Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks

Ben Gras
Vrije Universiteit Amsterdam

Kaveh Razavi
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Abstract

To stop side channel attacks on CPU caches that have allowed attackers to leak secret information and break basic security mechanisms, the security community has developed a variety of powerful defenses that effectively isolate the security domains. Of course, other shared hardware resources exist, but the assumption is that unlike cache side channels, any channel offered by these resources is insufficiently reliable and too coarse-grained to leak general-purpose information.

This is no longer true. In this paper, we revisit this assumption and show for the first time that hardware translation lookaside buffers (TLBs) can be abused to leak fine-grained information about a victim's activity even when CPU cache activity is guarded by state-of-the-art cache side-channel protections, such as CAT and TSX. However, exploiting the TLB channel is challenging, due to unknown addressing functions inside the TLB and the attacker's limited monitoring capabilities which, at best, cover only the victim's coarse-grained data accesses. To address the former, we reverse engineer the previously unknown addressing function in recent Intel processors. To address the latter, we devise a machine learning strategy that exploits high-resolution temporal features about a victim's memory activity. Our prototype implementation, TLBleed, can leak a 256-bit EdDSA secret key from a single capture after 17 seconds of computation time with a 98% success rate, even in presence of state-of-the-art cache isolation. Similarly, using a single capture, TLBleed reconstructs 92% of RSA keys from an implementation that is hardened against FLUSH+RELOAD attacks.

1 Introduction

Recent advances in micro-architectural side-channel attacks threaten the security of our general-purpose computing infrastructures from clouds to personal computers and mobile phones. These attacks allow attackers to leak secret information in a reliable and fine-grained way [13, 32, 36, 38, 59] as well as compromise fundamental security defenses such as ASLR [17, 20, 24, 28]. The most prominent class of side-channel attacks leak information via the shared CPU data or instruction caches. Hence, the community has developed a variety of powerful new defenses to protect shared caches against these attacks, either by partitioning them, carefully sharing them between untrusted programs in the system, or sanitizing the traces left in the cache during the execution [9, 21, 37, 52, 62].

In this paper, we argue that the problem goes much deeper. As long as there are other shared hardware resources, attackers can still reliably leak fine-grained, security-sensitive information from the system. In fact, we show this is possible even with shared resources that only provide a coarse-grained channel of information (whose general applicability has been questioned by prior work [46]), broadening the attack surface of practical side-channel attacks. To demonstrate this property, we present a practical side-channel attack that leaks information from the shared Translation Lookaside Buffers (TLBs) even in the presence of all the state-of-the-art cache defenses. Exploiting this channel is particularly challenging due its coarse (page-level) spatial granularity. To address this challenge, we propose a new analysis technique based on (supervised) machine learning. Our analysis exploits high-resolution temporal features on the victim's memory activity to combat side-channel coarsening and leak information.

Existing defenses against cache side channels The execution of a victim program changes the state of the shared CPU caches. In a cache side-channel attack, an attacker deduces sensitive information (e.g., cryptographic keys) by observing this change in the state. It is possible to rewrite existing software not to leave an identifiable trace in the cache, but manual approaches are errorprone [16] while automated ones incur several-fold performance overheads [48]. As an alternative, many proposed defenses attempt to stop attackers from observing changes that an unmodified victim program makes to the state of the CPU caches. This is done either by stopping precise timers that attackers need to use to tell the difference between cached or uncached memory accesses [10, 34, 40] or by partitioning shared CPU cache between mutually distrusting programs [21, 37, 47, 52, 62]. Given that attackers can find many new sources of timing [17, 34, 49], CPU cache partitioning is currently the only known generic mechanism that stops existing attacks.

Unfortunately, as we will show, protecting only the shared data and instruction caches is insufficient. Hardware threads (also known as hyperthreads) share other hardware resources such as TLBs on top of the CPU caches. The question we address in this paper is whether they can abused by attackers to leak sensitive information in a reliable and fine-grained way even in presence of state-of-the-art cache defenses and, if so, what the implications are for future attacks and defenses.

