Downfall: Exploiting Speculative Data Gathering

Daniel Moghimi

University of California, San Diego, CA, USA

Abstract

We introduce *Downfall* attacks, new transient execution attacks that undermine the security of computers running everywhere across the internet. We exploit the *gather* instruction on high-performance x86 CPUs to leak data across boundaries of user-kernel, processes, virtual machines, and trusted execution environments. We also develop practical and end-to-end attacks to steal cryptographic keys, program's runtime data, and even data at rest (arbitrary data). Our findings, exploitation techniques, and demonstrated attacks defeat all previous defenses, calling for critical hardware fixes and security updates for widely-used client and server computers.

1 Introduction

Over the past few years, computer manufacturers and cloud providers have deployed several software and microcode updates to defend against transient execution attacks [9, 19, 23, 45]. Transient execution attacks exploit software-accessible side channels to reveal vulnerabilities in the hardware that enable attackers to steal data across security boundaries [2, 5, 31, 32, 34, 48, 49, 52, 55, 57, 58, 60].

For most critical attacks including Meltdown [34], Foreshadow [57, 62], MDS [5, 52, 60], and LVI [58], users have to upgrade to new hardware that have fixes in the silicon. Without hardware fixes, users have to rely on software workarounds such as KPTI [15, 33], flushing core-private resources across context switching, disabling performance features such as simultaneous multithreading [19, 23], and compiler-based code transformations [25]. Unfortunately, these workarounds are either tremendously expensive [46, 47] or ineffective in completely eradicating data leakage [24, 56].

The common belief is that CPU manufacturers have eliminated several classes of attacks and know how to build processors resistant to data leaks due to the transient side effect of *invalid* memory accesses. For example, Intel, the prominent CPU manufacturer, has developed hardware fixes for Meltdown and Foreshadow in their 9th-generation CPUs and MDS and LVI in their 10th-generation Cascade Lake CPUs. Although it took several iterations to address all variants [1, 21, 24, 38, 49], new CPUs are secure against all previously-disclosed transient execution attacks due to invalid memory access speculation, e.g., Meltdown, MDS and LVI. Consequently, computers no longer require expensive software workarounds or microcode patches on these new CPUs. We challenge this status quo:

1. Gather Data Sampling (§3) We introduce *Gather Data Sampling (GDS)* that exploits the *gather* instruction to steal stale data from previously-undisclosed CPU components; *SIMD register buffers*. Since various memory operations share these buffers, GDS enables attackers to steal data from other security domains (e.g., across user-kernel, process, and VM boundaries). As a result, the latest Intel Ice Lake and Tiger Lake CPUs that claim to be resistant to data leaks expose users' data. Also, mitigations for earlier CPUs that rely on flushing microarchitectural buffers are ineffective since they do not flush the SIMD register buffers.

2. Cross-Process Covert Channel (§4) We develop cross-process covert-channel attacks and evaluate their performance across recent CPU generations. We extend previous multi-byte data sampling techniques to leak up to 22 bytes simultaneously per each execution of the attack on the latest Intel CPUs, enabling a high-speed data bandwidth across processes. Our results show attackers can construct a cross-process channel with up to 5.7 kB/s speed.

3. Stealing Cryptographic Keys (§5) We demonstrate end-to-end cross-VM attacks against widely-used modern cryptographic software that uses SIMD instructions for efficient and secure (constant-time) execution. We steal AES-128 and AES-256 keys from the off-the-shelf OpenSSL command line tool for encrypting data. Unlike previous such attacks, our attack is simple and reliable; In less than 10 seconds, we steal AES round keys, 8 bytes at a time, and combine them to break AES without any cryptanalysis, source code analysis, or any data analysis tricks required by previous attacks [52, 60].

4. Stealing Arbitrary Data (§6) We discover that GDS can steal data from *no-op* operations that do not do anything architecturally. Masked memory operations, when the masked bit is unset, and the streaming data copy instructions, when the data copy size is zero, should not execute architecturally. But, we found that Intel CPUs transiently prefetch data into the leaky SIMD register buffers when such no-op operations are

executed. As a result, GDS leaks data that is not accessed by the program. Based on this discovery, we develop attacks to steal arbitrary data from the Linux kernel. Since the Meltdown finding, this is the first hardware attack that enables a user to steal arbitrary data from the OS Kernel without relying on software vulnerabilities or Spectre gadgets [31].

5. Gather Value Injection (§7) We introduce the *gather value injection (GVI)* attack by combining GDS with the LVI [58] technique. Like LVI, we can turn the *gather* data leaks into microarchitectural data injections. Unlike LVI based on MDS data leak, GVI based on GDS does not rely on faults or other architectural behavior to inject stale microarchitectural states to the *gather* instruction. To exploit GVI, an attacker finds a *gather* instruction in the victim code followed by data-dependent operations, similar to a Spectre gadget. We introduce several GVI gadgets and exploit them to steal data from a victim process.

6. Breaking Intel SGX (§8) We show that Intel SGX is additionally vulnerable to GDS (therefore, GVI) attacks. We leak data from the secure enclave even on earlier CPUs that implicitly flushes microarchitectural buffers across SGX context switching. Ultimately, we develop an end-to-end attack to steal the SGX sealing key at its highest security configuration. Exposing the sealing key would allow an attacker to compromise SGX protections; while microcode mitigation could potentially address the issue, the mitigation can only be attested with key revocation and a TCB recovery.

7. Mitigation (§9) We discuss defense and testing techniques to mitigate Downfall attacks (GDS and GVI). First, we discuss potential software workarounds and hardware fixes and their implications. Then, we extend the Transynther tool [40] to test CPUs against GDS. We hope our results inspire CPU manufacturers to deploy automated testing approaches, e.g., fuzzing, to mitigate future instances of such vulnerabilities.

Experimental setup The experiments in this paper run on computers with the CPUs listed in table 1 and the Ubuntu 20.04.5 LTS running Linux kernel 5.15.0-48-generic.

CPU	Generation	Stepping	Mirocode
Core i7-1165G7	Tiger Lake	1	0xa4
Xeon Silver 4314	Ice Lake Server	6	0xd000363
Xeon(R) Gold 6230	Cascade Lake	7	0x5002f01
Core i7-8650U CPU	Kaby Lake	10	0xf0

Table 1: Tested CPUs.

Threat model We assume two scenarios of attacker and victim running on the same CPU core: (1) Concurrently running on sibling threads of the same core (Sections 4 to 7), (2) Context switching on the same CPU thread (Sections 4 and 8). We demonstrate these two scenarios using CPU core pinning following previous work [52, 60]. In section 8, the attacker also controls the OS following the well-understood OS adversary model that SGX expects [37, 59].

Responsible disclosure We reported our findings to Intel on August 24, 2022. They acknowledged our findings (CVE-2022-40982) and confirmed that previous hardware fixes and software mitigation do not mitigate Downfall attacks. Intel asked for our findings to be under embargo until August 2023. In ongoing discussions with Intel, they confirmed that they will mitigate Downfall with a microcode update, which will be deployed concurrently with the public release of our results. This microcode update presumably blocks transient results of *gather* instructions to prevent attacker code from observing speculative results of gather loads. We have not been provided further technical details about these updates at this time.

Artifact We will publish analysis tools, proof-of-concept attacks, and data sets and provide further guidelines on the impact of our findings at `downfall.page`.

2 Data Exposure in Superscalar Computers

This section covers background information on superscalar computing and relevant data leak vulnerabilities.

2.1 Superscalar Computing

High-performance CPUs that we use every day, like the Intel *Core* and *Xeon*, have multiple execution cores and high-speed memory units and support several features to execute parallel processes while enforcing isolation and security efficiently:

Virtual address space The CPU isolates the memory of each process using an isolated per-process virtual address space. The virtual address the software uses to access the memory will be translated to a page table entry (PTE). The CPU uses the PTE, which has the physical location of the data, access control, and status bits, to access the memory location and enforce access control. Segregated mapping of virtual-to-physical addresses blocks separate processes from accessing each other's memory, and the access control blocks user processes from accessing the OS kernel memory.

Context switching In preemptive multitasking, the OS instructs the CPU to switch to another process (context switching) after executing the current process for a limited time. The CPU also switches context when it receives an interrupt (e.g., local timer interrupt) or when a process collaboratively switches to another execution domain (e.g., switching to/from the OS kernel). Upon context switching, the virtual address space and register context have to be switched, so the new process will not have access to the memory and registers of previous processes.

