2019 IEEE Symposium on Security and Privacy

RIDL: Rogue In-Flight Data Load

Stephan van Schaik*, Alyssa Milburn*, Sebastian Österlund*, Pietro Frigo*, Giorgi Maisuradze†‡, Kaveh Razavi*, Herbert Bos*, and Cristiano Giuffrida*

*Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands
{s.j.r.van.schaik, a.a.milburn, s.osterlund, p.frigo}@vu.nl,
{kaveh, herbertb, giuffrida}@cs.vu.nl

†CISPA Helmholtz Center for Information Security
Saarland Informatics Campus
giorgi.maisuradze@cispa.saarland

***Abstract*—We present *Rogue In-flight Data Load* (RIDL), a new class of speculative unprivileged and constrained attacks to leak arbitrary data across address spaces and privilege boundaries (e.g., process, kernel, SGX, and even CPU-internal operations). Our reverse engineering efforts show such vulnerabilities originate from a variety of micro-optimizations pervasive in commodity (Intel) processors, which cause the CPU to speculatively serve loads using extraneous CPU-internal *in-flight* data (e.g., in the *line fill buffers*). Contrary to other state-of-the-art speculative execution attacks, such as Spectre, Meltdown and Foreshadow, RIDL can leak this arbitrary in-flight data with no assumptions on the state of the caches or translation data structures controlled by privileged software.**

The implications are worrisome. First, RIDL attacks can be implemented even from linear execution with no invalid page faults, eliminating the need for exception suppression mechanisms and enabling system-wide attacks from arbitrary unprivileged code (including JavaScript in the browser). To exemplify such attacks, we build a number of practical exploits that leak sensitive information from victim processes, virtual machines, kernel, SGX and CPU-internal components. Second, and perhaps more importantly, RIDL bypasses all existing "spot" mitigations in software (e.g., KPTI, PTE inversion) and hardware (e.g., speculative store bypass disable) and cannot easily be mitigated even by more heavyweight defenses (e.g., L1D flushing or disabling SMT). RIDL questions the sustainability of a per-variant, spot mitigation strategy and suggests more fundamental mitigations are needed to contain ever-emerging speculative execution attacks.

I. Introduction

Since the original Meltdown and Spectre disclosure, the family of memory disclosure attacks abusing speculative execution [1] has grown steadily [1], [2], [3], [4], [5]. While these attacks can leak sensitive information across security boundaries, they are all subject to strict addressing restrictions. In particular, Spectre variants [2], [4], [5] allow attacker-controlled code to only leak within the loaded virtual address space. Meltdown [1] and Foreshadow [3] require the target physical address to at least appear in the loaded address translation data structures. Such restrictions have exposed convenient anchor points to deploy practical "spot" mitigations against existing attacks [6], [7], [8], [9]. This shaped the common perception that—until in-silicon mitigations are available on the next generation of hardware—per-variant, software-only mitigations are a relatively pain-free strategy to contain ever-emerging memory disclosure attacks based on speculative execution.

In this paper, we challenge the common perception by introducing *Rogue In-flight Data Load* (RIDL), a new class of speculative execution attacks that lifts all such addressing restrictions entirely. While existing attacks target information at specific addresses, RIDL operates akin to a passive sniffer that eavesdrops on *in-flight* data (e.g., in the *line fill buffers* or *LFBs*) flowing through CPU components. RIDL is powerful: it can leak information across address space and privilege boundaries by solely abusing micro-optimizations implemented in commodity Intel processors. Unlike existing attacks, RIDL is non-trivial to stop with practical mitigations in software.

The vulnerability of existing vulnerabilities. To illustrate how existing speculative execution vulnerabilities are subject to addressing restrictions and how this provides defenders convenient anchor points for "spot" software mitigations, we consider their most prominent examples.

Spectre [2] allows attackers to manipulate the state of the branch prediction unit and abuse the mispredicted branch to leak arbitrary data within the accessible address space via a side channel (e.g., cache). This primitive by itself is useful in sandbox (e.g., JavaScript) escape scenarios, but needs to resort to *confused-deputy* attacks [10] to implement cross-address space (e.g., kernel) memory disclosure. In such attacks, the attacker needs to lure the victim component into speculatively executing specific "gadgets", disclosing data from the victim address space back to the attacker. This requirement opened the door to a number of practical software mitigations, ranging from halting speculation when accessing untrusted pointers or indices [7] to not speculating across vulnerable branches [6].

Meltdown [1] somewhat loosens the restrictions of the addresses reachable from attacker-controlled code. Rather than restricting the code to valid addresses, an unprivileged attacker can also access privileged address space mappings that are normally made inaccessible by the supervisor bit in the translation data structures. The reason is that, while any access to a privileged address

‡Work started during internship at Microsoft Research.

[1]Unless otherwise noted, we loosely refer to both speculative and out-of-order execution as speculative execution in this paper.

DOI 10.1109/SP.2019.00087

will eventually trigger an error condition (i.e., invalid page fault), before properly handling it, the CPU already exposes the privileged data to out-of-order execution, allowing disclosure. This enables cross-address space (user-to-kernel) attacks, but only in a traditional user/kernel unified address space design. This requirement opened the door to practical software mitigations such as KPTI [9], where the operating system (OS) isolates the kernel in its own address space rather than relying on the supervisor bit for isolation.

Foreshadow [3] further loosens the addressing restrictions. Rather than restricting attacker-controlled code to valid and privileged addresses, the attacker can also access physical addresses mapped by invalid (e.g., non-present) translation data structure entries. Similar to Meltdown, the target physical address is accessed via the cache, data is then passed to out-of-order execution, and subsequently leaked before the corresponding invalid page fault is detected. Unlike Meltdown, given the milder addressing restrictions, Foreshadow enables arbitrary cross-address space attacks. But this is only possible when the attacker can surgically control the physical address of some invalid translation structure entry. This requirement opened the door to practical software mitigations such as PTE inversion [8], where the OS simply masks the physical address of any invalidated translation structure entry.

A new RIDL. With RIDL, we show our faith in practical, "spot" mitigations being able to address known and future speculative execution attacks was misplaced. As we shall see, RIDL can leak in-flight data of a victim process even if that process is not speculating (e.g., due to Spectre mitigations) and it does not require control over address translation data structures at all. These properties remove all the assumptions that spot mitigations rely on. Translation data structures, specifically, enforce basic security mechanisms such as isolation, access permissions and privileges. Relaxing the requirement on translation data structures allows RIDL to mount powerful cross-address space speculative execution attacks directly from error-free and branchless unprivileged execution for the first time. In particular, by snooping on in-flight data in the CPU, attackers running arbitrary unprivileged code (including JavaScript in the browser) may leak information across arbitrary security boundaries. In essence, RIDL puts a glass to the wall that separates security domains, allowing attackers to listen to the babbling of CPU components.

To investigate the root cause of the RIDL class of vulnerabilities, we report on our reverse engineering efforts on several recent Intel microarchitectures. We show RIDL stems from micro-optimizations that cause the CPU to serve speculative loads with extraneous CPU-internal *in-flight* data. In the paper, we focus on instances serving arbitrary, address-agnostic data from the *Line Fill Buffers* (LFBs), which we found to be exploitable in the practical cases of interest. We also report on the challenges to exploit such instances in practice, targeting specific *in-flight* data to leak in particular. Moreover, we present a number of practical exploits that leak data across many common security boundaries (JavaScript sandbox, process, kernel, VM, SGX, etc.). We show exploitation is possible in both cross-thread and same-thread (no SMT) scenarios. This applies to all the existing Intel systems with the latest (microcode) updates and all the defenses up. In particular, RIDL bypasses all the practical mitigations against existing attacks and even the more heavyweight, default-off mitigations such as periodic L1 flushing. The lesson learned is that mitigating RIDL-like attacks with practical software mitigations is non-trivial and we need more fundamental mitigations to end the speculative execution attacks era.

Contributions. We make the following contributions:

- We present RIDL, a new class of speculative execution vulnerabilities pervasive in commodity Intel CPUs. RIDL enables unprivileged attackers to craft address-agnostic memory disclosure primitives across arbitrary security boundaries for the first time and has been rewarded by the Intel Bug Bounty Program.
- We investigate the root cause of practical RIDL instances abusing Line Fill Buffers (LFBs), presenting the first reverse engineering effort to analyze LFB behavior and related micro-optimizations.
- We present a number of real-world exploits that demonstrate an unprivileged RIDL-enabled attacker can leak data across arbitrary security boundaries, including process, kernel, VM and SGX, even with all mitigations against existing attacks enabled. For example, we can leak the contents of the `/etc/shadow` in another VM using a RIDL attack. Demos of these RIDL exploits can be found at https://ridl.eu.
- We place RIDL in the context of state-of-the-art attacks and mitigation efforts. Our analysis shows that, unlike existing attacks, RIDL is ill-suited to practical mitigations in software and more fundamental mitigations are necessary moving forward.

II. Background

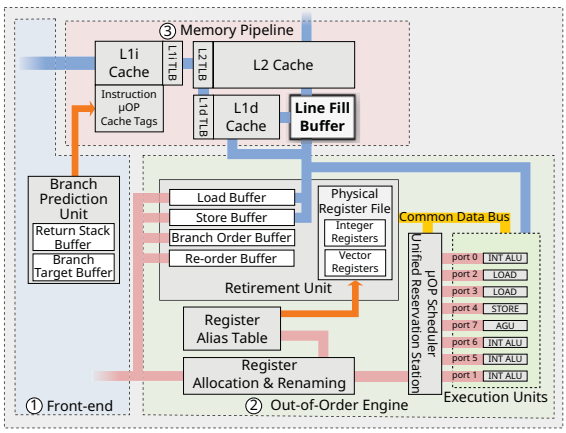

Fig. 1: An overview of the Intel Skylake microarchitecture.

Figure 1 shows an overview of the Intel Skylake microarchitecture. It consists of three stages: ① an in-order front-end that decodes instructions into μ-ops, ② an out-of-order execution engine, ③ and the memory pipeline.

Since the Intel Skylake microarchitecture is quite complex, we specifically focus on the cache hierarchy, out-of-order/speculative execution, and in-flight data.