TLBleed To answer these questions, we explore the architecture of TLBs in modern Intel processors. As very little information on TLBs has been made available, our analysis represents the first known reverse engineering effort of the TLB architecture. Similar to CPU data and instruction caches, there are multiple levels of TLBs. They are partitioned in sets and behave differently based on whether they help in the translation of instructions or data. We further find that the mapping of virtual addresses to TLB sets is a complex function in recent micro-architectures. We describe our efforts in reverse engineering this function, useful when conducting TLB-based attacks and benefiting existing work [54]. Armed with this information, we build TLBleed, a side-channel attack over shared TLBs that can extract secret information from a victim program protected with existing cache defenses [9, 21, 31, 37, 52, 62] Implementing TLBleed is challenging: due to the nature of TLB operations, we can only leak memory accesses in the coarse granularity of a memory page (4 KB on x86 systems) and due to the TLB architecture we cannot rely on the execution of instructions (and controlled page faults) to leak secret information similar to previous page-level side-channel attacks [58]. To overcome these limitations, we describe a new machine learning-based analysis technique that exploits temporal patterns of the victim's memory accesses to leak information.

Contributions In summary, we make the following contributions:

- The first detailed analysis of the architecture of the TLB in modern processors including the previously unknown complex function that maps virtual addresses to TLB sets.
- The design and implementation of TLBleed, a new class of side-channel attacks that rely on the TLB to leak information. This is made possible by a new machine learning-based analysis technique based on temporal information about the victim's memory accesses. We show TLBleed breaks a 256-bit `libgcrypt` EdDSA key in presence of existing defenses, and a 1024-bit RSA key in an implementation that is hardened against FLUSH+RELOAD attacks.
- A study of the implications of TLBleed on existing attacks and defenses including an analysis of mitigations against TLBleed.

2 Background

To avoid the latency of off-chip DRAM for every memory access, modern CPUs employ a variety of caches [23]. With caching, copies of previously fetched items are kept close to the CPU in Static RAM (SRAM) modules that are organized in a hierarchy. We will focus our attention on data caches first and discuss TLBs after. For both holds that low-latency caches are partitioned into *cache sets* of *n ways*. This means is that in an *n* way cache, each set contains *n* cachelines. Every address in memory maps to exactly one cache set, but may occupy any of the *n* cachelines in this set.

2.1 Cache side-channel attacks

As cache sets are shared by multiple processes, the activity in a cache set offers a side channel for fine-grained, security-sensitive cache attacks. For instance, if the adversary first occupies all the *n* ways in a cache set and after some time observes that some of these cachelines are no longer in the cache (since accessing the data now takes much longer), it must mean that another program—a victim process, VM, or the kernel—has accessed data at addresses that also map to this cache set. Cache attacks by now have a long history and many variants [5, 33, 38]. We now discuss the three most common ones.

In a PRIME+PROBE attack [42, 43, 45], the adversary first collects a set of cache lines that fully evict a single cache set. By accessing these over and over, and measuring the corresponding access latency, it is possible to detect activity of another program in that particular cache set. This can be done for many cache sets. Due to the small size of a cache line, this allows for high spatial resolution, visualized in a *memorygram* in [42]. Closely related is FLUSH+RELOAD, which relies on the victim and the attacker physically sharing memory pages, so that the attacker can directly control the eviction (flushing) of a target memory page. Finally, an EVICT+TIME attack [17, 43, 53] evicts a particular cache set, then invokes the victim operation. The victim operation has a slowdown that depends on the evicted cache set, which leaks information on the activity of the victim.

2.2 Cache side-channel defenses

As a response to cache attacks, the research community has proposed defenses that follow several different strategies. We again discuss the most prominent ones here.

Isolation by partitioning sets Two processes that do not share a cache cannot snoop on each others' cache activity. One approach is to assign to a sensitive operation its own cache set, and not to let any other programs share that part. As the mapping from to a cache set involves the physical memory address, this can be done by the operating system by organizing physical memory into non-overlapping cache set groups, also called colors, and enforcing an isolation policy. This approach was first developed for higher predictability in real-time systems [7, 30] and more recently also for isolation for security [9, 31, 50, 62].

