application that performs a faulty load from an address in a user page, such that the page offset of this address the same as the page offset the kernel module writes to. Exploiting the WTF optimization, the user application can retrieve the data written by the kernel. We now proceed to describe both parts of our proof-of-concept implementation.

The Kernel Module. Our kernel module performs a sequence of write operations each to a different page offset in a different kernel page. These pages, like other kernel pages, are not directly accessible to user code. On older processors, such addresses may be accessible indirectly via Meltdown. However, we do not exploit this and assume that the user code does not or cannot exploit Meltdown.

The Attacker Application. The attacker application aims to retrieve kernel information that would normally be inaccessible outside the kernel. The attacker code first uses `mprotect` to revoke access to a page. It then invokes the kernel module to perform the kernel writes. When the kernel module returns, the attacker performs a faulty load from the protected page, before transiently leaking the value through a covert cache channel.

Increasing the Window for the Faulty Load. To increase the time window for the faulty load, our attacker code further delays processing the kernel store by performing a sequence of store operations before invoking the kernel module. Store buffer entries are processed and stored in the cache in program order [2, 3, 18, 25]. Thus, filling the store buffer delays processing of later stores. We further increase the effect of these store operations by first flushing the addresses they write to from the cache.

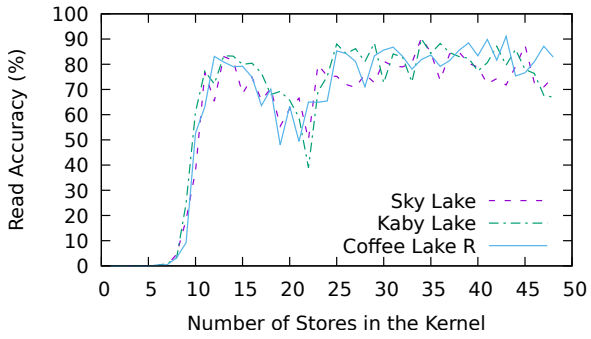

Figure 5: Probability of recovering kernel values from user space as a function of the number of kernel stores.

Experimental Evaluation. We measure the number of stores that the kernel needs to perform for Fallout to be able to recover a value it stores before returning user space. We use our three Intel machines with a fully updated Ubuntu 16.04, keeping the kernel mapped in the process's address space. Fig. 5 shows

the results of our evaluation, where each experiment is repeated 409,600 times, 100 at each possible page offset. As the figure shows, after about 10 kernel writes the attacker can use Fallout to recover a value written by the kernel on both machines with about 80% probability.

On processors vulnerable to Meltdown, leaving the kernel mapped in the process's address space disables KPTI, allowing Meltdown attacks on the kernel. For the Coffee Lake R processor, which includes hardware countermeasures for Meltdown, KPTI is disabled by default. In particular, the experiments for this processor in Fig. 5 are with the default Ubuntu configuration. Ironically, this means that the hardware countermeasures in Intel's latest CPU generations make them more vulnerable to Fallout.

5 Using Fallout to Break KASLR

We now show how Fallout can be used to break Kernel Address Space Layout Randomization (KASLR).

5.1 KASLR Background

Code injection attacks are a type of vulnerability where the attacker injects code to the address space of the victim and subsequently diverts the victim's control flow to execute the injected code. A common protection for such attacks is to adopt a policy where memory pages are either writable or executable, but never both.

ROP and Return-to-Libc Attacks. Return-to-libc [46] and return oriented programming (ROP) [44] are two related techniques that reuse existing code for exploiting memory corruption vulnerabilities. In a nutshell, by overwriting the stack, the attacker can hijack the control flow, and direct execution into *gadgets* that exist in the victim's code or in linked libraries. Shacham [44] demonstrates that a typical library contains enough gadgets that, when threaded, can perform arbitrary computation.

ASLR. Address Space Layout Randomization (ASLR) is a probabilistic countermeasure for ROP. The main idea is to introduce randomness the in the victim memory layout, hiding it from the attacker. That is, when a process is initialized, ASLR randomizes the locations of the code and the data (see Fig. 6 (top)). With ASLR, the attacker needs to find the addresses of code gadgets to be able to use them.