A. Caches

To overcome the growing performance gap between processors and memory, the processor contains small memory buffers, called caches, to store frequently and recently used data to hide memory latency. Modern processors have multiple levels of caches with the smallest and fastest close to the processor, and the largest but slowest being the furthest away from the processor. The Intel Core microarchitecture has three levels of CPU caches. At the first level, there are two caches, L1i and L1d, to store code and data respectively, while the L2 cache unifies code and data. Where these caches are private to each core, all cores share the L3 or last-level cache (LLC). The LLC is inclusive of the lower-level caches and set-associative, i.e., divided into multiple cache sets where part of the physical address is used to index into the corresponding cache set.

Gullasch et al. [11] use `clflush` to evict targets to monitor from the cache. By measuring the time to reload them the attacker determines whether the victim has accessed them—a class of attacks called FLUSH + RELOAD [12]. Another variant is PRIME + PROBE [13], [14], [15], [16], in which the attacker builds an eviction set of memory addresses to fill a specific cache set. By repeatedly measuring the time it takes to refill the cache set, the attacker can monitor memory accesses to that cache set.

B. Out-of-Order Execution

To improve the instruction throughput, modern CPUs implement a superscalar out-of-order execution pipeline similar to Tomasulo's algorithm [17], [18]. Out-of-order execution engines generally consist of three stages: (1) in-order register allocation & renaming, (2) out-of-order execution of instructions or μ-ops, (3) and in-order retirement.

a) Register renaming: Once the decoded μ-ops leave the front-end, they pass through the register allocation & renaming unit that renames the registers to eliminate *Write-after-Read* (WAR) and *Write-after-Write* (WAW) hazards. More specifically, this unit renames source/destination operands for μ-ops by allocating pointers to freely available physical registers from the *Physical Register File* (PRF) and maintaining the corresponding mappings in the *Register Alias Table* (RAT). After renaming the μ-op, the unit allocates an entry for the μ-op in the *Re-Order Buffer* (ROB) to preserve the original programming order and sends the μ-op to the reservation station.

b) Out-of-order scheduling: To eliminate *Read-after-Write* (RAW) hazards, the reservation station stalls each μ-op with unavailable operands. Once all the operands are available, the scheduler dispatches the μ-op to the corresponding execution unit, possibly before scheduling older μ-ops. After executing the μ-op, the reservation station stores its result, updates the μ-ops that depend on it, and marks the corresponding ROB entry as completed.

c) Retirement: The *Retirement Unit* retires completed μ-ops in their original program order by committing the architectural state for memory/branch operations and freeing up any allocated physical registers. In case of a mispredicted branch, the *Retirement Unit* retires the offending branch instruction, flushes the ROB, resets the reservation station, and replays the execution stream from the correct branch. The *Retirement Unit* also detects faulty instructions and generates precise exceptions once the offending μ-op reaches a non-speculative state.

C. Speculative Execution

To predict the target of a branch, modern processors feature a *Branch Prediction Unit* (BPU). The BPU predicts the branch target such that the processor can execute a stream of instructions speculatively. In case the predicted path is wrong, the processor reverts the state to the last known useful state and starts executing from the correct branch instead. There are several instances where speculation occurs, such as: conditional branches, indirect branches and calls, return instructions and transactions.

Recent Intel processors employ a *Branch Order Buffer* (BOB) to keep track of all in-flight branches and whether the branch is in retired or speculative state [19], [20]. The BOB is also used to implement memory transactions through *Transactional Synchronization eXtensions* (TSX). In particular, the `xbegin` instruction marks the start of a transaction and adds an entry to the BOB as a checkpoint. Transactions end once the processor encounters an `xend` instruction, an `xabort` instruction, or a fault. In case of an `xend` instruction, the processor commits the transaction, otherwise the processor rolls back the transaction by reverting back to the original state before the `xbegin`.

D. In-flight Data

There are many potential sources of in-flight data in modern CPUs such as the *Re-Order Buffer* (ROB), the *Load* and *Store Buffers* (LBs and SBs) [21], [22], [23], [24], the *Line Fill Buffers* (LFBs), and the *Super Queue* (SQ) [25], [26]. We focus here on two prominent examples: store buffers and line fill buffers.

Store Buffers (SBs) are internal buffers used to track pending stores and in-flight data involved in optimizations such as *store-to-load forwarding* [21], [24]. Some modern processors enforce a strong memory ordering, where load and store instructions that refer to the same physical address cannot be executed out-of-order. However, as address translation is a slow process, the physical address might not be available yet, and the processor performs memory disambiguation to predict whether load and store instructions refer to the same physical address [27]. This enables the processor to speculatively execute unambiguous load and store instructions out-of-order. As a micro-optimization, if the load and store instructions are ambiguous, the processor can speculatively *store-to-load forward* the data from the store buffer to the load buffer.

Line Fill Buffers (LFBs) are internal buffers that the CPU uses to keep track of outstanding memory requests and perform a number of optimizations such as merging

multiple in-flight stores. Sometimes, data may already be available in the LFBs and, as a micro-optimization, the CPU can speculatively load this data (similar optimizations may also be performed on *load/store buffers*, etc.). In both cases, modern CPUs that implement aggressive speculative execution may speculate without any awareness of the virtual or physical addresses involved. In this paper, we specifically focus on LFBs, which we found particularly amenable to practical, real-world RIDL exploitation.

III. Threat Model

We consider an attacker who wants to abuse speculative execution vulnerabilities to disclose some confidential information, such as private keys, passwords, or randomized pointers. We assume a victim Intel-based system running the latest microcode and OS version, with all the state-of-the-art mitigations against speculative execution attacks enabled. We also assume the victim system is protected against other classes of (e.g., software) vulnerabilities. Finally, we assume the attacker can only run unprivileged code on the victim system (e.g., JavaScript sandbox, user process, VM, or SGX enclave), but seeks to leak information across arbitrary privilege levels and address spaces.

IV. Overview

Figure 2 illustrates the main steps and the underlying mechanism enabling the RIDL leaks. First, as part of its normal execution, the victim code, in another security domain, loads or stores some secret data[2]. Internally, the CPU performs the load or store via some internal buffers—*Line Fill Buffers* (LFBs) in the RIDL instances considered in this paper. Then, when the attacker also performs a load, the processor speculatively uses in-flight data from the LFBs (with no addressing restrictions) rather than valid data. Finally, by using the speculatively loaded data as an index into a Flush + Reload buffer (or any other covert channel), attackers can extract the secret value.

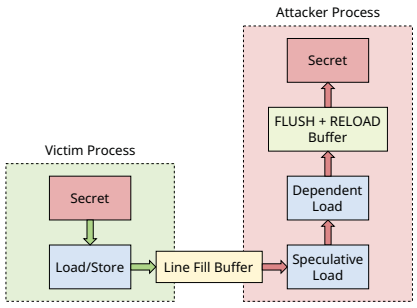

Fig. 2: An overview of the RIDL attack.

A simple example of our attack is shown in Listing 1. As shown in the listing, the code is normal, straight-line code without invalid accesses (or, indeed, error suppression), which, as we will show, can also be implemented in managed languages such as JavaScript. Lines 2–3 only flush the buffer that we will later use in our covert channel to leak the secret that we speculatively access in Line 6. Specifically, when executing Line 6, the CPU speculatively loads a value from memory in the hope it is from our newly allocated page, while really it is in-flight data from the LFBs belonging to an arbitrarily different security domain.

```
1  /* Flush flush & reload buffer entries. */
2  for (k = 0; k < 256; ++k)
3    flush(buffer + k * 1024);
4
5  /* Speculatively load the secret. */
6  char value = *(new_page);
7  /* Calculate the corresponding entry. */
8  char *entry_ptr = buffer + (1024 * value);
9  /* Load that entry into the cache. */
10 *(entry_ptr);
11
12 /* Time the reload of each buffer entry to
13    see which entry is now cached. */
14 for (k = 0; k < 256; ++k) {
15   t0 = cycles();
16   *(buffer + 1024 * k);
17   dt = cycles() - t0;
18
19   if (dt < 100)
20     ++results[k];
21 }
```

Listing 1: An example of RIDL leaking in-flight data.

When the processor eventually detects the incorrect speculative load, it will discard any and all modifications to registers or memory, and restart execution at Line 6 with the right value. However, since traces of the speculatively executed load still exist at the micro-architectural level (in the form of the corresponding cache line), we can observe the leaked in-flight data using a simple (Flush + Reload) covert channel—no different from that of other speculative execution attacks. In fact, the rest of the code snippet is all about the covert channel. Lines 8-10 speculatively access one of the entries in the buffer, using the leaked in-flight data as an index. As a result, the corresponding cache line will be present. Lines 12-21 then access all the entries in our buffer to see if any of them are significantly faster (indicating that the cache line is present)—the index of which will correspond to the leaked information. Specifically, we may expect *two* accesses to be fast, not just the one corresponding to the leaked information. After all, when the processor discovers its mistake and restarts at Line 6 with the right value, the program will also access the buffer with this index.

Our example above use demand paging for the loaded address, so the CPU restarts the execution only after handling the page-in event and bringing in a newly mapped page. Note that this is not an error condition, but rather a normal part of the OS' paging functionality. We found many other ways to speculatively execute code using in-flight data. In fact, the accessed address is not at all important. As an extreme example, rather than accessing a newly mapped page, Line 6 could even dereference a `NULL` pointer and later suppress the error (e.g., using TSX). In

[2]Strictly speaking, this is not even a hard requirement, as we can also leak data from inactive code by forcing cache evictions.

general, any run-time exception seems to be sufficient to induce RIDL leaks, presumably because the processor can more aggressively speculate on loads in case of exceptions. Clearly, one can only "speculate" here, but this behavior seems consistent with existing vulnerabilities [1], [3]. Similarly, we noticed the address accessed by the attacker should be part of a page-aligned cache line to induce leaks.

While the basic concept behind in-flight data may be intuitive, successfully implementing an attack turned out to be challenging. Unlike prior work that builds on well-documented functionality such as branch prediction, page tables and caches, the behavior of internal CPU buffers such as LFBs is largely unknown. Moreover, different microarchitectures feature different types of buffers with varying behavior. Furthermore, based on publicly available documentation, it was not clear whether many of these buffers even *exist*. For our attack to succeed, we had to resort to extensive reverse engineering to gain a better understanding of these buffers and their interaction with the processor pipeline. The next section discusses how we determine exactly which in-flight data buffers are responsible for the leak, how to manipulate the processor state in such a way that we can perform a speculative load that uses the in-flight data (so that we can use our covert channel to obtain the content), and how to ensure the data we want to leak actually ends up in the buffers.