Isolation by partitioning ways Similarly to partitioning the cache by sets, we can also partition it by ways. In such a design, programs have full access to cache sets, but each set has a smaller number of ways, non-overlapping with other programs, if so desired. This approach requires hardware support such as Intel's Cache Allocation Technology (CAT) [37]. Since the number of ways and hence security domains is strictly limited on modern architectures, CATalyst's design uses only two domains and forbids accesses to the secure domain to prevent eviction of secure memory pages [37].

Enforcing data cache quiescence Another strategy to thwart cache attacks, while allowing sharing and hence not incurring the performance degradation of cache partitioning, is to ensure the quiescence of the data cache while a sensitive function is being executed. This protects against concurrent side channel attacks, including PRIME+PROBE and FLUSH+RELOAD, because these rely on evictions of the data cache in order to profile cache activity. This approach can be assisted by the Intel Transactional Synchronization Extensions (TSX) facility, as TSX transactions abort when concurrent data cache evictions occur [21].

2.3 From CPU caches to TLBs

All the existing cache side-channel attacks and defenses focus on exploitation and hardening of shared CPU caches, but ignore caching mechanisms used by the Memory Management Unit (MMU).

On modern virtual memory systems, such mechanisms play a crucial role. CPU cores primarily issue instructions that access data using their virtual addresses (VAs). The MMU translates these VAs to physical addresses (PAs) using a per-process data structure called the page table. For performance reasons, the result of these translations are aggressively cached in the Translation Lookaside Buffer (TLB). TLBs on modern Intel architectures have a two-level hierarchy. The first level (i.e., L1), consists of two parts, one that caches translations for code pages, called L1 instruction TLB (L1 iTLB), and one that caches translations for data pages, called L1 data TLB (L1 dTLB). The second level TLB (L2 sTLB) is larger and shared for translations of both code and data.

Again, the TLB at each level is typically partitioned into *sets* and *ways*, conceptually identical to the data cache architecture described earlier. As we will demonstrate, whenever the TLB is shared between mutually distrusting programs, this design provides attackers with new avenues to mount side-channel attacks and leak information from a victim even in the presence of state-of-the-art cache defenses.

3 Threat Model

We assume an attacker capable of executing unprivileged code on the victim system. Our attack requires monitoring the state of the TLB shared with the victim program. In native execution, this is simply possible by using CPU affinity system calls to achieve core co-residency with the victim process. In cloud environments, previous work shows it is possible to achieve residency on the same machine with a victim virtual machine [55]. Cloud providers may turn hyperthreading on for increased utilization (e.g., on EC2 [1]) making it possible to share cores across virtual machines. Once the attacker achieves core co-residency with the victim, she can mount a TLBleed attack using the shared TLB. This applies to scenarios where a victim program processing sensitive information, such as cryptographic keys.

4 Attack Overview

Figure 1 shows how an attacker can observe the TLB activity of a victim process running on a sibling hyperthread with TLBleed. Even if state-of-the-art cache side-channel defenses [21, 37, 47, 52, 62] are deployed and the activity of the victim process is properly isolated

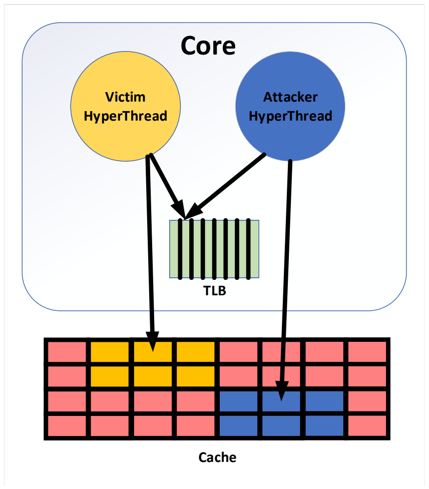

Figure 1: How TLBleed observes a sibling hyperthread's activity through the TLB even when shared caches are partitioned.

from the attacker with cache partitioning, TLBleed can still leak information through the shared TLB.