Simultaneous multithreading Simultaneous multithreading (SMT) allows multiple threads to execute on the same core simultaneously while sharing the same hardware resources. The software perceives these CPU threads as separate virtual

CPU cores; each thread is architecturally isolated from others and only accesses its allocated virtual address space and registers. Intel CPUs support two simultaneous threads (virtual CPU cores) per physical core.

Speculative execution Speculative execution allows the CPU core to execute instructions from a single execution thread in parallel. When an instruction depends on prior unresolved operations, the CPU speculatively executes it based on some prediction. When, hopefully rarely, the prediction is incorrect, the CPU flushes incorrect executions and re-executes them to get correct results. Speculative execution is architecturally invisible to the software, but its side effects can be observed [31, 34].

Single instruction multiple data (SIMD) SIMD enables data-level parallelism; a SIMD instruction computes the same function multiple times with different data. The maximum number of parallel computations depends on the register size and data type. AVX2 and AVX-512 (introduced in Cascade Lake) are the key SIMD extensions for x86, in which AVX-512 supports 512-bit vector registers and can compute up to 16 $\times$ double-word (dword) values or 8 $\times$ quad-word (qword) values.

Cache The CPU has a last-level cache (LLC) shared across execution cores and an interconnect bus that connects the LLC, cores, and DRAM. Each core also has core-private caches; Intel CPUs have an L1 and L2 cache, and each layer of cache can store several cache lines, 64 bytes each. When the software accesses a memory location, the CPU uses its address to find it within the closest cache. If the data is not present in a cache level (cache miss), the corresponding cache line is fetched from the next level of cache or the DRAM.

Temporal buffers The CPU core uses various temporal buffers to optimize micro-operations [3, 29, 49, 60]. For example, when the CPU accesses a memory location missing in the L1 cache, it can use a fill buffer to fetch cache line data bits and forward them to the dependent operations before bringing the entire cache line into the L1 [52, 60]. Similarly, the store buffer holds the data for memory writes before committing them to the cache [5, 29].

Memory types The CPU supports various memory types, configurable by the OS, to enforce caching policies: write-back (WB), write-through (WT), write-protect (WP), write-combining (WC), and uncacheable (UC). As relevant examples to our work, both UC and WC are uncacheable.

2.2 Microarchitectural Data Leak

We now discuss previous transient execution and data-leak vulnerabilities affecting mainstream CPUs:

Spectre Spectre attacks [31] exploit misprediction of instructions (control-flow [32, 35] or data-flow prediction [8, 16, 61]) inside the victim code. Speculative execution of mispredicted instructions may result in out-of-bounds data access within the victim address space, which should be restricted to an attacker. If the out-of-bound access is followed by encoding the data to an architecturally-visible state like the cache [63], an attacker can leak the data by observing the cache. Current CPUs support the software with knobs, so the software can mitigate Spectre by selectively disabling speculative execution [9, 20], or partitioning predictors [18, 45]. But mitigating some attack variants requires extensive software modifications [9]. Developing more efficient hardware-software defense against Spectre is an open area of research [8, 10, 64].

Meltdown Meltdown [34] bypasses the access control within the CPU, thus enabling an attacker inside a user process to leak data from the OS kernel. Unlike Spectre, Meltdown does not rely on instructions in another address space. In its address space, an attacker accesses the kernel memory and encodes the data to the cache. Although the CPU enforces access control by blocking the attacker from directly reading the memory, a vulnerable CPU would forward the kernel data to succeeding instructions, transiently exposing the data to the attacker. Meltdown-like attacks can also bypass access control for other execution environments including Intel SGX [57], VM hypervisors [62], and memory protection keys (MPK) [6]. Intel has deployed hardware fixes for Meltdown in their 9th generation and later CPUs [17].

Microarchitectural data sampling (MDS) RIDL [60], ZombieLoad [52], and Fallout [5], broadly called MDS, demonstrate that transient execution after invalid memory accesses can unintentionally expose the contents of internal temporal buffers. They discovered that when the CPU executes a memory read that faces an exception (e.g., page fault, invalid permission) or a microcode assist (e.g., TSX abort [21]), it may forward stale data from temporal CPU buffers to succeeding instructions. As a result, an attacker can leak data from another process running on the same CPU core sharing these buffers (e.g., fill buffer, store buffer). The OS has to give up on SMT or enforce restrictive scheduling policies and flush microarchitectural buffers across context switching using the `verw` instruction to defend against MDS on earlier CPUs [23]. Researchers have extensively evaluated these mitigations discovering new variants [21, 24, 38], but most recent Intel CPUs (10th generation and later) defend against all these attacks [17].

Load value injection (LVI) LVI [58] exploits the same root cause as MDS, but the attacker induces a transient fault into the victim's code instead of crafting arbitrary gadgets in the attacker's code. Spectre-style exploitation techniques can be used to transiently hijack control flow and leak data through code gadgets available in the victim code. Like Meltdown and MDS, LVI does not need software-based mitigation on recent Intel CPUs [25].

MMIO/ÆPIC leak Data leak vulnerabilities can also plague temporal buffers inside the CPU through the architecturally-accessible system configuration address space. ÆPIC

```
1 vpgatherdd %xmm1, 0(%rsi, %xmm2, 2), %xmm3
2 vpgatherdd 0(%rsi, %zmm2, 1), %zmm3{%k1} // AVX-512
```

Listing 1: Examples of *gather* in x86: Line 1 calculates addresses of 4 dwords at `(%rsi+%xmm2[i]*2)` and merges their values into the 128-bit `%xmm3` register, depending on the corresponding mask bits (per dword) in `%xmm3`. Line 2 calculates addresses of 16 dwords at `(%rsi+%zmm2[i])` and merges them into 512-bit `%zmm3` register depending on the `%k1` mask.

Leak [3] discovered that reading from legacy APIC addresses can leak stale data from the super queue, a buffer between the L2 cache and LLC. Intel recently published about vulnerabilities that enable attackers with access to the MMIO address space to steal stale data from various microarchitectural buffers [26]. These vulnerabilities mostly affect execution environments like Intel SGX, where accessing system configuration address space is part of the threat model. For other environments, the OS can use the `verw` instruction when context switching, which was updated to flush these CPU buffers [26].

3 Gather Data Sampling

We propose and analyze *Gather Data Sampling (GDS)* attack.

3.1 Gather Instruction

The *gather* instruction (hereafter referred to simply as *gather*) enables the software to collect non-contiguous data from memory into a vector register efficiently. Figure 1 illustrates the *gather* operation, which reads non-contiguous doubleword (dd) values from different memory locations. The addresses of the non-contiguous (dword or qword) values are calculated based on a base register and a wide index register that holds several indices. The values which do not have their corresponding mask bit (in the mask register) set are discarded, and finally, the results are merged into the resulting wide register.

Listing 1 provides examples of *gather* in x86. In line 1, the *gather* `(v)(vp)gather*` collects 4 dword values using `%xmm2` as memory index and merges them into `%xmm3` register, accessing memory locations at `(%rsi+%xmm2[i]*2)`, depending on the mask bits in `%xmm1`. Alternatively, Line 2 shows that, with AVX-512, which supports wider registers and dedicated 16-bit mask registers (`%k0-%k7`), *gather* collects 16 dword values, merges them depending on the set bits of `%k1`, and discards values corresponding to unset bits.

Microarchitectural optimizations The straightforward way to implement *gather* on a CISC architecture like x86 is to translate it to legacy read micro-ops. However, this will be similar to a software implementation that is not very fast. The

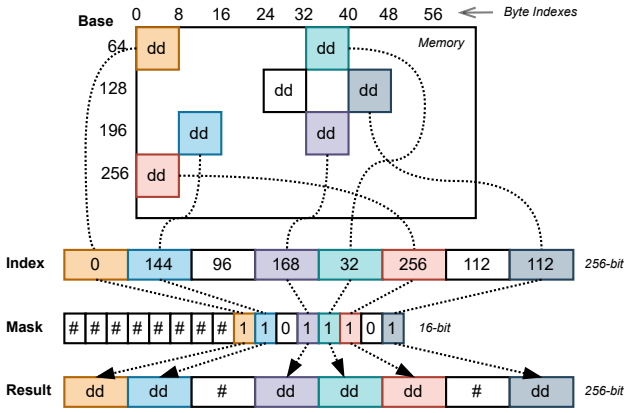

Figure 1: The gather instruction calculates dword addresses based on the indices (base+index[i]), then discards elements using the least significant 8 bits of the mask, and finally merges dword values for the set masked bits.