V. Line fill buffers and how to use them

To perform the attack described in the previous section, we first need to understand the core building blocks for the RIDL variant considered in the paper: the *Line Fill Buffers* (LFBs). Using reverse engineering and experimentation, we verify that we do indeed leak from the LFBs (and not some other buffer) and examine their interaction with the processor pipeline. After that, we discuss how attackers can control what to leak by synchronizing with the victim.

In Intel processors, the LFB performs multiple roles: it enables non-blocking caches, buffers non-temporal memory traffic [28], [29], [30], [31], and performs both load squashing [32], [33], [34] and write combining [35], [36], [37]. To help the reader understand the remainder of this paper, we now briefly discuss each of these functions.

Non-blocking cache. Cache misses have a serious impact on performance as they block the data cache until the data is available. To allow non-blocking reads, the LFB implements multiple *Miss Status Holding Registers* (MSHRs) to track the physical addresses of outstanding requests until the data is available [38], [39]. For example, the Haswell microarchitecture maintains 10 L1 MSHRs in its LFB to service outstanding L1 cache misses [40], [41] These MSHRs free up the L1d cache to allow load instructions that hit the L1d cache to bypass cache misses.

Load squashing. To further optimize performance, the LFB squashes multiple load misses to the same physical address. If there is already an outstanding request in the LFB with the same physical address, the processor assigns the same LFB entry to a load/store with the same address.

Write combining. For weakly-ordered memory, the processor keeps stores to the same cache line within the LFB to perform *write combining*. That is, the processor merges multiple stores in a single LFB entry before writing out the final result through the memory hierarchy.

Non-temporal requests. Finally, modern processors support non-temporal memory traffic where the programmer already knows that caching the data is of no benefit at all. In that case, the processor performs non-temporal loads and stores exclusively through the LFB.

A. Solving a RIDL: LFB leaks on loads and stores

Unaware of the source of the RIDL leaks initially, we discovered that they originate from the LFBs, rather than from other processor state, by conducting several experiments on a workstation featuring an Intel Core i7-7700K (Kaby Lake). For our experiments, we use a kernel module to mark memory pages in our victim thread as *write-back* (WB), *write-through* (WT), *write-combine* (WC) and *uncacheable* (UC) [42], [43]. We use Intel TSX to implement the attack for our analysis and perform $10,000$ rounds to leak the data during every run of the experiment. Furthermore, we run every experiment 100 times and report the average.

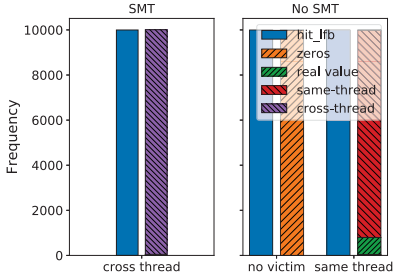

Fig. 3: In each pair of bars, one bar shows the LFB hit count, and the other one the number of attacks. With SMT, we always leak the secret. Without SMT and no victim code, RIDL only reads zeros, but with victim and attacker in the same hardware thread, we still leak the secret in most cases (top/red bar), while occasionally finding the value the CPU *should* have loaded.

Our first experiment performs the attack discussed earlier against a victim running in the same or other hardware thread and repeatedly storing a secret to a fixed memory address. We then compare the number of hits in the LFB, measured using the *lfb_hit* performance counter, to the number of attack iterations. Consider the left-most (SMT) plot of Figure 3 which shows close correspondence between the number of LFB hits and the number of attempts for leaking (and correctly obtain the secret). This strongly suggests that the source of our leak is the LFB.

In our second experiment, a victim thread initially writes a known value A to a fixed memory address, and then reads it back in a loop where each read is followed by an `mfence` instruction (serializing all loads and stores issued prior to the `mfence`). We mark this address as WB, WT, WC and UC. Optionally, we also flush the address using `clflush` (which flushes it from the cache). Figure 4 shows how often a RIDL attacker reads the secret value correctly. Note that we do not observe the signal for WB

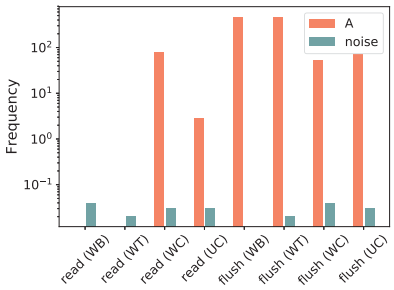

Fig. 4: Leaking the secret A which is *read* by the victim for write-back (WB), write-through (WT), write-combine (WC) and uncacheable (UC) memory, with and without a cache flush.

and WT memory if we do not flush the entry from the cache, but when we do, we observe the signal regardless of the memory type. Both observations indicate that we are not leaking from the cache. Also, since we leak from a load, this cannot be the effect of *store-to-load forwarding* either. Furthermore, since we observe the signal from WC and UC memory, which have to go through the LFB, the source of our leak must again be the LFB.

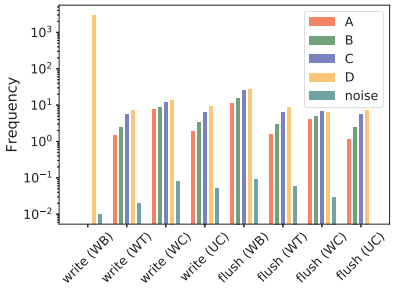

Fig. 5: Leaking the secrets A, B, C and D, *written* by the victim, for write-back (WB), write-through (WT), write-combine (WC) and uncacheable (UC) memory, with and without a cache flush.

To gather further evidence that we are leaking from the LFB, we perform a third experiment where we run a victim thread which, in a loop, writes four different values to four sequential cache lines, followed by an `mfence`. Optionally, the victim thread again flushes the cache lines. We again rule out any leaks via store-to-load forwarding, by turning on *Speculative Store Bypass Disable* (SSBD [44]) for both attacker and victim. Figure 5 shows the RIDL results. For WB without flushing, there is a signal only for the last cache line, which suggests that the CPU performs write combining in a single entry of the LFB before storing the data in the cache. More importantly, we observe the signal regardless of the memory type when flushing. Since both flushing and the WT, WC and UC memory types enforce direct invalidation of writes, they must go through the LFB. This third experiment again indicates that the source of our leak must be the LFB.

Conclusion: our RIDL variant leaks from the Line Fill Buffers (LFBs).

To launch a RIDL attack, we still need to understand the interaction between loads, stores, the L1d cache and the LFB, such that we can massage the data from the victim into the LFB. Recall that in our read experiment (Figure 4), we did not observe a signal if we do not flush the address, even with multiple consecutive reads like we did with writes (Figure 5). As the data is read constantly, all loads simply hit in the L1d cache, preventing any interaction of future loads with the LFB. In contrast, when we do flush, the future loads miss and allocate an entry in the LFB to await the data. In case of WC and UC memory, the processor avoids the L1d cache and enforces the loads to always go through the LFB. Our second experiment (Figure 5) shows a signal for all memory types and especially those that bypass the L1d cache, suggesting that memory writes go via the LFB.

Conclusion: reads that are not served from L1d pull data through the LFB, while writes push data through the LFB to either L1d or memory.

B. Synchronization

To leak information, the attacker must make sure that the right data is visible in the LFB at the right time, by synchronizing with the victim. We show that there are three ways to do so: serialization, contention and eviction.

Serialization. Intel CPUs implement a number of barriers to perform different types of serialization [45], [46], [47], [48]. `lfence` guarantees that all loads issued before the `lfence` become globally visible, `sfence` guarantees the same for all store instructions, and `mfence` guarantees that both load and stores before the `mfence` become globally visible. To enforce this behavior, the processor waits for these loads and stores to retire by draining the load and/or store buffers, as well as the corresponding entries in the LFB. The `mfence` instruction therefore forms a point of synchronization that allows us to observe the last few loads and stores before the buffers are completely drained.

Contention. Another way of synchronizing victim and attacker is to create contention within the LFB, ultimately forcing entries to be evicted. Doing so allows us to obtain some control over the entries that we leak, and should not depend on SMT. To verify this, we perform the RIDL attack *without SMT* by writing values in our own thread and observing the values that we leak from the same thread. Figure 3 shows that if we do not write the values ("no victim"), we leak only zeros, but with victim and attacker running in the same hardware thread (e.g., in a sandbox), we leak the secret value in almost all cases.

Eviction. Finally, we can control the values that we leak from the victim by evicting cache entries from the cache set in which we are interested. To show that we can use this for synchronization, we conducted an experiment where the victim writes a value to the same cache line within a number of pages. After a while, these writes end up evicting the previous cache lines from the L1d cache. As these cache lines are dirty, the processor has to write them back through the memory hierarchy and will do this through the LFB. We extend the victim thread to alternate

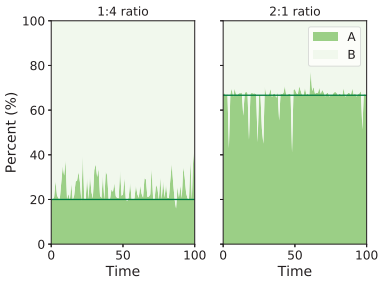

Fig. 6: Leaking the secrets A and B *written* by the victim to a series of cache lines to trigger continuous eviction. On the left, the victim writes A then B using a 1:4 ratio. On the right, the victim writes A then B using a 2:1 ratio.

between two different values to write after finishing every loop and also vary the amount of pages to write during the loop. For the first test, we write the first value 1024 times and the second value 256 times (ratio 1:4) and for the second test, we write the first value 512 times and the second 1024 times (1:2 ratio). Figure 6 shows the results of this experiment, where we observe the first value 80% of the times and the second value 20% of the times in the case of ratio 1:4 and the first value 33.3% of the times and the second value 66.6% of the times in the case of ratio 2:1. Hence, we conclude that we can control the (dirty) cache entry to leak through eviction.

Conclusion: we can use serialization, contention and eviction to synchronize attacker and victim.

VI. Exploitation with RIDL

The techniques described in the previous section allow us to leak in-flight CPU data in a controlled fashion. Since the underlying buffers are independent of address spaces and privilege levels, we can mount attacks across these security boundaries.