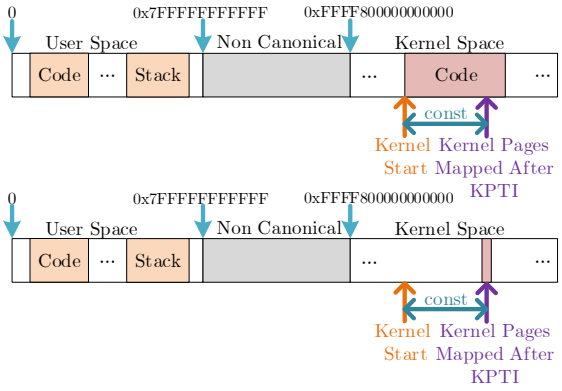

Figure 6: (Top) Address space layout with KASLR but without KPTI. (Bottom) User address space with KASLR and KPTI. Most of the kernel is not mapped in the process's address space anymore.

KASLR on Linux Systems. On Linux systems, KASLR had been supported since kernel version 3.14 and enabled by default since around 2015. As Jang et al. [26] note, the amount of entropy present depends on the kernel address range as well as on the alignment size which is usually multiple of page size.

KASLR and KPTI. As a countermeasure to the Meltdown attack [33], OSs running on Intel processors up to the latest Coffee Lake architecture have deployed the Kernel Page Table Isolation (KPTI) mechanism, which removes the kernel from the address space of user processes (see Fig. 6 (bottom)). To allow the process to switch to the kernel address space, the system leaves at least one kernel page in the address space of the user process. Because the pages required for the switch do not contain any secret information, there is no need to hide it from Meltdown.

The KPTI patch is based on KAISER [14], which was originally designed to protect the kernel from side-channel attacks that break KASLR [15, 20, 26]. We now proceed to show that Fallout can reveal the location of the kernel entry page left in the user address space, thereby breaking KASLR.

5.2 Using Fallout to Break Kernel ASLR

Attack Overview. Our attack is based on the disparity between the effects of causes of faults (see Table 1). Specifically, we note that when accessing an unmapped kernel page, the WTF optimization is not triggered and the Fallout attack fails. Thus, to perform the attack, we replace the read from

`attacker_page` in Line 9 with a read from a page within the kernel address range. When the page we access is mapped, Fallout succeeds and we retrieve a value from the store buffer. Otherwise no value is retrieved from the store buffer.

Experimental Setup. We evaluate Fallout on two Intel machines, a Kaby Lake i7-7600U and a Coffee Lake R i9-9900K. Both machines run a fully updated Ubuntu 16.04 system, with all countermeasures in their default configuration. On both systems, we empirically test the possible locations on the kernel in its address space obtaining about 490 locations, implying about 9 bits of entropy.

Experimental Results. We run the attack 1000 times each, on both the Kaby Lake and the Coffee Lake machines. Our attack can recover the kernel location with 100% accuracy on both machines, within about 0.27 seconds.

6 Conclusions and Future Work

Flushing-Based Countermeasures. Because the store buffer is not shared across hyperthreads, leaks can only occur when the security domain changes within a hyperthread. Thus, flushing the store buffer on security domain change is sufficient to mitigate the attack. In particular, we verified that using MFENCE as part of the switch from kernel mode to user mode thwarts the attack.

Limitations. As mentioned above, the attacks described in Section 4 are unable to leak information across hyperthreads . Moreover, as Meltdown software countermeasures (KPTI) flush the buffer on leaving the kernel, and as the store buffer is automatically flushed on change of the `CR3` register (i.e., on context switch), only latest generation Coffee Lake R machines are vulnerable to the attack described in Section 4. Ironically, the hardware mitigations present in newer generation Coffee Lake R machines make them more vulnerable to Fallout than older generation hardware.