CPU can speed up *gather* by implementing several microarchitectural optimizations [27, 28]. It can

- eliminate unnecessary memory reads for unset mask bits, which must be discarded anyway.
- reuse the data from the same cache line when indexing the same cache line multiple times.
- execute multiple reads in parallel and speculatively, and discard results when at least one of the reads has failed.
- preserve the state of partial execution of *gather* to avoid re-executing already-executed reads when there is an interruption in the middle of multiple reads.

While the CPU can quickly eliminate some of the reads early on for unset mask bits and duplicate indices, it can use a temporal buffer to implement the other optimizations. When multiple reads target the same cache line but different offsets, a temporal buffer can retain the cache line and forward different word values to the succeeding instructions independently. A temporal buffer also facilitates out-of-order and speculative memory reads from different cache lines. According to one of Intel's patent [27], the CPU may preserve the data gathered from cache or DRAM when the *gather* instruction (e.g., reading up to 16 dword values) is interrupted or terminated before completion. When *gather* is terminated due to a fault (e.g., invalid permission), the CPU can discard the preserved values. Otherwise, during non-failing interrupts or microcode assist (e.g., setting PTE status bits), it can restart in the middle of the multiple reads to gather the remaining missing elements.

Temporal buffers shared across independent security domains could have adverse side effects [5, 26, 52, 60]. If optimization choices mentioned here use a temporal buffer, they raise security-critical questions that we answer in this paper:

- *Can gather access stale data from such a temporal buffer?*

```
// Step (i): Increase the transient window
lea addresses_normal, %rdi
clflush (%rdi)
mov (%rdi), %rax
// Step (ii): Gather uncacheable memory
lea addresses_uncacheable, %rsi
mov $0b1, %rdi
kmovq %rdi, %k1
vpxord %zmm1, %zmm1, %zmm1
vpgatherdd 0(%rsi, %zmm1, 1), %zmm5{%k1}
// Step (iii): Encode (transient) data to cache
movq %xmm5, %rax
encode_eax
// Step (iv): Scan the cache
scan_flush_reload
```

Listing 2: Testing Gather Data Sampling.

- *Can it forward stale data to the following instructions?*
- *What instructions are affected if there is a stale data leak?*

3.2 Exploiting Gather Data Sampling

The code in listing 2 introduces *Gather Data Sampling (GDS)*: An attacker *(i)* executes a cache miss to increase the speculative execution window, which increases the chance of transiently-forwarded data becoming architecturally visible through the cache, *(ii)* executes the *gather* instruction that accesses a single cache line but faces a cache miss due to target memory being uncacheable (UC), *(iii)* encodes a dword from the 512-bit vector register into 4×256 cache lines so it can reveal up to 4 bytes of transient data, *(iv)* scans the cache using the Flush+Reload technique [63] to infer forwarded data bytes.

The victim executes *gather* on the sibling virtual CPU core, similar to step (ii), but accesses a standard cacheable memory.

Testing We executed listing 2 on the Tiger Lake CPU (from table 1) and leaked 809 dword values from the other *gather* execution in the sibling CPU thread in one second. Alternatively, we can run the first three steps several times to increase the chance of encoding the full dword before scanning the cache. Next, we executed steps (i-iii) 32 times before scanning the cache, which leaked 903 dword values in one second.

Discovered vulnerability The observed data leak confirms a critical vulnerability that is exploitable from user space. The *gather* instruction appears to use a temporal buffer shared across sibling CPU threads, and it transiently forwards data to later dependent instructions, and the data belongs to a different process and *gather* execution running on the same core.

Fault/assist After establishing the data leak vulnerability by executing the `gather` on UC memory, we use the *PTEditor* [36] tool to test other conditions (faults and microcode assists). Here is a summary of our observations: GDS leaks data by accessing UC and WC memories, which are both uncacheable. GDS can also leak data by all faulty memory accesses, which can occur during the execution of *gather*: permission faults due to accessing kernel or protection keys, page faults due to accessing unmapped memory, and address generation faults due to accessing non-canonical addresses. Inspired by previous works [52], we could see slow data leak by accessing a page which its access (A) bit in the PTE is unset, but we can not confirm if the data leak is explicitly due to a microcode assist caused by the A bit.

Systematization Previous work [6] categorizes Meltdown-type attacks based on their fault/trigger mechanism. Following their terminology, GDS can leak data by applying Meltdown-US, Meltdown-MPK, Meltdown-NC, Meltdown-P, and Meltdown-UC, and potentially Meltdown-A[1]. Next, we will show that unlike previous Meltdown and MDS attacks, GDS does not require any architectural behavior such as changing accessed bits or accessing weird page tables, types, or exotic and faulty addresses, but rather it can only rely on events that can occur during normal execution.

Playing with the transient window In the previous experiments, we used a cache miss independent from the target of the *gather*, but we can use other techniques to increase the transient window. We have empirically confirmed that in addition to cache misses, we can use other critical memory accesses such as page faults or atomic load-modify-store (LMS) instructions (e.g., `xchg, lock inc`) to prepare the transient window for *gather* to leak.

Through trial and error, we observed that an attacker could prepare the transient window in a way that it does not require accessing an exotic address to leak data. Listing 3 is one example of a code snippet that prepares the speculative-execution window using a combination of cache misses and LMS executions. In this case, GDS leaks data only via accessing regular memory addresses with cacheable data without any explicit fault or microcode assist. This finding suggests that unlike MDS attacks exploiting memory accesses that experience faults or assists, *gather* can forward stale data upon normal speculative execution, similar to Spectre. This finding is critical for execution environments like the Javascript that sanitize addresses since the attacker does not have access to exotic addresses.

Mask-bit Next, we test if a data leak is possible for a memory index with an unset mask bit. We modify the index register so that *gather* accesses 16 different pages and set up one of these pages as uncacheable or with the wrong protection key. We observed data leaks only when the mask bit was set for the troubling memory index. We also tested mask bits for the case where the target memory index is not exotic (no faults and cachable) and observed data leaks when at least one of the mask bits was set, and in this case, the data leak corresponds to the dword with the set mask bit. These observations confirm that *gather* avoids accessing the leaky temporal buffer for unset mask bits even during speculative execution, which

[1] Meltdown-TAA does not apply since Intel has disabled TSX.

```
// Step 1: Increase the transient window a lot
lea addresses_normal_helper, %rdi
.set i,0
.rept 8
clflush 64*i(%rdi)
mov 64*i(%rdi), %rax
.set i,i+1
.endr
xchg %rax, 0(%rdi)
// Step 2: Gather cachable memory (no fault)
lea addresses_normal, %rsi
...
```

Listing 3: Delaying the speculative-execution window, thus increasing the chance of data leak without exotic addresses.

could be a sign that the access for unset mask bits is discarded at an early stage of the pipeline.

3.3 Affected Instructions

We showed that GDS leaks data from another *gather* instruction, but now, we automatically test all x86 instructions that accept memory operands to see which instructions leak to GDS. We develop a tool that first scrapes all instruction that accepts a memory operand from the x86 assembly test cases as part of the LLVM compiler tool chain[2] and stores them in a standard form (e.g., `vblendps CONST,(R64),YMM,YMM`). Then it generates code snippets that access memory based on these instructions, which include all supported memory instructions and their operands. Finally, it sets ups the memory layout and registers with known values before the tested memory access, so we can detect the leaking of known values on the sibling thread when the victim is accessing them, a similar approach used by *Transynther* [40].

Evaluation After excluding test files for ISA extensions that our CPU does not support, we ended up with 16304 test cases, of which 16232 were successfully compiled. We executed each compiled test case for 2 seconds on the Tiger Lake CPU while running GDS on the sibling thread. Roughly 95 percent of test cases (15523 samples from 1604 instructions) were executed without exception, of which 47 percent (7380 samples from 850 instructions) leaked data to the sibling thread.