We have verified that we can leak information across arbitrary address spaces and privilege boundaries, even on recent Intel systems with the latest microcode updates and latest Linux kernel with all the Spectre, Meltdown, L1TF default mitigations up (KPTI, PTE inversion, etc.). In particular, the exploits we discuss below exemplify leaks in all the relevant cases of interest: process-to-process, kernel-to-userspace, guest-to-guest, and SGX-enclave-to-userspace leaks. Not to mention that such attacks can be built even from a sandboxed environment such as JavaScript in the browser, where the attacker has limited capabilities compared to a native environment.

We stress that the only requirement is the presence of in-flight secret data managed by the processor. In a non-SMT single-core attack scenario, this is data recently read-/written by the victim before a mode switching instruction (`iret`, `vmenter`, etc.). In an SMT attack scenario, this is data concurrently read/written by another hardware thread sharing the same CPU core. Once we have speculatively leaked a value, we use the techniques discussed earlier (based on Flush + Reload, or Evict + Reload when `clflush` is not available) to expose the desired data. The key difference with prior Meltdown/L1TF-style attacks that cross privilege boundaries and address spaces is that the target address used by the attacker can be perfectly *valid*. In other words, the attack does not necessarily require a TSX transaction or an invalid page fault, but can also be applied to a correct, branchless execution with demand paging (i.e., a valid page fault) as we showed in Section V. This bypasses side-channel mitigations deployed on all the major operating systems and extends the threat surface of prior cross-address space speculative execution attacks to managed sandboxes (e.g., JavaScript). In the next sections, we explore how RIDL can be used to leak sensitive information across different security boundaries.

Covert channel. We performed an extensive evaluation of RIDL over a number of microarchitectures, showing that it affects all recent Intel CPUs. To verify that RIDL works across all privilege boundaries, we implemented a proof-of-concept covert channel, sending data across address space and privilege boundaries.

In Table I, we present the bandwidth of the covert channel. Note that our implementation is not yet optimized for all architectures. For convenience, we utilize Intel TSX on the architectures where available, as this gives us the most reliable covert channel. Using TSX, we achieve a bandwidth of 30-115 kB/s, where the limiting factor is Flush + Reload. Where TSX is not available, we present numbers from an unoptimized proof-of-concept implementation which uses either demand paging, or exception suppression using speculative execution.

Challenges. In the previous sections, we discussed how the building blocks of RIDL are used to leak in-flight data and that we can use RIDL to leak information across security domains. Applying these techniques to exploit real-world systems—leaking confidential data—presents some additional challenges that we need to overcome:

1) **Getting data in-flight**. We need to find ways to get restricted data that we want to leak into the LFB. There are some obvious mechanisms for an unprivileged user to get privileged data in-flight: interaction with the kernel (i.e., syscalls), and interaction with a privileged process (i.e., invoking a setuid binary). There are also many other possibilities, such as manipulating the page cache.

2) **Targeting**. Due to the high amount of LFB activity, getting the desired data out of the LFB poses a challenge. We describe two mechanisms for targeting the data we want to leak: *synchronizing the victim*, and *aligning the leaked data* by repeating the attack multiple times while filtering out the noise.

In the next sections, we demonstrate a number of exploits that use RIDL. We evaluated all the exploits on the Intel Core i7-7800X running Ubuntu 18.04 LTS.

A. Cross-process attacks

In a typical real-world setting, synchronizing at the exact point when sensitive data is in-flight becomes non-trivial, as we have limited control over the victim process.

TABLE I: Our results for 15 different microarchitectures and the measured bandwidth across security domains.

CPU	Year	Microcode	Page Fault Demand Paging		Misaligned Read		TSX		SGX		Bandwidth (B/s)			
			Same-Thread	Cross-Thread	Same-Thread	Cross-Thread	Same-Thread	Cross-Thread	Same-Thread	Cross-Thread	Cross-Process	Cross-Privilege	Cross-VM	SGX
Intel Xeon Silver 4110 (Skylake SP)	2017	0x200004d	R/W	R/W	R/W	R/W	✓	✓	—	—	45k	25k	3k	—
Intel Core i9-9900K (Coffee Lake R)	2018	0x9a	✗	✗	R/W	R/W	✓	✓	✓	✓	71k	48k	10k	8k
Intel Core i7-8700K (Coffee Lake)	2017	0x96	R/W	R/W	R/W	R/W	✓	✓	✓	✓	54k	49k	46k	65k
Intel Core i7-7800X (Skylake X)	2017	0x200004d	R/W	R/W	R/W	R/W	✓	✓	—	—	37k	36k	31k	—
Intel Core i7-7700K (Kaby Lake)	2017	0x8e	R/W	R/W	R/W	R/W	✓	✓	✓	✓	65k	46k	63k	114k
Intel Core i7-6700K (Skylake)	2015	0xc6	R/W	R/W	R/W	R/W	✓	✓	✓	✓	68k	20k	76k	83k
Intel Core i7-5775C (Broadwell)	2015	0x1e	R/W	R/W	R/W	R/W	✓	✓	—	—	21k	16k	27k	—
Intel Core i7-4790 (Haswell)	2014	0x25	R/W	R/W	R/W	R/W	—	—	—	—	100	50	110	—
Intel Core i7-3770K (Ivy Bridge)	2012	0x20	R/W	R/W	R/W	R/W	—	—	—	—	92	41	89	—
Intel Core i7-2600 (Sandy Bridge)	2011	0x2e	R/W	R/W	R/W	R/W	—	—	—	—	107	73	106	—
Intel Core i3-550 (Westmere)	2010	0x07	R/W	R/W	R/W	R/W	—	—	—	—	1k	245	1k	—
Intel Core i7-920 (Nehalem)	2008	0x12	R/W	R/W	R/W	R/W	—	—	—	—	79	32	70	—
AMD Ryzen 5 2500U (Raven Ridge)	2018	0x810100b	✗	✗	✗	✗	—	—	—	—	—	—	—	—
AMD Ryzen 7 2600X (Pinnacle Ridge)	2018	0x800820b	✗	✗	✗	✗	—	—	—	—	—	—	—	—
AMD Ryzen 7 1600X (Summit Ridge)	2017	0x8001137	✗	✗	✗	✗	—	—	—	—	—	—	—	—

Instead, by repeatedly leaking the same information and aligning the leaked bytes, we can retrieve the secret without requiring a hard synchronization primitive. For this purpose, we show a noise-resilient *mask-sub-rotate* attack technique that leaks 8 bytes from a given index at a time.

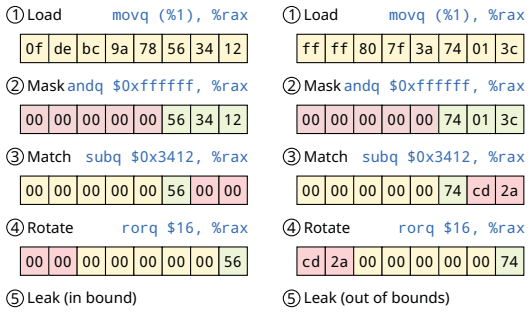

Fig. 7: Using mask, subtract, and rotate we can selectively filter data in speculative execution to match prior observations, eliminating a large amount of noise.

As shows in Figure 7, ① suppose we already know part of the bytes we want to leak (either by leaking them first or knowing them through some other means). ② In the speculative path we can mask the bytes that we do not know yet. ③ By subtracting the known value, ④ and then rotating by 16 bytes, values that are not consistent with previous observations will be out of bounds in our FLUSH + RELOAD buffer, meaning we do not leak them. This technique greatly improves the observed signal.

We use this technique to develop an exploit on Linux that is able to leak the contents of the `/etc/shadow` file. Our approach involves repeatedly invoking the privileged `passwd` program from an unprivileged user. As a result the privileged process opens and reads the `/etc/shadow` file, that ordinary users cannot access otherwise. Since we cannot modify the victim to introduce a synchronization point, we repeatedly run the program and try to leak the LFB while the program reads the `/etc/shadow` file. By applying our previously discussed technique, with the additional heuristic that the leaked byte must be a printable ASCII character, we are able to leak the contents of the file even with the induced noise from creating processes.

One observation is that the first line of the `/etc/shadow` file contains the entry for the root user. Therefore we can apply our alignment technique by fixing the first five characters to the known string `root:` to filter data from `/etc/shadow`. This approach is especially powerful as it does not require any additional information about the memory layout of the system. The attacker simply passively listens to all LFB activity, and matches the data with previous observations. As seen in Figure 8, we recover 26 characters (leaking 21 unknown bytes) from the `shadow` file after 24 hours. The hash entry of the root user consists of 34 characters, which leaves 8 characters (or several hours) left to leak. As this was only our initial attempt to utilize RIDL for real-world attacks, we already know we can improve the speed significantly.

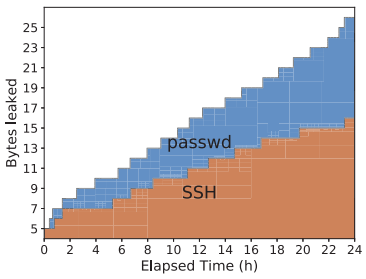

Fig. 8: Characters leaked the `/etc/shadow` file using the `passwd` and `SSH` attack over a period of 24 hours.

B. Cross-VM attacks

When two different virtual machines are executing simultaneously on the same physical core, a user process

running inside one VM can observe in-flight data from the other VM. We verified that RIDL works on KVM [49] and even on Microsoft Hyper-V (on both Windows 10 and Windows Server 2016) with all side-channel mitigations enabled (HyperClear [50]). KVM has deployed defenses against L1TF which flush the L1D cache on `vmenter`. By default—for performance reasons—the L1D is not flushed on specific (manually) audited code paths. This defense does not hinder RIDL. In fact, flushing the L1D might actually force sensitive data to be in-flight.

We also implemented a cross-VM attack where a co-located attacker leaks the `/etc/shadow` file from the victim VM by repeatedly trying to authenticate through SSH, confirming that virtual machine isolation does not mitigate this class of vulnerabilities. The attacker repeatedly opens a connection to the victim, trying to authenticate using invalid credentials. Similarly to the previous `passwd` attack, this strategy causes the victim to read the `/etc/shadow` file, allowing us to leak the contents. For our proof-of-concept exploit, we assume we have two co-located VMs running on co-located SMTs. We are able to retrieve 16 characters from the `passwd` file over a period of 24 hours, This is slightly slower than the previous `passwd` attack, since the execution path when SSH reads the `shadow` file is significantly longer than for the `passwd` program.