Acknowledgments

This research was supported in part by Intel Corporation. The research presented in this paper was partially supported by the Research Fund KU Leuven. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO). The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] 2018. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass. (2018). [Online; accessed 30-January-2019].

[2] Jeffery M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, and Paul D Madland. 2002. Method and apparatus for performing a store operation. US Patent 6,378,062. (April 23 2002).

[3] Jeffrey M Abramson, Haitham Akkary, Andrew F Glew, Glenn J Hinton, Kris G Konigsfeld, Paul D Madland, David B Papworth, and Michael A Fetterman. 1998. Method and Apparatus for Dispatching and Executing a Load Operation to Memory. US Patent 5,717,882. (Feb. 10 1998).

[4] ARM Limited. 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. (2018). https://developer.arm.com/support/security-update

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441* (2018).

[6] Ian Cutress. 2018. Analyzing Core i9-9900K Performance with Spectre and Meltdown Hard-

ware Mitigations. https://www.anandtech.com/show/13659/analyzing-core-i9-9900k-performance-with-spectre-and-meltdown-hardware-mitigations. (2018). [Online; accessed 30-January-2019].

[7] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security*. 51–67.

[8] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptographic Engineering* 8, 1 (2018), 1–27.

[9] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-by Key-extraction Cache Attacks from Portable Code. In *ACNS*. 83–102.

[10] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth be with you: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *CCS*. 845–858.

[11] Andy Glew, Glenn Hinton, and Akkary Haitham. 1997. Method and Apparatus for Performing Page Table Walks in a Microprocessor Capable of Processing Speculative Instructions. US Patent 5,680,565. (1997).

[12] Brendan Gregg. 2018. KPTI/KAISER Meltdown Initial Performance Regressions. (2018). http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html

[13] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*. 217–233.

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *ESSoS*.

[15] Daniel Gruss, Clmentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.

[17] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.

[18] Sebastien Hily, Zhongying Zhang, and Per Hammarlund. 2009. Resolving False Dependencies of Speculative Load Instructions. US Patent 7.603,527. (2009).

[19] Jann Horn. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. (2018). https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*.

[21] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. (July 2018). https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf

[22] Intel. 2018. Speculative Execution Side Channel Mitigations. (May 2018). Revision 3.0.

[23] Intel Corporation 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

[24] Alex Ionescu. 2017. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER). (2017). https://twitter.com/aionescu/status/930412525111296000

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *arXiv preprint arXiv:1903.00446* (2019).

[26] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*. 380–392.

[27] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[28] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[29] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.

[30] Steffen Kosinski, Fernando Latorre, Niranjan Cooray, Stanislav Shwartsman, Ethan Kalifon, Varun Mohandru, Pedro Lopez, Tom Aviram-Rosenfeld, Jaroslav Topp, and Li-Gao Zei. 2012. Store Forwarding for Data Caches. US Patent 9,507,725. (2012).

[31] Jonathan Levin. 2012. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons.

[32] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.

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

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[35] LWN. 2017. The Current State of Kernel Page-Table Isolation. (Dec. 2017). https://lwn.net/Articles/741878/

[36] Giorgi Maisuradze and Cihristian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.

[37] Julius Mandelblat. Technology Insight: Intels Next Generation Microarchitecture Code Name Skylake. In *Intel Developer Forum (IDF15)*. https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf.

[38] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Rmer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[39] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is Here to Stay: An Analysis of Side-Channels and Speculative Execution. *arXiv preprint arXiv:1902.05178* (2019).

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.

[41] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.

[42] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clmentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[43] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

[44] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *CCS*. 552–561.

[45] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*.

[46] Solar Designer. 1997. Getting around non-executable stack (and fix). Bugtraq mailing list. (Aug. 1997).

[47] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480* (2018).

[48] Robert M Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[49] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[50] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *S&P*.

[51] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. https://foreshadowattack.eu/foreshadow-NG.pdf. (2018).

[52] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*. 22–25.

[53] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptographic Engineering* 7, 2 (2017), 99–112.

[54] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS*. 858–870.