We also manually tested instructions for which our tool cannot generate proper operands, and thus they did not compile. Table 2 provides a compressed list of affected/leaky instructions, which we discuss some of them further:

- **SIMD read** All SIMD operations that reads wide data (128/256/512 bits) from memory are affected regardless of their function: e.g., `vmov*` only read, `vpxor**` read and compute the `xor`. These generate-purpose instructions are used everywhere, e.g., compilers spread wide data reads to optimize memory access routines. Various optimized and secure implementations of cryptographic codes rely upon SIMD instructions, including wide data reads.
- **SIMD write** The only SIMD write operations that are affected are the *compress* (`(v)(vp)compress*`) instructions.
- **Cryptographic extensions** Cryptographic extensions, including AES-NI and SHA-NI (SHA1 and SHA256), when accepting a memory operand, are affected. Data leaks from these instructions expose plaintext data and the secret key, e.g., AES or HMAC-SHA.
- **Fast memory copy** Fast memory copies of various data types: byte, word, dword, qword using `rep movs*` instructions are affected. These are widely used to speed up common memory operations such as `memcpy` and `memove`.
- **Register context restore** Special instructions to more efficiently store/restore the register context (e.g., `xsave/xrstor`) are affected. GDS leaks the register context of both standard registers due to `xsave/xrstor` and wide registers due to `fxsave/fxrstor`.
- **Direct store** The direct store is affected. Intel has recently added support for a direct store instruction that can copy a 64 bytes cache line from a source to a destination address. `MOVDIR64B (%rsi), %rdi` copies data from the address in `%rsi` to the address in `%rdi`. Our code generator failed to produce the correct address operand for `%rdi`, but we could manually verify the data leak.

Instruction buckets:	(v)(vp)(p)blend*{19}	(v)(vp)(p)cmp*{217}
(v)(vu)(u)comi*{8}	(v)insert*{12}	(v)(vp)(p)align*{4}
(v)(vp)maskmov*{4}	(v)(vp)(p)mov*{47}	(v)perm*{22}
(v)(vp)compress*{4}	(v)(vp)gather*{8}	(v)(vp)max*{12}
(v)scale*{4}	(v)(vp)(p)shuf*{17	(v)rsqrt*{7}
(v)sqrt*{6}	(v)fixup*{4}	(v)fpclass*{10}
(v)getmant*{4}	(v)(vp)xor*{5}	(v)(vp)or*{5}
(vp)rol*{4}	(v)pack*{4}	(vp)(p)srl*{10}
(v)(vp)andn*{5}	(v)(vp)and*{5}	(v)getexp*{4}
(vp)lzcnt*{2}	(v)lddqu{1}	(vp)dpwssd*{2}
(v)dbpsadbw{1}	(vp)sadbw{1}	(v)rndscale*{4}
sha*{6}	(vp)madd*{4}	(vp)ror*{4}
(v)cvt*{74}	(v)dpp*{4}	(v)gf2p8*{6}
(v)(vp)(p)hadd*{10}	(vp)(p)abs*{7}	(vp)(p)clmul*{7}
(v)phmin*{2}	(v)(vp)min*{12}	(v)popcnt*{4}
(v)div*{4}	(v)(vp)broadcast*{17}	(v)fm*{36}
(v)(vp)(p)test*{12}	(vp)multishift{1}	(v)(vp)(p)mul*{13}
(v)rcp*{7}	(v)round*{8}	(v)reduce*{4}
(v)range*{4}	(v)(vp)expand*{6}	(vp)ternlog*{2}
(v)addsub*{2}	(v)(vp)add*{12}	(v)(vp)sub*{12}
(vp)conflict*{2}	(vp)(p)sll*{9}	(vp)(p)sra*{8}
(vp)dpbus*{2}	rep(ne) mov*{8}	xsave/xrstor*{2}
fxsave/fxrstor*{3}	(v)(vp)(p)hsub*{10}	(vp)sign*{3}
(v)(vp)unpck*{12}	(v)fnm*{24}	(vp)(p)ins*{6}
(vp)shl*{6}	(vp)2intersect*{2}	(v)mpsad*{2}
(vp)shr*{6}	(vp)avg*{2}	(v)aes*{12}

Table 2: Affected instructions: `{n}` is the number of instructions from an affected category. `(v)(vp)(p)` are vector instruction prefixes. `*` indicates various data types or functions within the same bucket.

[2] https://github.com/llvm/llvm-project/tree/main/llvm/test/MC/X86

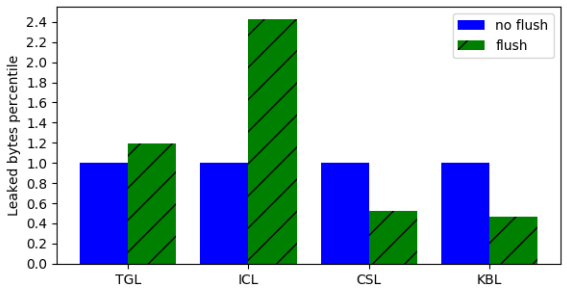

Figure 2: Normal data leak compared with leak after flushing microarchitectural buffers across different CPU generations.

False positives We observed a slow data leak from standard memory reads (e.g., `mov`), but we confirmed that this is due to the `rxstor` instruction, which is executed periodically upon a periodic timer interrupt initiated by the OS, resulting in the register that holds the read value from memory being saved and restored. Although this shows that GDS does not leak from standard memory reads and writes, it highlights that an attacker who can interrupt the execution frequently can steal data from CPU registers. We exploit this behavior to leak the data from SGX enclave registers in section 8.

3.4 Temporal Buffer Analysis

We analyze the size of the leaky buffer entries and the effect of flushing microarchitectural buffers.

Entry size We test data leaks from different portions of the wide memory read by permuting and shuffling the leaked data values. We observed that we could leak any part of the 512-bit loaded data (`%zmm`) on the Tiger Lake CPU that supports AVX-512, which means each buffer entry has to be 64 bytes, the size of a cache line. On older CPUs that do not support AVX-512, we can only leak 32 bytes, the maximum size for vector registers (`%ymm`).

Flushing buffers Next, we test if flushing microarchitectural buffers would mitigate the data leak within the same CPU thread. Before executing GDS, we flush microarchitectural buffers using the `verw` (as suggested by `md_clear` [23] and `fb_clear` [26] hardware mitigations), and with the help of a kernel module, we flush the L1 data cache using the `MSR_IA32_FLUSH_CMD` MSR. We tested this across different CPU generations, and as we can see in figure 2, we can efficiently leak data on all tested CPUs even after flushing everything that we can. In fact, in some cases, we see more data leaks, likely due to changing the speculative-execution window, but it confirms that previous hardware mitigations do not flush our newly discovered buffers.

CPU Generation	GDS		MDS				VERW
	SMT	Switch	SMT	Switch	>TAA	>MMIO	Cycles
Tiger Lake	ϑ	ϑ	θ	θ	θ	θ	80
Ice Lake	ϑ	ϑ	θ	θ	θ	Δ	592
Cascade Lake	ϑ	ϑ	θ	θ	⇙	Δ	324
Kaby Lake	ϑ	ϑ	ϑ	Δ	⇙	Δ	696

ϑ Vulnerable θ Not affected ⇙ TSX disabled Δ Buffer flush

Table 3: Comparison of GDS with MDS and MMIO leak vulnerabilities: The `verw` cycle counts show that it flushes buffers when at least one previous vulnerability applies. GDS leaks regardless of `verw` behavior. The leaky SIMD register buffers has more entries on recent CPUs.

3.5 Summary

Table 3 summarizes our findings: it shows that current CPUs, regardless of previous MDS and MMIO data leak vulnerabilities, are all affected by GDS in both context-switching and SMT scenarios. We also see the cycle count for executing the `verw` instruction on these CPUs. These cycle counts suggest that `verw` does not do anything on the recent Tiger Lake CPU not affected by any previous data leaks. But on other CPUs, `verw` flushes some microarchitectural buffers. Specifically, on Kaby Lake, which is affected by all previous attacks, the `verw` flushes every known buffer (store buffer, fill buffer, load buffer) [23]. However, GDS still leaks regardless of the behavior of `verw`. The ineffectiveness of previous mitigations and only SIMD memory accesses are being affected confirms that the data leak is from a different temporal buffer than previous works, *SIMD register buffers*, which Intel confirms. Other instructions like `rep mov` that benefit from special microcode optimizations[3] can also share these buffers.

4 Cross-Process Covert Channel

We demonstrate how GDS leaks more data simultaneously compared to previous attacks. We implement multiword data sampling and characterize cross-process data leaks by demonstrating covert channels and measuring their bandwidth.

4.1 Multiword data sampling

Following previous work, we can encode the leaked data into many cache lines to leak several data bytes at the same time [53]. For example, the Meltdown attack used 256 cache lines to leak 1 byte, and ZombieLoad used 3×256 cache lines to leak 3 bytes. Here, we try to leak up to 32 bytes simultaneously using 32×256 cache lines and extracting different portion of a wide register (512/256 bits) with the help of `vextract*`, `pextr*`, `vperm*` instructions.