C. Kernel attacks

To verify that the privilege level does not affect our attack, we implement a user program that opens the `/proc/self/maps` file (or any other `/proc` file) and reads 0 bytes from that file. The read system call causes the kernel to generate the string containing the current mappings of the process, but copies 0 bytes to the address space of the calling program. Using the previously mentioned attacker program running on a sibling hardware thread, we are able to leak the first 64 bytes of the victim process memory mappings without these ever having been copied to user space. Our proof-of-concept exploit is able to do this reliably in a matter of milliseconds.

The kernel also provides us with a convenient target for attacks which do not require SMT. We can easily leak kernel pointers and other data stored on the stack close to the end of a system call, by executing a syscall and then performing our attack immediately after the kernel returns control to userspace. Since the kernel writes also use the LFBs, we also implemented proof-of-concept exploits that leak kernel memory writes occurring in the middle of normal execution (for example, in a `/proc` handler) in a few milliseconds. We observe the values of these writes after the kernel has already returned from the system call, as the cache lines are written back to memory via the LFB.

D. Leaking arbitrary kernel memory

RIDL can leak secrets accessed by the victim via both regular and speculative memory accesses. We demonstrate this property by implementing a proof-of-concept exploit that can leak arbitrary kernel memory. In absence of software bugs, unsanitized user-supplied pointers are never accessed by the kernel. However, speculative memory accesses are still possible. For example, we found that the function `copy_from_user()` in the Linux kernel (version 4.18) allows speculative memory accesses to user-supplied pointers. It is important to note that this attack is only possible if Supervisor Mode Access Prevention (*SMAP*) is not enabled, otherwise all accesses to user memory will be surrounded with serializing instructions (`clac/stac`), effectively stopping speculation. Our exploit assumes that SMAP is disabled, for example due to lack of hardware/-software support.

In our exploit, we use the `setrlimit()` system call to reach the `copy_from_user()` function. We start by calling `setrlimit()` multiple times with a user-land pointer to train the directional branch predictor. After the training, we call `setrlimit()` with the pointer to the kernel data we want to leak. The speculative memory access reads the data from memory, notably via the LFB, allowing us to leak it in our program after returning from the system call. To measure the performance of our exploit, we tried leaking the data from a kernel memory page (4,096 bytes) containing ASCII text. On average, leaking the entire page took us around 712 seconds, i.e., approximately 6 B/s.

E. Page table disclosure

We implemented proof-of-concept exploits to leak page table entries, since the MMU uses the LFBs to read these entries from memory at every page table walk. This discloses the physical addresses of pages in our own process, which is important information to mount other attacks (such as Rowhammer attacks [51], [52], [53], [54], [55], [56], [57], [58], [59]). This also allows us to observe page table walks and the page table entries used by a process running on a sibling hardware thread. Furthermore, by performing sliding similar to the AnC attack [60], this primitive allows us to break ASLR in a managed sandbox.

F. SGX attacks

We also verified that SGX enclaves are vulnerable to our cross-process attacks when SMT is enabled, allowing an attacker on the same physical core to leak SGX-initiated reads and writes to memory. We built SGX enclaves in pre-release mode (with debugging disabled), and successfully reproduced the cross-process experiments. Our proof-of-concept exploit trivially leaks reads and writes from a victim enclave running on a sibling hardware thread.

Our exploit can also leak the values of registers used by the enclave, since microcode reads and writes the contents of enclave registers to memory when the enclave is interrupted. By using `mprotect` to cause faults when accessing enclave pages, we repeatedly interrupt the enclave (to synchronize), allowing us to leak the contents of enclave registers. Unlike the Foreshadow attack [3], we are able to perform these attacks solely from *user space*, with no need to manipulate privileged state such as page tables. This means that SGX enclave secrets should be considered compromised on any machine where SMT is enabled, even if an attacker does not have control over the kernel or hypervisor. When an attacker is able to modify the kernel, this attack can be further improved with SGX-Step [61], using timer interrupts to single-step through enclave code and provide a fine-grained synchronization primitive.

G. JavaScript attacks

To further demonstrate the implications of RIDL, we show that RIDL can be exploited even within restricted sandboxed environments such as JavaScript. In recent years, browser vendors have been proactively working on mitigations to protect against side-channel attacks [62], [63], [64], [65]—speculative execution side channels, in particular. For instance, Chrome fast-forwarded the deployment of process-per-origin [63] as a mitigation against Spectre attacks. However, these mitigation efforts assume that data cannot leak across privilege boundaries, and fail to prevent in-flight data from being leaked with RIDL.

Building a RIDL attack from the browser requires a high level of control over the instructions executed by the JavaScript engine. Conveniently, *WebAssembly* allows us to generate code which meets these requirements and is available as a standard feature in modern browsers. We found that we can use WebAssembly in both Firefox and Chrome to generate machine code which we can use to perform RIDL-based attacks. Furthermore, all the major browsers try to reduce the memory footprint of the WebAssembly heap by relying on demand paging [60], which we can use to perform an attack along the lines of the one previously presented in Listing 1. That is, we can rely on the *valid* page fault generated by our memory access to trigger an exception and spill the *in-flight* data.

Generating the correct machine code and triggering the page fault are relatively straightforward. However, constructing a reliable feedback channel for speculative attacks within the browser presents some challenges. The absence of the `clflush` instruction forced our implementation to rely on an Evict + Reload channel to leak the in-flight data. Since the process of evicting entries from the L1D cache makes extensive use of the LFBs—due to TLB misses as well as filling cache lines—this adds a significant source of noise. We also need to ensure that the TLB entries for our reload buffer are still present after the eviction process, adding another source of noise to our attack. Finally, we need a reliable high-resolution timer to measure cache evictions for our Evict + Reload channel. While built-in high-resolution timers have been disabled as part of browser mitigations against side-channel attacks [64], [63], prior work has demonstrated a variety of techniques to craft new high-resolution timers [66], [56], [60], such as SharedArrayBuffer [60] and GPU-based counters [56]. The SharedArrayBuffer feature was recently re-enabled in Google Chrome, after the introduction of Site Isolation [67], [68]. Mozilla Firefox is currently working on a similar Process Isolation strategy [69].

Despite these challenges, we successfully implemented a proof-of-concept exploit on top of Firefox' SpiderMonkey JavaScript engine to reliably leak data from a victim process running on the same system. For simplicity, our exploit uses an old-style built-in high-resolution timer in SpiderMonkey to measure cache evictions. When targeting a victim process repeatedly writing a string to memory, our exploit running in the JavaScript sandbox on a different hardware thread is capable of leaking the victim string at a rate of ~1 B/s. We also implemented a high-resolution timer in Chrome using WebAssembly threads which provided sufficient accuracy for our Evict + Reload channel. At the time of writing, any site can opt into this functionality using the 'origin trials' system. Although we do not currently have a reliable RIDL exploit running inside unmodified Chrome, we believe that our results already cast doubt on the effectiveness of site isolation as a mitigation against side-channel attacks.

VII. Speculative execution attacks

Since Horn [79] initially reported this new class of speculative execution side-channel attacks, researchers started digging into modern processor microarchitectures to spot the next generation of vulnerabilities. The result is a plethora of new attacks and attack vectors [1], [2], [3], [75], [5], [70], [4], [71].

The taxonomy of these attacks is confusing (at best) since *attacks* and *attack vectors* oftentimes have been reported as equivalent and frequently interchanged. In this section, we try to shed some light on the topic describing similarities and differences among the different classes of attacks and categorizing them based on their nature, capabilities, and constraints. We summarize our categorization in Table II. We divide the currently existing attacks based on the nature of their speculation: *control speculation* vs. *data speculation*. We further introduce a subcategorization of data speculation attacks, which we define as *exception deferral* attacks (e.g., RDCL and L1TF).

A. Control Speculation

Control speculation can be triggered in multiple ways. In Section II, we already described *Out-of-Order execution* and *Transactional Synchronization eXtensions* explaining how these trigger speculative execution. Here we focus on the three main forms of control instructions that can be speculated upon: ① direct branches, ② indirect branches and calls, and ③ return instruction.

Direct branches: Direct (or conditional) branches are optimized in hardware by the *Branch Prediction Unit* (BPU). This unit keeps track of the previous outcomes of a conditional branch in order to predict which code path will be taken, and the out-of-order execution engine then continues execution along the predicted path. Mistraining the BPU allows attacks known as *Bounds Check Bypass* (BCB) [2], [5], such as the one in Listing 2.

```
1  if (x < arr_size)
2      y = probeTable[arr[x] * 4096];
```

Listing 2: An example of Bounds Check Bypass.

An attacker who controls the variable `x` can mistrain the conditional branch to always take the `if` code path. When an out-of-bounds `x` is later passed to his code, the BPU will speculate on the code path to take, resulting in a speculative OoB access which can be leaked through a cache-based covert channel. A variant of this attack targets bounds check bypass on stores (BCBS) [5], which shows the issue is not limited to speculative loads.

TABLE II: List of currently disclosed attacks categorized by nature, capabilities and constraints. A checkmark (✓) under **capabilities** reports an attack demonstrated in the literature. A checkmark under the **constraints** represents a requirement to perform the attack. We report supervisor in both Intra/Cross- address space scenarios both pre- and post- KPTI [9].

Attacks	Leak cause	Exception deferral	Capabilities						Constraints	
			Intra-address space			Cross-address space			Victim cooperation	Valid address translation
			SB	SP†	SGX	XPC	SP‡	XVM		
Control Speculation										
BCB{S} [2], [5]	Direct branch		✓						✓	✓
BTI [2], [70]	Indirect branch		✓		✓				✓	✓
RSB [4], [71]	Return stack		✓		✓				✓	✓
Data Speculation										
SSB [72]	Memory Disambiguation		✓	✓					✓	✓
RDCL [1] RSRR [73]	L1D	✓	✓	✓						✓
LazyFP [74]	FPU	✓				✓		✓		
L1TF [75]	L1D	✓	✓	✓	✓		✓	✓		✓
RIDL	LFB		✓	✓	✓	✓	✓	✓		

SB = Sandbox, SP† = Supervisor pre-KPTI, SP‡ = Supervisor post-KPTI, XPC = Cross-process, XVM = Cross Virtual Machines

TABLE III: List of existing mitigations against currently disclosed speculative execution attacks grouped based on the nature of the defense.