Maximum simultaneous leaks We execute GDS with the multiword encoding to leak up to 32 bytes. On

[3]The newly introduced FSRM extension optimizes `rep mov` with size less than 128 bytes (2 cache lines) differently [22].

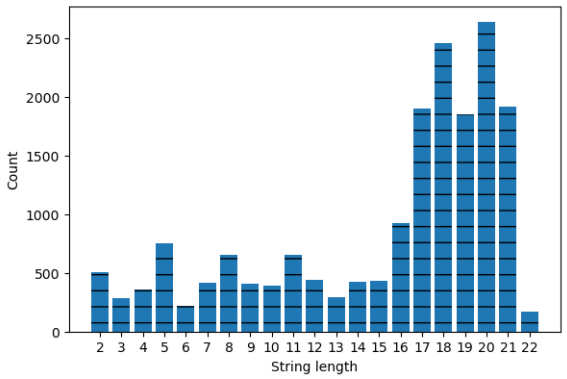

Figure 3: The frequency of different leaked words is between 2-22 bytes (maximum observed) on the Tiger Lake CPU.

the sibling thread, we execute the `vmov` instruction that loads a cache line holding a continuous data pattern: `size_of(A..Za..z0..9#!)=64`. Figure 3 shows the frequency of leaking different continuous words in an optimized execution of the attack on the Tiger Lake CPU. As we can see, the maximum number of simultaneous leaks is 22 bytes, and most leaks are between 16-21 bytes long, which is much higher than any previous attacks reported.

Data leak pattern Next, we change the victim thread to execute three `vmov` to three different cache lines, each holding a different data pattern. In the attacker thread, we alternatively use `gather{dd}` for dword and `gather{qd}` for qword leaks. Listing 4 shows data leak patterns from each of these instructions. We have two observations: **(1)** The *gather* instructions leak data from multiple entries of the SIMD register buffers filled by different `vmov` executions; therefore, simultaneous data leaks are not necessarily continuous. **(2)** The choice of dword or qword for *gather* leaks two different data patterns.

As we will see in later sections, the data leak pattern is essential for developing various attacks. In a covert-channel attack, the attacker can choose how to simultaneously send and receive data to leak up to 22 bytes of continuous data. But for attacks on other applications, we can only guarantee to leak one continuous dword or qword simultaneously since other operations holding uninteresting data may fill up some of the SIMD register buffer entries. However, an attacker can access another dword/qword by permuting the leaked wide register.

4.2 Covert Channel

We execute cross-process covert-channel attacks based on multiword data sampling and evaluate their bandwidth. In these attacks, we execute three variants of GDS with fault, uncacheable memory, and normal cacheable memory without any fault and also use three different affected instructions to send data: `vmov`, `rep mov`, and `aes`.

```
dword:
ABCDEFGHIJKLMNOPQRST
ABCDXXXXIJKLXXXXQRST
ABCDYYYYIJKLYYYYQRST
XXXXEFGHXXXXMNOPXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXYYYYXXXXYYYYXXXX
YYYYEFGHYYYYMNOPYYYY
YYYYXXXXYYYYXXXXYYYY
YYYYYYYYYYYYYYYYYYYY
```

```
qword:
ABCDEFGHIJKLMNOPQRST
ABCDEFGHXXXXXXXXQRST
ABCDEFGHYYYYYYYYQRST
XXXXXXXXIJKLMNOPXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXXXXXYYYYYYYYXXXX
YYYYYYYYIJKLMNOPYYYY
YYYYYYYYXXXXXXXXYYYY
YYYYYYYYYYYYYYYYYYYY
```

Listing 4: Leaked data pattern from multiple memory reads.

Evaluation We execute each covert channel for 60 seconds across different CPUs from table 1. Table 4 shows bytes leaked per second for each experiment. As we can see, in the best-case scenario, we leaked 5870.3 bytes per second on the Tiger Lake CPU. Since we have developed these covert-channel attacks on the Tiger Lake CPU, we have optimized the transient execution window for this machine, one of the reasons the covert channel performs significantly better on this machine. In theory, the covert channel should have a higher bandwidth for CPUs with more SIMD register buffer entries (more chances of picking up one copy of the data) and ROB entries (higher likelihood of encoding all the quadwords into the cache) We can observe this across Tiger lake, Cascade lake, and Kaby lake. The only anomaly here is the Ice Lake CPU. Although we could see a similar quadword leak on both Ice Lake and Tiger Lake, we did not investigate why the data rate is lower. But the results serve our purpose of confirming that all tested CPUs are affected by cross-process covert channel due to GDS.

5 Stealing Cryptographic Keys

In this section, we develop end-to-end attacks against AES encryption executed by the OpenSSL command line tool. Our attack does not rely on cryptanalysis, knowing the plaintext or ciphertext, or analyzing the source code. We evaluate our attack against AES-128 and AES-256 keys running on a separate VM, each VM on sibling thread of the same CPU core.

5.1 Blind Attack against AES

In this attack, our only assumption is that the attacker knows that OpenSSL uses AES-NI for fast encryption; thus, the encryption uses SIMD memory reads. The attacker sends a request to another VM to encrypt random data (from `devurandom`) with salt and discards the output [4].

Knowing that AES uses 10 round keys derived from a master key to encrypt data, in a separate VM instance running

[4] openssl aes-256-cbc -salt -e -in /dev/urandom -out /dev/null -K $k -iv $i

CPU Generation	vmov			rep mov			fxstor			aes		
	●	∪	×	●	∪	×	●	∪	×	●	∪	×
Tiger Lake	4128.78	5584.57	5870.3	3318.15	1438.53	1414.55	92.35	1465.13	178.68	688.27	1763.57	1101.7
Ice Lake	0.73	2.48	6.25	11.67	58.13	30.97	0.0	0.57	3.05	0.1	6.68	7.42
Cascade Lake	133.27	72.47	2424.83	19.23	14.23	2569.78	76.2	3.98	1209.13	8.0	75.77	1395.7
Kaby Lake	0.03	26.45	11.12	0.2	3.87	70.2	0.03	0.1	0.07	0.0	0.13	2.03

● Cacheable no fault ∪ uncacheable × Page fault

Table 4: Covert channel data rates (Bytes/Second).

in the sibling CPU thread, we execute the following attack: *(i)* We steal 8 bytes (qword) of random data and create a histogram for qwords with a high frequency. *(ii)* We repeat the above for the second leaked qword since AES-NI uses 128-bit (16 bytes) reads to load round keys. *(iii)* We combine the high-frequency qwords to construct AES-128 or AES-256 round key candidates. *(iv)* We search through the key candidates, and one of the round keys should be the master key.

The blind attack against AES works due to the following: When AES encrypts data regardless of its mode, e.g., CBC, it repeatedly calls a low-level function for encryption that loads all round keys to encrypt a new block. These round keys, 128-bit or 256-bit, always use the same type of SIMD reads since AES-NI only supports 128-bit registers. GDS naturally ignores non-SIMD reads, and the only random data exposed to SIMD reads, which are repeated, are AES round keys since the victim encrypts random but ephemeral data.

Evaluation In 10 seconds, we steal AES-128 or AES-256 round keys. We execute the attack's first and second qword for 5 seconds each. We discard qwords that have occurred less than 16 times to filter out random data leaks that are not from round keys, i.e., are not repeated. To create round key candidates for AES-128, we combine first+second qwords. For AES-256 with 32-byte keys, we combine (first+second)×2

Averaging over executing the attack 100 times, we observe a 7.40-bit key space (roughly 128 candidates) for AES-128 and a 15.99-bit space for AES-256. We then search through these key candidates by having access to a single plaintext/ciphertext block and searching each candidate as a master key to encrypt and check, which means the complexity of our attack will be the size of the candidate key space.

Alternatively, we perform a blind search by just checking the key candidates against each other, assuming each candidate is a master key, compute round keys, and then check if one of the other leaked qwords appears in the computed round keys. Our evaluation shows that this alternative blind attack has a complexity of 10.29 bits for AES-128 and 17.99 bits for AES-256, which a single-core desktop CPU can trivially compute them. It could be possible to develop a more efficient blind search by looking closely at the round keys relationships, but that is unnecessary.

For 100 different keys, the first run of the attack was 100% successful for AES-128. The first run of the attack was 86% successful for AES-256. On failed attempts, one of the master key's qwords did not appear with a high frequency in 10 seconds. For these rare cases, we just rerun the attack multiple times to recover the entire key.

6 Stealing Arbitrary Data

In this section, we introduce other variants of GDS that can steal arbitrary data at rest [61]. We introduce practical code gadgets that enable data-at-rest leaks and demonstrate practical attacks against the Linux kernel based on them.

6.1 Leaking Data without Accessing

We discovered two conditions where the CPU prefetches data at rest[5] into the SIMD register buffers, thus GDS can leak it, despite the fact that the data is not read by the software:

- **OOB prefetch:** The software reads n bytes, but the CPU leaks more than n bytes up to x cache lines.
- **NOOP prefetch:** The software reads 0 bytes, but the CPU leaks up to x cache lines.

We discuss how masked move (`maskmov`) and repeated move (`rep mov`) operations match the above conditions.

Masked move We observed that even if only a subset of masks are set, e.g., only reads a single dword, `maskmov` prefetches up to $x = 1$ cache line, which we can leak. Interestingly, even if the mask bits are all zero, which is a no-op for the software, GDS still leaks 64 bytes from the target address.

Constant-time cryptography can benefit from the SIMD masked move operations [12, 50]. Although SIMD programs are generally vulnerable to GDS, the masked move may expose more secrets due to NOOP and OOB prefetching.

Besides, a memory allocation unaligned to cache lines and accessible to `maskmov` may enable an attacker to leak out-of-bounds (OOB) data: Imagine that the software does not intend to read a secret at a particular time window and enforces bounds-checks correctly. The software does not architecturally access those secrets since it does not read them, but suppose the software uses a masked move instruction to read public data that happens to be before the secret data in the program's memory layout. The attacker can coerce the

[5] Data that is mapped to the victim address space but not used.

software to load those secrets into the SIMD register buffers and leak them by requesting the software to access the public data. We leave further analysis of leaking data-at-rest from the masked move for future work.

Repeated move The `rep mov{t}` instruction copies `%rcx` × `size_of(t)` from address `%rsi` to address `%rdi`, where `t` is the data granularity: `byte`, `word`, `dword`, `qword`. We observed on the latest Tiger Lake CPU that we can leak OOB data from `rep mov` up to $x = 2$ cache lines (128 bytes) even if the copy size is less than that, disregarding the data granularity and `%rcx` values. More surprisingly, we can leak if `%rcx`=0, which is a no-op for the software. This data leak is potentially due to the speculative behavior of the `rep mov` instruction, as confirmed by concurrent work [43], which results in polluting the SIMD register buffers with data that is not architectural accessed.

Software compilers spread `rep mov` instructions everywhere for common operations: e.g. `memcpy` and `memmove` are part of the `c` standard library, and compilers use `rep mov` to execute them efficiently. This popularity has several consequences for the application's security. *(i)* The `rep mov` handles a lot of confidential data, driving the standard data copy functions used everywhere by the application. *(ii)* The `rep mov` can leak up to 128 bytes of OOB data, resembling a transient buffer overflow. *(iii)* The `rep mov` can act as a confused deputy gadget enabling an attacker to steal arbitrary data. In the next section, we discuss common code gadgets based on `memcpy` that enable attackers to steal arbitrary data from a program's address space.

6.2 Data Leak Gadgets

Here we discuss code sequences for copying memory content that is not vulnerable to traditional buffer overflow attacks or Spectre [31] but can be used to get interesting data into the SIMD register buffers, hence leaking the target memory content with GDS. An attacker can trigger these gadgets from a different security domain (e.g., over the network or from userspace to kernel) to leak secrets even if the target code does not explicitly access those secrets. An everyday use case of `memcpy` (or `memmove`) is to copy the user's input from a user-facing memory allocation to a local allocation in the program's address space for later processing. For the three code gadgets in listing 5, we assume that the software copies user data from the `source` to the `local` buffer, only accessible to the program. The software should also sanity-checks the user-inputs: `copySize` and the source `index` to avoid traditional memory-corruption software bugs [44]:

Gadget 1: Safe check Line 1-5 shows a correct input sanity-checking which avoids both OOB read and write bugs; hence it is safe for the software. Although this is safe for the software, an attacker can use GDS to leak past the `source` buffer. This is especially easy since the attacker controls the `index`, a reasonable assumption. For example, if

```
1  // Gadget 1: Safe check
2  if(copySize < sizeof(local) &&
3    copySize+index < sizeof(source)){
4    memcpy(local, source+index, copySize)
5  }
6  // Gadget 2: Safe no-op check
7  if(copySize >= sizeof(local) ||
8    copySize+index >= sizeof(source)){
9    copySize = 0
10 }
11 memcpy(local, source+index, copySize)
12 // Gadget 3: Buggy unexploitable
13 if(copySize < sizeof(local))
14   memcpy(local, source+index, copySize)
```

Listing 5: Sanitizing and copying user's data.

`size_of(source)=64`, `index=63` and `copySize=1` fullfill the sanity-checks, but the next two cache lines, 128 bytes will be prefetched by the `rep mov` and leaks.

Gadget 2: Safe no-op check Line 6-10 shows an alternative sanity-checking supported by the C programming language [51] that is correct and safe for the software. Here, the `if` statement simply sets the `copySize` to zero when it detects an OOB access. Since zero `memcpy` is officially supported by the C language and `rep mov`, the CPU architecturally treats this as a no-op. However, as mentioned before, GDS can leak from `rep mov` even when the size register is zero, so an attacker can use this to steal arbitrary data, even if this is architecturally safe for the software.

Gadget 3: Buggy unexploitable Line 11-15 exemplifies a typical case where the input sanity-checking is buggy, but it does not necessarily result in an exploitable software vulnerability. Here, the software checks to ensure `copySize` is less than the size of the `local` buffer, so a malicious user cannot perform an OOB write, overwriting the program's memory with malicious input and potentially executing arbitrary code. Although there is no sanity check on the index, which results in an OOB-read bug, a software exploit cannot read arbitrary OOB data since `local` buffer is never exposed to the user. However, an attacker can exploit GDS to steal arbitrary data from the target application's address space by relying on this bug to prefetch arbitrary data into the SIMD register buffers.

6.3 Reading Kernel Memory from User-space

These simple gadgets are a nightmare for software's security. We especifically focus on the Linux Kernel, but other scenarios like a web application that accept user's input can also be vulnerable to the demonstrated attack technique. We discussed the simplest form of `memcpy` gadgets that leak arbitrary data at rest, but it is inevitable that complex input-checking code will contain similar gadgets. Developers already have to spend a lot of resources to address software bugs. Now, they also have to look for other code patterns that are traditionally

unexploitable or denial-of-service bugs (Gadget 3), or worst, they are not even bugs (Gadgets 1, 2) but an attacker can exploit them to leak kernel data.

We demonstrate proof-of-concept attacks against the Linux kernel. For this, we develop a loadable-kernel module that implements Gadgets 1-3 and accepts user inputs through an `ioctl`. We sanitize the input correctly (Gadget 1,2) and do not expose the destination buffer in the user space (Gadget 1,2,3) to ensure that the user cannot attack the kernel using a software attack.

Evaluation We use one process to perform `ioctl` calls from the user space and provide the `index` and `copySize` used by the `memcpy` to access data and a process on the sibling CPU thread to execute GDS. The attacker, who controls the `index` so it can navigate the victim into prefetching memory offsets, leaks one dword at a time (based on our observations in section 4). Iteratively, we increase `index` by 2 bytes and combine the next dword if the first 2 bytes of the current dword matches the last 2 bytes of the previous one, similar to the ZombieLoad attack [52], but with a 16-bit window applicable to our dword leak which makes the attack much more reliable. Alternatively, GDS allows us to leak up to one qword at a time and use a bigger window, e.g., 32-bit, to combine the leaked qwords.

We executed attacks with the described gadgets on the Tiger Lake CPU. We can exploit gadget 1 to steal OOB data past the `source` buffer. In this case, we assume the source buffer is cache-aligned, which means gadget 1 prefetches two additional cache lines, 128 bytes, to the SIMD register buffers. On average, after executing the attack ten times, we leaked the 128 OOB data exploiting Gadget 1 in 1.04 seconds. We found that we can exploit gadgets 2 and 3 to steal arbitrary kernel data. Gadget 2 does not architecturally access nor copy the data but prefetches it. Gadget 3 accesses and copies arbitrary memory to a local memory allocation, inaccessible to the user. Averaging the attack among ten tries, we leaked 238 bytes of the Linux banner string, `linux_banner` exploiting Gadget 2 in 1.37 seconds and Gadget 3 in 1.58 seconds.