Attacks	Inhibit trigger								Hide Secret					
	LFENCE [76]	IRBS, IBPB STIBP [76]	SSBD [44]	RSRR Fix [73]	Retpoline [6]	RSB Filling [77]	Eager FPU [74]		KPTI [9]	Array Index Masking [7]	Multi-Process Isolation [63]	L1D Flushing [78]	PTE Inversion [78]	HyperClear [50]
Control Speculation														
BCB{S} [2], [5]	G									G	SB			
BTI [2], [70]		G			G	G					SB			
RSB [4], [71]											SB			
Data Speculation														
SSB [72]	G		G								SB			
RDCL [1]									SP†					
RSRR [73]				G					SP†					
LazyFP [74]						G								
L1TF [75]												G	SB, SGX	XVM
RIDL														

G = Generic, SB = Sandbox, SP† = Supervisor pre-KPTI, XPC = Cross-process, XVM = Cross Virtual Machines,

Indirect branches and calls: These branches are also targets of speculative execution optimizations. The *Branch Target Buffer* (BTB) is a unit embedded in modern CPUs that stores a mapping between the source of a branch or call instruction and its likely destination. An attacker running on the same physical core of the victim can poison the BTB to perform attacks known as *Branch Target Injection* (BTI). The attacker pollutes the buffer by injecting arbitrary entries in the table to divert the victim's indirect branches to the target of interest—within the victim address space. No checks are performed on the `pid`, hence the possibility of cross-process mistraining. This makes it possible to build *speculative* code-reuse (e.g., ROP) attacks to leak data. Branch target injection has been demonstrated effective to escape sandboxed environments (e.g., JavaScript sandbox) [2], to build cross-process attacks [2] and to leak data from SGX enclaves [70].

Return speculation: The use of the BTB is inefficient for the prediction of *return* instructions, as functions may have many different call sites. Therefore, modern processors employ a *Return Stack Buffer* (RSB), a hardware stack buffer to which the processor pushes the return address whenever it executes a *call* instruction. Whenever the processor stumbles upon a *return* instruction, it pops the address from the top of the RSB to predict the return point of the function, and it executes the instructions along that path speculatively. The RSB misspeculates when the return address value in the RSB does not match the one on the software stack. Unfortunately, the RSB consists of a limited number of entries and employs a round robin replacement policy. As a result, an attacker can overflow the RSB to overwrite the "alleged" return address of a function and speculatively execute the code at this address. Researchers have reported RSB attacks against sandboxes and enclaves [4], [71].

Constraints: Control speculation attacks, while powerful,

are only effective in intra-address space attacks (e.g., sandboxes). Furthermore, their exploitation requires (some) cooperation from the victim's code. In situations where the attacker can generate code inside the victim (e.g., JIT compilation inside a browser, eBPF in the Linux kernel) it is easy to meet this constraint. In the other cases (e.g., enclaves or Linux kernel without eBPF), this constraint is harder to meet. The attacker needs to mount *confused-deputy* attacks that lure the victim component into speculatively executing specific "gadgets", making exploitation more difficult. Perhaps more importantly, this class of attacks can be mitigated in software using either compiler support for emitting safe code or manually stopping speculation when deemed dangerous.

B. Data Speculation

Data speculation is the second type of speculative execution. This type of speculation does not divert the control flow of the application, but instead speculates on the value to be used. As discussed in Section VII-A, manipulating control flow is not enough to build effective cross-privilege and cross-address space attacks. Attackers can overcome these constraints by taking advantage of data speculation.

Speculative Store Bypass (SSB) [72] takes advantage of the address prediction performed by the *Memory Disambiguator*. This unit is in charge of predicting read-after-write hazards. If the prediction fails, the attacker may be able to leak stale L1 cache lines previously stored at that address. However, this attack provides a small window of exploitation and works only within intra-address space boundaries, making it hard to exploit in practice.

Exception deferral: To bypass this limitation and allow cross-boundary leaks, researchers identified a new class of data speculation attacks that we refer to as *exception deferral* attacks. A similar distinction was previously made in the literature [5] under the nomenclature of *exception speculation*. However, as explained in Section II, speculation represents a misleading terminology of the actual issue under scrutiny. The CPU does not perform any type of speculation on the validity of the operation. It simply executes instructions speculatively out-of-order and eventually retires them in-order. The flaw of this design is that the Retirement Unit is officially in charge of handling CPU exceptions. Thus, an attacker can perform loads that trigger exceptions (e.g., *Page Faults*) during the speculative execution window, but such loads will not fault until the Retirement Unit performs the compulsory checks.

Multiple attacks have exploited CPUs' exception deferral in order to circumvent different security checks. RDCL [1] (known as Meltdown) and RSRR [73] exploited the deferral of a *page fault* (`#PF`) exception caused by the presence of the supervisor bit in order to read privileged kernel memory and *Model Specific Registers* (MSRs). LazyFP [74] took advantage of the deferral of the *Device not available* (`#NM`) exception to leak *Floating Point Units* (FPUs) register state and break cryptographic algorithms. Foreshadow [75] disclosed how the deferral of a *Terminal Fault* (`#TF`) generated by a failed check on the present or reserved bits of a PTE allows leaking arbitrary contents from the L1 cache. Given that Foreshadow operates on physical memory addresses, it can leak information across privilege and address space boundaries breaking kernel, enclaves, and VM isolation. Crafting Foreshadow attacks, however, requires control over addresses residing in PTEs as we discuss next.

Constraints: Data speculation allows attackers to operate on data they are not supposed to have access to. Furthermore, when combined with exception deferral, they gain the capability of not relying on victim code to leak data. With RDCL, for example, attackers can directly read from kernel memory without relying on "gadgets" in the kernel code. Most of these attacks, however, are still limited by the necessity of a valid address translation. That is, a valid (and known) address to leak the data from. For instance, in the case of Foreshadow, the attacker can theoretically read any arbitrary cache line in the L1d cache. However, since L1d cache lines are physically tagged, the attacker needs control over virtual-to-physical address mappings (PTEs). This constraint is easily met in situations where the attacker controls these mappings, such as inside guest VMs or SGX enclaves. In the other cases, this constraint is harder to meet, such as when attacking the kernel or another user process.

C. Comparing with RIDL

While RIDL still falls under the umbrella of data speculation attacks, it presents a unique feature that makes it stand out among the other attacks: the ability to induce leaks that are completely agnostic to address translation. All the other attacks other than LazyFP [74] (which is limited to leaking stale floating point registers) require a valid address for performing tag checks before retrieving the data. If this check fails, the speculation aborts. On the other hand, in the case of RIDL, the attacker can access any *in-flight* data currently streaming through internal CPU buffers without performing any check. As a result, address space, privilege, and even enclave boundaries do not represent a restriction for RIDL attacks.

VIII. Existing defenses

In response to the plethora of attacks described in Section VII, hardware and software vendors have been struggling to catch up with mitigations that can safeguard vulnerable systems. In this section, we perform an exhaustive analysis of all the existing state-of-the-art mitigations, pinpointing the current shortcomings of such solutions when applied to the RIDL family of attacks.

These mitigations can operate at three different layers: ① inhibiting the trigger of the speculation, ② protecting the secret the attacker is trying to disclose, or ③ disrupting the channel of the leakage. We focus on the first two classes, which are specific to speculative execution attacks; the third typically applies to any timing side-channels (e.g., disabling high-precision timers in browsers [63]). We summarize all the currently deployed mitigations and their effects on currently-known attacks in Table III.

A. Inhibiting the trigger

To protect against control speculation attacks, vendors have released mitigations that prevent the hardware from executing (speculatively) unsafe code paths. For instance, Intel released a microcode update with three new capabilities: IRBS, STIBP and IBPB to prevent indirect branch poisoning instructions and to protect against BTI attacks [76]. Another suggested mitigation uses the `lfence` instruction to restrict control speculation. This can be applied as a compiler-based defense, mitigating multiple families of attacks. ① To protect against BCB attacks, the compiler inserts an `lfence` instruction after conditional branches to stop the BPU from speculating on the code path taken. ② To protect against BTI attacks, the `lfence` instruction is introduced as part of the *Retpoline* [6] mitigation. Researchers have also suggested extending Retpoline to guard `ret` instructions and prevent RSB attacks [71]. The Retpoline mitigation converts each indirect jump into a direct call to a stub function, that returns to the destination of the initial indirect branch. This is achieved by altering the stack, replacing the return address of the function. Since return instructions also trigger speculation, an `lfence` loop is inserted at the expected return site of the stub, to inhibit further code execution. Retpoline can also perform RSB filling [77]. This is required for Intel architectures newer than Haswell where, in case of an empty RSB, the BTB provides the speculation address.

Software mitigations such as Retpoline do not apply for data speculation attacks since there is no need to (speculatively) divert the control flow of the application. As such, most defenses against data speculation have been in the form of microcode updates, such as:

- SSBD: *Speculative Store Bypass Disable* adds an MSR which can be used to prevent loads from executing before addresses of previous stores are known [44].
- RSRR fix: Intel's mitigation for Rogue System Register Reads patches `rdmsr` to avoid speculative L1 loads of MSR data for unprivileged users [73].

Finally, to protect against LazyFP, it suffices to enable Eager FPU context switching. This restores FPU register state when performing a context switch, preventing speculative execution on stale register contents [74].

B. Protect the secret

When preventing the CPU from speculating becomes unfeasible, the other solution is to conceal the sensitive information from the attacker. Defenses falling under this category are clearly context sensitive. That is, they highly depend on the environment they get deployed on since different environments secure different secrets. A primary example is *Kernel Page Table Isolation* (KPTI) [9]. KPTI was effectively the first mitigation deployed against speculative execution attacks and was introduced to protect the Kernel against RDCL (i.e., Meltdown) by separating kernel and user address spaces.

A similar compartmentalization approach was then deployed in other environments. ① Array index masking was deployed in the kernel and in sandboxed environments to protect against intra-address space BCB attacks. ② Multi-process isolation, similarly to KPTI, protects sandboxed environments from cross-domain attacks (e.g., JavaScript VMs) by generating a different process for every origin—hence a different address space.

These mitigations were considered effective until the disclosure of the Foreshadow attack. Foreshadow relaxed the requirement of victim cooperation for cross-address space attacks by leaking any data present in the L1d cache. Protecting against this new class of attacks requires stronger solutions targeting physical addresses. Two instances of such mitigations are PTE inversion and L1d flush [78]. PTE inversion protects kernel and enclave memory from being leaked through the L1d cache by scrambling the physical address in the PTE when a page is marked as non-present. L1d flush removes any secret information from the cache during a context switch, making it impossible to retrieve any data. The latter is part of a set of mitigations intended for environments such as the cloud, where an attacker may have control of PTE contents.