Impact on Linux Kernel Similar code patterns, as discussed in listing 5 can be found in widely-used software. For instance, we searched through the Linux kernel binary and found 2728 instances of `rep* mov*` instructions. We also searched through the latest Linux kernel source code and discovered 25992 `memcpy` and 860 `memove`, which both execute `rep* mov*`. Many of these are reachable through multiple code paths. We leave further analysis of the reachability of such code gadgets for future work.

7 Gather Value Injection

We introduce the *gather value injection (GVI)* attack by combining GDS with the LVI [58] technique. To exploit GVI, attackers only need to find a *gather* instruction in the victim code followed by data-dependent operations, similar to a Spectre gadget. We introduce several GVI gadgets and exploit them to steal out-of-bounds data from a victim process.

```
1 // Gadget A: Gather followed by a load
2 new_index[i] = gather(index_base, index[i]);
3 value = data_base[new_index[0]];
4 leak_to_side_channel(value);
5 // Gadget B: Double gather
6 new_index[i] = gather(index_base, index[i]);
7 values[i] = gather(data_base, new_index[i]);
8 leak_to_side_channel(values[i]);
```

Listing 6: Example GVI gadgets.

7.1 GVI Gadgets

Microarchitectural side-channel attacks can typically leak secrets from data-dependent operations, but if such operations rely on *gather* to access data, they can form a code gadget exploitable by GVI, leaking arbitrary data in addition to the data the software was supposed to access. For example, if a program executes *gather* instruction and uses its output to index into another allocated memory, an attacker can coerce the program to access OOB by injecting stale memory indexes into *gather* during transient execution to leak arbitrary data outside that allocation and from the victim address space.

Listing 6 provides two such examples: In lines 2-3, the dword/qword output of the *gather* is used as a memory index to read data from another allocation. In line 4, the software performs a data-dependent operation that would leak this data over a covert channel like the cache. Naturally, software that is not buggy would encode `new_index` in a way that it does not access out-of-bounds memory, but here, we can transiently inject arbitrary index so `new_index` would point out to an arbitrary address inside the victim's address space, resulting in leaking arbitrary data. In lines 6-9, we see another example, but this time, it uses two *gather* instructions, one to find the correct index and the second one to access the data, which should be a widespread use case for the *gather* instruction.

7.2 Exploiting GVI Gadgets

We demonstrate that an attacker can exploit the code gadgets mentioned earlier to leak data out-of-bounds from a victim program's address space. The victim runs Gadget A or B to process some data based on the `index` and also sanity-checks the input to ensure that the `index` is not OOB with respect to the memory allocation `data_base`. The attacker does not have direct access to any of the variables in these gadgets, including `index`, but it can learn the output of this operation using the Flush+Reload [63] covert channel. These are common assumptions for a regular microarchitectural side-channel attack [13, 31]. The attacker running on the sibling

thread executes multiple `vmov` to fill up the SIMD register buffers with invalid out-of-bounds indexes.

Evaluation For evaluation, we set the target of the *gather* as UC to ensure it efficiently picks up stale data. We also introduce a cache miss before the *gather* execution to increase its transient window, similar to the preparation step we introduced for the GDS in section 3. Real-world code could unintentionally open the speculative window necessary for gather (similar to listing 3). We executed this attack 100 times, and for 10 seconds each, on the Tiger lake machine. Our results show that we can leak 8734.3 bytes per second of OOB data from the victim's address space, confirming that GVI is a practical attack exploiting the root cause of GDS.

Discussion GVI does not require accessing an exotic memory address (fault, assist, uncacheable), making it more practical to exploit against user-level applications. A code sequence may introduce the proper speculative execution window for *gather* similar to listing 3, so the attacker does not need to create faults or rely on uncacheable memory to exploit a GVI gadget. The *gather* instructions are becoming more common for indexing into non-continuous memory efficiently. For example, it is used in various data encoding/decoding libraries inside the firefox web browser and optimized implementation of cryptographic routines. Where the software double-index using *gather* [11], the GVI attack technique can likely be applied to leak data/secrets. Future work can investigate the prevalence of GVI gadgets in applications. Depending on the upcoming hardware mitigation for Downfall, automated ways to find such gadgets in large applications could be relevant for both attackers and defenders.

8 Breaking Intel SGX

In this section, we show that GDS leaks data from the Intel SGX execution environment at its highest security configuration since the SIMD register buffers is not cleared when exiting SGX. Consequently, we leak wide register values (due to `fxsave/fxrstor` instruction) and use them to steal SGX sealing keys, which is critical to the enclave root of trust and its remote attestation.

8.1 Leaking Enclave Registers

Intel SGX (no longer supported on recent client CPUs but supported on server CPUs) provides a trusted execution environment for software *enclaves* that can run on top of a potentially-compromised OS. SGX enclaves reside in the virtual address space of a user-space process, but the CPU enforces the confidentiality and integrity of the enclave by isolating its runtime memory so that even the OS cannot access it. When the CPU receives an interrupt or exception in enclave mode, the CPU securely saves and scrubs the register context and exits the enclave. The untrusted software can resume the enclave's execution using `eresume` instruction. Intel SGX also supports remote attestation, which is critical for remote users to verify the integrity of the enclave and the CPU hardware.

On a Kaby Lake CPU vulnerable to Foreshadow [57], MDS [52, 60] and LVI [58], customers can only rely on SGX's highest security by applying microcode patches and disabling SMT. Microarchitectural buffers and the L1D cache are flushed before exiting the enclave on the latest microcodes to prevent leaks across context switching. And the remote attestation protocol informs the user that the enclave is executing at its highest security guarantee when the system has SMT disabled [23, 26].

We bypass these defenses and leak secrets from secure enclaves at their highest security configurations. We develop a simple enclave application that populates wide registers (`%xmm0-15`) with a known data pattern for this proof of concept. We use the SGX-STEP[59] tool to control the enclave execution in the popular OS adversary model that SGX supports [25, 37, 59] and pause the enclave's execution after the program fills these registers. Then we apply zero-stepping technique [41, 54, 59] by marking the enclave code page as non-executable. While zero-stepping on the same instruction, we execute GDS on the same execution thread after every exit of the enclave to see if it leaks the same CPU register values from repeated execution of the `fxsave/fxrstor` instructions. We executed this proof-of-concept, and the observation confirms SGX is vulnerable to GDS even without using hyperthreading and on the latest microcode update for the Kaby Lake CPU. Now that we have established vulnerability of SGX to GDS during context switching, we can use the same attack technique to break the SGX root of trust.

8.2 Stealing Sealing Key

Stealing the Intel SGX sealing keys compromises the root of trust for SGX enclaves. Intel SGX remote attestation protocol relies on data sealing, which encrypts data using keys derived from the CPU fuse before writing them to the disk. Intel EPID protocol [4] used by SGX seals the attestation key inside the *provisioning* enclave and unseals it in the *quoting* enclave, used by almost all enclaves to perform remote attestation. Using an approach similar to previous work (Foreshadow [57]), we steal the sealing key, compromising the security of remote attestation for SGX enclaves; therefore, SGX enclaves cannot be trusted anymore.

The `sgx_seal_data` inside Intel SGX SDK accesses executes the `sgx_get_key` function, which uses a special CPU instruction to derive a hardware key from the CPU. Then, it executes Intel's IPP implementation of the AES cryptography to encrypt a data blob. During encryption, IPP calls `l9_aes128_KeyExpansion_NI` function to expand the master AES-128 key into AES round keys.

We attack the `l9_aes128_KeyExpansion_NI` function by pausing the execution of `sgx_seal_data` at the begining of this function and right after it loads the AES-128 master key into the `%xmm0` register (Listing 7). Once we are there, we execute the same attack described earlier, combining GDS with zero stepping to iteratively leaks key values that are exposed while saving and restoring wide registers.

```
<l9_aes128_KeyExpansion_NI>:
endbr64
vmovdqu (%rsi),%xmm0
vpslldq $0x4,%xmm0,%xmm2 // <-- Zero Stepping
```

Listing 7: Targeted code to steal SGX sealing key.

Evaluation We used `vpgatherdd` and `vpermdd` to leak different `dword` values upon context switching. We executed this attack for 10 seconds for each of the four dwords corresponding to a 128-bit AES key. The correct dwords leaks with the highest frequency, which we simply combined to construct the master AES key. The attack may sometimes fail due to an unrelated register value leaking with higher frequency since we cannot choose which entry of the SIMD register buffers we leak. However, among all the register values during zero stepping, the dwords from the AES key have a noticeable uniformly random distribution. Therefore, we simply rerun the attack a few times until we see the correct dwords that have a uniformly random distribution, as expected for AES keys.

9 Mitigation

9.1 Software-based workarounds

We discuss software workarounds to mitigate GDS and GVI.

Disabling SMT Disabling SMT, i.e., hyperthreading can partially mitigate GDS and GVI attacks in exchange for losing performance. A computer with hyperthreading is 30% faster than an identical system [7], which makes disabling SMT expensive for customers. Besides, it does not prevent data leaks across context switching.

Disallowing affected instructions The OS and compiler can disallow certain instructions that leak secrets to *gather* to mitigate data leaks. The compiler can rewrite SIMD memory instructions with equivalent normal reads to prevent applications from directly leaking data. The OS can disable commonly used instructions that use the SIMD register buffers, `rep mov` and `xsave/xrstor`, to prevent leaking arbitrary memory and registers. However, this could not fully mitigate the attack if the software misses some instructions that still leak and could be disruptive for some applications.

Disabling *gather* Intel could issue a microcode patch that disables the *gather* instruction, slowing down or breaking applications that rely on this performance feature. However, this is impractical and requires changing the ISA since *gather* is a built-in part of the AVX2.

Preventing transient forwarding Preventing transient forwarding of data to following instructions can mitigate Downfall attacks. Adding a load fence `lfence` after the *gather* in applications may prevent GVI attacks, ensuring that *gather* does not transiently forward data to the following instructions. Similarly, in environments where the compiler is trusted, and the attacker cannot choose native instructions (e.g., WASM), the compiler can add `lfence` to *gather* to mitigate GDS. Intel plan to release a microcode update that prevents transient forwarding of data from *gather* to mitigate GDS and GVI.

9.2 Testing

Researchers have proposed several tools to automatically test for transient execution attacks [14, 30, 40, 42], which could be promising for finding these vulnerabilities earlier. Mainly, Transynther [40] uses a fuzzing-based approach to discover Meltdown/MDS-like attacks. We analyzed Transynther [39] and discovered that it does not generate the `gather` instruction, one of the reasons it has not found GDS. We modified it to only test for `gather` and executed 1000 tests. Looking through the tests, we confirm that Transynther can rediscover GDS leaking stale SIMD data from both CPU threads and in various trigger conditions (e.g., Meltdown-MPK, Meltdown-UC, Meltdown-US). Although this analysis does not quantify the performance of Transynther in finding GDS since we artificially modified it only to target *gather* and not other instructions, it serves as a proof-of-concept that fuzzing can be a promising direction to find these vulnerabilities automatically.

10 Conclusion

We conclude that the majority of superscalar CPUs are vulnerable to Gather Data Sampling, exposing users' data across various security domains. The previous defense is ineffective. Recent CPUs are not secure, but they also leak more data compared to older generations and are easier to exploit, as demonstrated by leaking multiword AES keys. We also show that *gather* enables a whole new class of transient execution attacks exposing arbitrary and OOB data from the applications process and the OS kernel.

Mitigating GDS without eradicating the root cause in hardware is expensive. As the size of microarchitectural data structures also grows, mitigations based on flushing buffers would also be less efficient, i.e., more flushing. On the other hand, automated testing can practically find new vulnerabilities in CPUs, but such tools need to have better coverage of the hardware and the supported instructions, which are challenging due to the complexity and proprietary aspect of the hardware.

Our analysis and findings focus on Intel CPUs with the majority market share for superscalar CPUs. Intel states that

newer CPUs such as Alder Lake, Raptor Lake, and Sapphire Rapids are unaffected, although not a security consideration and seems just a side effect of a significantly modified architecture. However, our findings are alarming for other CPU vendors as well. The attack technique we demonstrated can broadly apply, even though each vendor implements Gather and SIMD register buffers differently. Our preliminary tests on AMD Zen2 showed no sign of data leaks, but we plan to continue our investigation of automated and scalable testing of other CPUs manufactured by Intel and different vendors. Intel has shared the paper (with our permission) with other CPU and software vendors so that those organizations can assess the impact on their products..

Acknowledgments

We thank Alyssa Milburn, Thais Moreira Hamasaki, Ke Sun, Priya Iyer from Intel, Kurt Thomas from Google, and the anonymous reviewers, for their constructive feedback. We thank Thomas Eisenbarth from the University of Luebeck and Deian Stefan from UCSD for giving us access to their lab servers that we used for some of our experiments.

References

[1] AMD. Transient Execution of Non-canonical Accesses. https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1010, 2020.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *USENIX Security*, 2022.

[3] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. AEPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security Symposium*, 2022.

[4] Ernie Brickell and Jiangtao Li. Enhanced privacy id: A direct anonymous attestation scheme with enhanced revocation capabilities. In *ACM workshop on Privacy in electronic society*, 2007.

[5] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019.

[7] Shawn Casey. How to determine the effectiveness of hyper-threading technology with an application. *Intel Technology Journal*, 2011.

[8] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *PLDI*, 2020.

[9] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[10] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative Privacy Tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO*, 2021.

[11] CRYSTALS-KYBER. Selected Algorithms: Public-key Encryption and Key-establishment Algorithms. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip.

[12] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. Software toolkit for hfe-based multivariate schemes. In *Cryptographic Hardware and Embedded Systems*, 2019.

[13] Cesar Pereida García and Billy Bob Brumley. Constant-Time callees with Variable-Time callers. In *USENIX Security Symposium*, 2017.

[14] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: a pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems*. Springer, 2017.

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

[17] Intel. Affected Processors: Transient Execution Attacks & Related Security Issues by CPU. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html.

[18] Intel. Indirect Branch Predictor Barrier . https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html, January 2018.

[19] Intel. L1 Terminal Fault. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1-terminal-fault.html, August 2018.

[20] Intel. Speculative Store Bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html, May 2018.

[21] Intel. TSX Asynchronous Abort. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort.html, November 2018.

[22] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[23] Intel. Microarchitectural Data Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html, May 2019.

[24] Intel. L1D Eviction Sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/l1d-eviction-sampling.html, January 2020.

[25] Intel. Load Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/load-value-injection.html, January 2020.

[26] Intel. Processor MMIO Stale Data Vulnerabilities. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/processor-mmio-stale-data-vulnerabilities.html, June 2022.

[27] Intel patent. Gather using index array and finite state machine . https://patents.google.com/patent/US8972697B2/, 2012.

[28] Intel patent. Speculative non-faulting loads and gathers. https://patents.google.com/patent/US9189236/, 2012.

[29] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*, 2019.

[30] Qian Ke, Chunlu Wang, Haixia Wang, Yongqiang Lyu, Zihan Xu, and Dongsheng Wang. Model checking for microarchitectural data sampling security. In *IEEE International Conference on Data Science in Cyberspace (DSC)*, 2022.

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

[32] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies*, 2018.

[33] Linux Kernel. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html, 2018.

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

[35] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[36] Michael Schwarz. PTEditor. https://github.com/misc0110/PTEditor.

[37] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2017.

[38] Daniel Moghimi. Data sampling on mds-resistant 10th generation intel core (ice lake). *arXiv preprint arXiv:2007.07428*, 2020.

[39] Daniel Moghimi. Medusa Code Repository. https://github.com/d4nielMoghimi/medusa, 2020.

[40] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security Symposium*, 2020.

[41] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level attacks on enclaves. In *USENIX Security Symposium*, 2020.

[42] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: testing black-box cpus against speculation contracts. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[43] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[44] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 1996.

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

[46] Phoronix. Fresh Linux 4.16 Kernel Benchmarks With KPTI & Retpolines. https://www.phoronix.com/review/linux-416early-spectremelt, March 2018.

[47] Phoronix. The Brutal Performance Impact From Mitigating The LVI Vulnerability. https://www.phoronix.com/review/lvi-attack-perf, March 2020.

[48] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[49] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[50] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*, 2015.

[51] Herbert Schildt. C the complete reference. *McGraw-Hill*, 2021.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[53] Youngjoo Shin. Multibyte microarchitectural data sampling and its application to session key extraction attacks. *IEEE Access*, 2021.

[54] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[55] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[56] Stephen Röttger. Escaping the Chrome Sandbox with RIDL . https://googleprojectzero.blogspot.com/2020/02/escaping-chrome-sandbox-with-ridl.html, February 2020.

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

[58] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[59] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

[60] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

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

[63] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX) Security 14)*, 2014.

[64] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) a comprehensive protection for speculatively accessed data. In *MICRO*, 2019.