Another example of such solutions is HyperClear [50], deployed by Microsoft to safeguard Hyper-V. The mitigation consists of three units: ① The *Core Scheduler*, which performs safe scheduling of sibling logical processors by allocating resources for a single VM on the same physical processor. ② Address space isolation per virtual processor, which limits the hypervisor access to memory only belonging to the VMs running on the same physical core—preventing cross-VM leaks. ③ Sensitive data scrubbing, which protects nested hypervisors from leaking sensitive data. This is done by zeroing the latter before switching VMs and avoiding the performance impact of a complete L1d flush. Similar solutions have been deployed on other hypervisors such as KVM [8].

C. Defenses vs. RIDL

In Section VI, we reported the results of all our proof-of-concept exploits on fully patched systems with the latest microcode updates. As the positive results demonstrate, the currently deployed mitigations fail to protect against RIDL attacks. As we discussed in Section VIII-A, mitigations for data speculation attacks usually rely on microcode patches. Since the existing defenses trying to inhibit speculation do not account for the Line Fill Buffer, RIDL is not impacted by any of them.

On the other hand, defenses aiming at protecting the secret fail at defending from RIDL attacks for a different reason: they all consider a valid address a strict requirement. RIDL demonstrates that not all the sources of data speculation rely on this assumption. Our results for the i9-9900K show the risk of relying on "spot" mitigations in hardware; although the address-based page faults used by Meltdown-style attacks have been mitigated in silicon, RIDL attacks using other exceptions continue to work. Furthermore, it demonstrates for the first time a cross-address space and cross-privilege attack that relies only on in-flight, CPU-internal data, demonstrating the latent danger introduced by the related microptimizations.

IX. New Mitigations

The response to the disclosure of speculative execution attacks has so far been the deployment of spot mitigations in software before mitigations become available in hardware [80]. For example, for Meltdown, the first deployed software mitigation (i.e., KPTI) was the separation of address spaces between user space and kernel space by the operating system. While effective, on top of increasing complexity in the kernel, KPTI has been shown to have performance penalties under certain workloads [81]. We now describe how this spot mitigation approach is not well-suited for the LFB variant of RIDL presented in this paper.

Mitigating RIDL in software. Since sensitive information can be leaked from sibling hardware threads, it is clear that SMT must be disabled to mitigate RIDL. However, it is still possible to leak sensitive information from another privilege level within a single thread (as some of our exploits demonstrated), including information from internal CPU systems such as the MMU. To protect sensitive information in the kernel or a different address space, the kernel needs to flush the LFBs before returning to userland similar to the L1 flush in the Foreshadow mitigation. Similarly, the hypervisor needs to flush the LFBs before switching to VM execution. In the case of hardware-based components such as SGX or the MMU, the LFB flushing cannot be easily done in software.

Perhaps more importantly, while Intel could provide a L1 flush mechanism via a microcode update, it is not clear whether it is possible to expose a similar mechanism for flushing the LFBs. Furthermore, even if such a mechanism was possible, its cost will likely be even more expensive than the L1 flush on every context switch. Remember that the entire L1 cache needs to be flushed first since the entries go through the LFBs. After that, the mechanism needs to wait until the LFBs are drained before safely resuming the execution. We believe that such a mechanism will be too expensive to be useful in practice.

Moving forward. In this paper, we focused on speculation done on LFB entries. However, we believe there are several other sources of in-flight data—especially given decades of performance optimizations in the CPU pipeline. Furthermore, as discussed in this section, because these optimizations are applied deeply in the CPU pipeline, spot mitigations will likely be expensive. Moving forward, we see two directions for mitigating RIDL: 1) As Intel could release a microcode update that mitigated SSB by completely disabling speculative store forwarding, we believe it should make a similar mitigation possible for all possible sources of speculation when applying micro-optimizations. It will then be up to system software to decide which optimizations to turn off until hardware mitigations become available. 2) Finding all instances of RIDL will likely take a long time due to the complexity of these micro-optimizations. Hence, rather than spot mitigations that are often ineffective against the next discovered attack, we need to start the development and deployment of more fundamental mitigations against the many possible classes of speculative execution attacks.

X. Conclusion

We presented RIDL, a new class of speculative execution vulnerabilities able to leak arbitrary, address-agnostic in-flight data from normal execution (without branches or errors), including sandboxed execution (JavaScript in the browser). We showed RIDL can be used to perform attacks across arbitrary security boundaries and presented real-world process-, kernel-, VM-, and SGX-level exploits. State-of-the-art mitigations against speculative execution attacks (including the in-silicon mitigations in Intel's recent CPUs) are unable to stop RIDL, and new software mitigations are at best non-trivial. RIDL puts into question the current approach of "spot" mitigations for individual speculative execution attacks. Moving forward, we believe we should favor more fundamental "blanket" mitigations over these per-variant mitigations, not just for RIDL, but for speculative execution attacks in general.

Disclosure

The authors from VU Amsterdam (VUSec) submitted PoC exploits for the RIDL class of vulnerabilities to Intel on September 12, 2018. Intel immediately acknowledged the vulnerability and rewarded RIDL with the Intel Bug Bounty (Side Channel) Program. Since then, Intel led the disclosure process, notifying all the affected software vendors and other hardware vendors potentially susceptible to similar issues (see details below). VUSec submitted the end-to-end analysis presented in this paper including all the exploits (except the one in Section VI-D) to IEEE Symposium on Security & Privacy on November 1, 2018.

Giorgi Maisuradze independently discovered the same class of vulnerabilities in June 2018 as an intern in a side-channel project at Microsoft Research. The findings were reported to Intel via the Microsoft Security Response Center. Section VI-D is entirely based on his findings.

Volodymyr Pikhur independently discovered and reported a RIDL-class exploit (L1TF mitigation bypass over uncached memory) to Intel on August 25, 2018. Dan Horea Lutas' team at Bitdefender reported an issue related to the RIDL vulnerabilities to Intel on August 17, 2018.

Statements that we received from CPU vendors about RIDL are available in Appendix A.

Acknowledgements

We would like to thank our shepherd, Hovav Shacham, and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by the United States Office of Naval Research (ONR) under contracts N00014-17-1-2782 and N00014-17-S-B010 "BinRec", by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] 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 *USENIX Security'18*.

[2] 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 *S&P'19*.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security'18*.

[4] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *USENIX WOOT'18*.

[5] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and Defenses," in *arXiv'18*.

[6] P. Turner, "Retpoline: a Software Construct for Preventing Branch Target Injection," https://support.google.com/faqs/answer/7625886, Jan 2018.

[7] F. Pizlo, "What Spectre and Meltdown Mean For WebKit," https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, Jan 2018.

[8] "Linux: L1TF - L1 Terminal Fault," https://www.kernel.org/doc/html/latest/admin-guide/l1tf.html#mitigation-control-kvm Retrieved 15.10.2018.

[9] "KPTI - Linux Documentation," https://www.kernel.org/doc/Documentation/x86/pti.txt Retrieved 15.10.2018.

[10] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *USENIX Security'18*.

[11] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games–Bringing Access-Based Cache Attacks on AES to Practice," in *S&P'11*.

[12] Y. Yarom and K. Falkner, "FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." in *USENIX Security'14*.

[13] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *Cryptographers Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[14] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES," in *S&P'15*.

[15] M. Kayaalp, D. Ponomarev, N. Abu-Ghazaleh, and A. Jaleel, "A High-Resolution Side-Channel Attack on Last-Level Cache," in *DAC'16*.

[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P'15*.

[17] R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, 1967.

[18] M. E. Thomadakis, "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms," 2011.

[19] D. W. Clift, D. D. Boggs, and D. J. Sager, "Processor with Registers Storing Committed/Speculative Data and a RAT State History Recovery Mechanism with Retire Pointer," Oct 2003, US Patent 6,633,970.

[20] D. D. Boggs, S. Weiss, and A. Kyker, "Branch Ordering Buffer," Sep 2004, US Patent 6,799,268.

[21] J. M. Abramson, D. B. Papworth, H. H. Akkary, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, and P. D. Madland, "Out-Of-Order Processor With a Memory Subsystem Which Handles Speculatively Dispatched Load Operations," Oct 1995, US Patent 5,751,983.

[22] G. N. Hammond and C. C. Scafidi, "Utilizing an Advanced Load Address Table for Memory Disambiguation in an Out of Order Processor," Dec 2003, US Patent 7,441,107.

[23] H.-S. Kim, R. S. Chappell, C. Y. Soo, and S. T. Srinivasan, "Store Address Prediction for Memory Disambiguation in a Processing Device," Sep 2013, US Patent 9,244,827.

[24] V. Mekkat, O. Margulis, J. M. Agron, E. Schuchman, S. Winkel, Y. Wu, and G. Dankel, "Method and Apparatus for Recovering From Bad Store-To-Load Forwarding in an Out-Of-Order Processor," Dec 2015, US Patent 9,996,356.

[25] T. Kurts, Z. Wayner, and T. Bojan, "Apparatus and Method for Bus Signal Termination Compensation During Detected Quiet Cycle," Dec 2002, US Patent 6,842,035.

[26] A. Koker, T. A. Piazza, and M. Sundaresan, "Scatter/Gather Capable System Coherent Cache," May 2013, US Patent 9,471,492.

[27] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling," in *ISCA'99*.

[28] H. Akkary, J. M. Abramson, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, P. D. Madland, M. S. Joshi, and B. E. Lince, "Methods and Apparatus for Caching Data in a Non-Blocking Manner Using a Plurality of Fill Buffers," Oct 1996, US Patent 5,671,444.

[29] H. Akkary, J. M. Abramson, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, P. D. Madland, M. S. Joshi, and B. E. Lince, "Cache Memory System Having Data and Tag Arrays and Multi-Purpose Buffer Assembly With Multiple Line Buffers," Jul 1996, US Patent 5,680,572.

[30] S. Palanca, V. Pentkovski, S. Tsai, and S. Maiyuran, "Method and Apparatus for Implementing Non-Temporal Stores," Mar 1998, US Patent 6,205,520.

[31] S. Palanca, V. Pentkovski, and S. Tsai, "Method and Apparatus for Implementing Non-Temporal Loads," Mar 1998, US Patent 6,223,258.

[32] J. M. Abramson, H. Akkary, A. F. Glew, G. J. Hinton, K. G. Koningsfeld, and P. D. Madland, "Method and Apparatus for Performing Load Operations in a Computer System," Dec 1997, US Patent 5,694,574.

[33] M. Bodas, G. J. Hinton, and A. F. Glew, "Mechanism to Improved Execution of Misaligned Loads," Dec 1998, US Patent 5,854,914.

[34] J. M. Abramson, H. Akkary, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, and P. D. Madland, "Method and Apparatus for Blocking Execution of and Storing Load Operations during their Execution," Mar 1999, US Patent 5,881,262.

[35] A. Glew, N. Sarangdhar, and M. Joshi, "Method and Apparatus for Combining Uncacheable Write Data Into Cache-Line-Sized Write Buffers," Dec 1993, US Patent 5,561,780.

[36] M. S. Joshi, A. F. Glew, and N. V. Sarangdhar, "Write Combining Buffer for Sequentially Addressed Partial Line Operations Originating From a Single Instruction," May 1995, US Patent 5,630,075.

[37] S. Palanca, V. Pentkovski, N. L. Cooray, S. Maiyuran, and A. Narang, "Method and System for Optimizing Write Combining Performance in a Shared Buffer Structure," Mar 1998, US Patent 6,122,715.

[38] J. D. Dundas, "Repair of Mis-Predicted Load Values," Mar 2002, US Patent 6,883,086.

[39] Y.-K. Chen, C. J. Hughes, and J. M. Tuck, III, "System and Method for Cache Coherency in a Cache With Different Cache Location Lengths," Dec 2004, US Patent 7,454,576.

[40] D. Kanter, "Intel's Haswell CPU microarchitecture," 2012.

[41] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," Jun 2016.

[42] Intel, "Write Combining Memory Implementation Guidelines," 1998.

[43] A. F. Glew and G. J. Hinton, "Method and Apparatus for Processing Memory-Type Information Within a Microprocessor," Dec 1996, US Patent 5,751,996.

[44] Intel, "Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115," https://software.intel.

com/security-software-guidance/software-guidance/speculative-store-bypass Retrieved 15.10.2018.

[45] S. Palanca, V. Pentkovski, S. Maiyuran, L. Hacking, R. A. Golliver, and S. S. Thakkar, "Synchronization of Weakly Ordered Write Combining Operations Using a Fencing Mechanism," Mar 1998, US Patent 6,073,210.

[46] S. Palanca, S. A. Fischer, S. Maiyuran, and S. Qawami, "MFENCE and LFENCE Micro-Architectural Implementation Method and System," Jul 2002, US Patent 6,651,151.

[47] L. E. Hacking and D. Marr, "Synchronization of Load Operations Using Load Fence Instruction in Pre-Serialization/Post-Serialization Mode," Feb 2001, US Patent 6,862,679.

[48] L. E. Hacking and D. Marr, "Globally Observing Load Operations Prior to Fence Instruction and Post-Serialization Modes," Jan 2004, US Patent 7,249,245.

[49] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of the Linux symposium*, vol. 1. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[50] D. Hepkin, "Hyper-V HyperClear Mitigation for L1 Terminal Fault," https://blogs.technet.microsoft.com/virtualization/2018/08/14/hyper-v-hyperclear/, Aug 2018.

[51] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS'16*.

[52] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *SEC'16*.

[53] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA'16*.

[54] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *SP'16*.

[55] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," in *BHUS'15*.

[56] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P'18*.

[57] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P'19*.

[58] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC'16*.

[59] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID'18*.

[60] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS'17*.

[61] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX'17*.

[62] T. C. Projects, "Mitigating Side-Channel Attacks," https://www.chromium.org/Home/chromium-security/ssca Retrieved 31.12.2018.

[63] M. Bynens, "Untrusted Code Mitigations," https://v8.dev/docs/untrusted-code-mitigations, Jan 2018.

[64] L. Wagner, "Mitigations Landing for New Class of Timing Attack," Jan 2018, https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/ Retrieved 31.12.2018.

[65] T. Ritter, "Firefox - Fuzzy Timers Changes," Oct 2018, https://hg.mozilla.org/mozilla-central/rev/77626c8d6bee.

[66] D. Kohlbrenner and H. Shacham, "Trusted Browsers for Uncertain Times." in *USENIX Security'16*.

[67] "Re-enable SharedArrayBuffer + Atomics," https://bugs.chromium.org/p/chromium/issues/detail?id=821270.

[68] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is Here to Stay: An Analysis of Side-Channels and Speculative Execution."

[69] "Process Isolation in Firefox," https://mozilla.github.io/firefox-browser-architecture/text/0012-process-isolation-in-firefox.html.

[70] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution."

[71] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution using Return Stack Buffers," 2018.

[72] J. Horn, "Speculative Store Bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, May 2018.

[73] Intel, "Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115," https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read, May 2018.

[74] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," 2018.

[75] 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.

[76] Intel, "Intel Analysis of Speculative Execution Side Channels: White paper," Jan 2018, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf Retrieved 15.10.2018.

[77] D. Woodhouse, "x86/retpoline: Fill RSB on Context Switch for Affected CPUs," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c995efd5a740d9cbafbf58bde4973e8b50b4d761, Jan 2018.

[78] Intel, "Deep Dive: Intel Analysis of L1 Terminal Fault," https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault Retrieved 15.10.2018.

[79] J. Horn, "Reading Privileged Memory with a Side-channel," https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, Jan 2018.

[80] "Intel Announces 9th Generation Core CPUs, Eight-Core Core i9-9900K," https://www.tomshardware.com/news/intel-9th-generation-coffee-lake-refresh,37898.html.

[81] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs," in *ATC'18)*.

Appendix A
Statements from CPU Vendors

A. Statement from Intel

"We have disclosed details about the issue described in VU Amsterdam's paper with multiple parties in the computing ecosystem who have the ability to help develop mitigations. This includes operating system vendors such as MSFT and Redhat, hypervisor vendors such as VMWare and Citrix, silicon vendors or licensors such as AMD and ARM, select operating system providers or maintainers for open source projects, and others. These disclosures have been conducted under coordinated vulnerability disclosure for purposes of architecting, validating, and delivering mitigations, and all parties agreed to mutual confidentiality and embargo until 10AM PT May 14, 2019".

B. Statement from AMD

"After reviewing the paper and unsuccessfully trying to replicate the issue, AMD believes its products are not vulnerable to the described issue".

C. Statement from ARM

"After reviewing the paper and working with architecture licensees we are not aware of any Arm-based implementations which are affected by this issue. We thank VU Amsterdam for their research".

Appendix B
Extended Results

Figure 9 shows a more complete diagram of the Intel Skylake microarchitecture.

Figure 10 shows a screenshot of our tool to test for existing vulnerabilities as well as RIDL, to check what mitigations are available and enabled and to provide a general overview including the installed microcode version. We will release this tool as open source on May 14, as well as provide binaries of this tool for various platforms including Microsoft Windows and Linux.

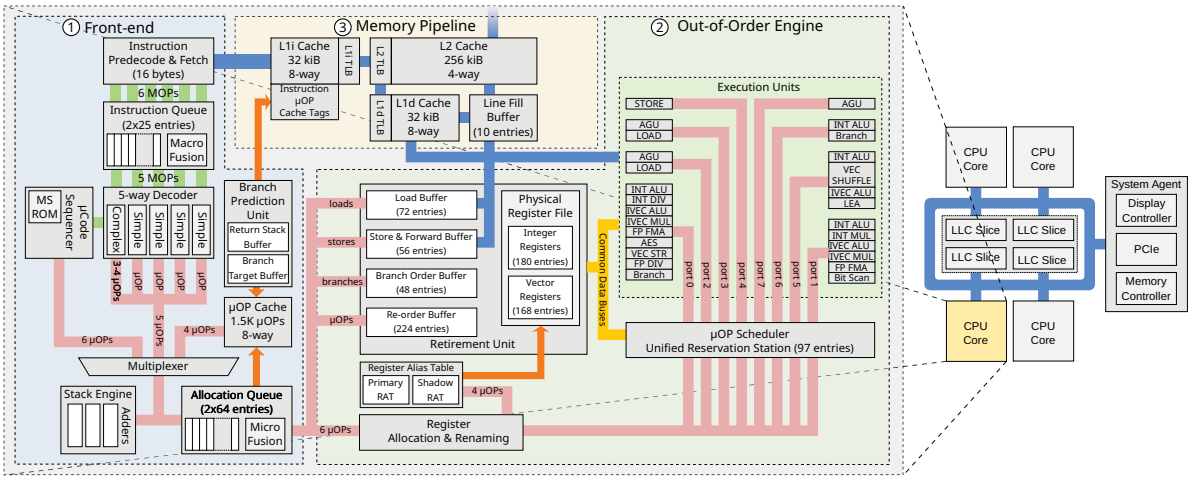

Fig. 9: a full overview of the Intel Skylake microarchitecture.

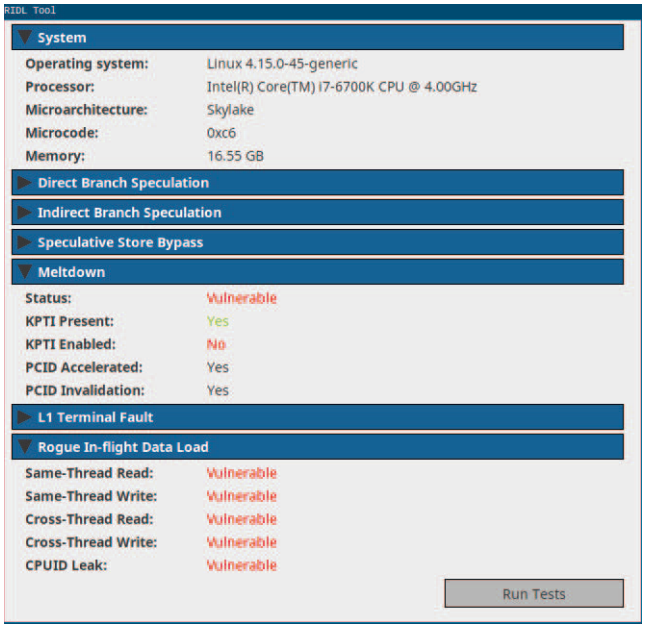

Fig. 10: A screenshot of our tool to test for vulnerabilities and mitigations including RIDL.