Mounting TLBleed on real-world settings comes with a number of challenges and open questions. The first set of challenges come from the fact that the architecture of the TLB is mostly secret. Mounting successful TLBleed, however, requires detailed knowledge of the TLB architecture. More specifically, we need to answer two questions:

Q1 How can we monitor TLB sets? More specifically, how do virtual addresses map to multi-level TLBs found in modern processors?

Q2 How do sibling hyperthreads share the TLB sets for translating their code and data addresses?

Once the attacker knows how to access the same TLB set as a victim, the question is whether she has the ability to observe the victim's activity:

Q3 How can an unprivileged process (without access to performance counters, TLB shootdown interrupts, etc.) monitor TLB activity reliably?

Finally, once the attacker can reliably measure the TLB activity of the victim, the question is whether she can exploit this new channel for attractive targets:

Q4 Can the attacker use the limited granularity of 4 kB "data" pages to mount a meaningful attack? And how will existing defenses such as ASLR complicate the attack?

We address these challenges in the following sections.

5 TLB Monitoring

To address our first challenge, **Q1**, we need to understand how virtual addresses (VAs) are mapped to different sets in the TLB. On commodity platforms, the mapping of VAs to TLB sets is microarchitecture-specific and currently unknown. As we shall see, we found that even on a single processor, the mapping algorithms in the different TLBs vary from very simple linear translations to complex functions that use a subset of the virtual address bits XORed together to determine the target TLB set.

To understand the details of how the TLB operates, we need a way to reverse engineer such mapping functions on commodity platforms, recent Intel microarchitectures in particular. For this purpose, we use Intel Performance Counters (PMCs) to gather fine-grained information on TLB misses at each TLB level/type. More specifically, we rely on the Linux `perf` event framework to monitor certain performance events related to the operation of the TLB, namely `dtlb_load_misses.stlb_hit` and `dtlb_load_misses.miss_causes_a_walk`. We create different access patterns depending on the architectural property under study and use the performance counters to understand how such property is implemented on a given microarchitecture. We now discuss our reverse engineering efforts and the results.

Linearly-mapped TLB We refer to the function that maps a virtual address to a TLB *set* as the *hash function*. We first attempt to find parameters under the hypothesis that the TLB is linearly-mapped, so that $target\ set = page_{VA} \mod s$ (with s the number of sets). Only if this strategy does not yield consistent results, we use the more generalized approach described in the next section.

To reverse engineer the hash function and the size of linearly-mapped TLBs, we first map a large set of testing pages into memory. Next, we perform test iterations to explore all the sensible combinations of two parameters: the number of sets s and the number of ways w. As we wish to find the smallest possible TLB eviction set, we use $w+1$ testing pages accessed at a stride of s pages. The stride is simply s due to the linear mapping hypothesis.

At each iteration, we access our testing pages in a loop and count the number of evictions evidenced by PMC counters. Observing that a minimum of $w+1$ pages is necessary to cause *any* evictions of previous pages, we note that the smallest w that causes evictions across all our iterations must be the right wayness w. Similarly, the smallest possible corresponding s is the right number of sets. As an example on Intel Broadwell, Figure 2 shows a heatmap depicting the number of evictions for each combination of stride s and number of pages w. The smallest w generating evictions is 4, and the smallest cor-

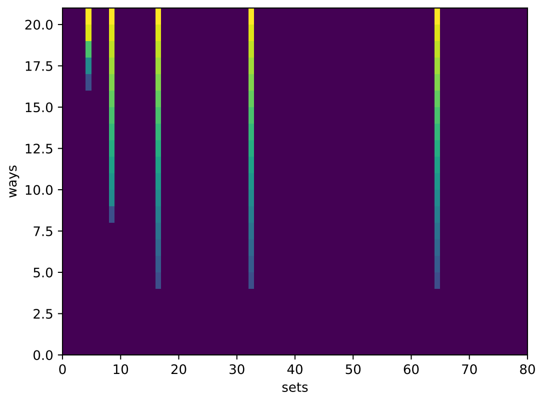

Figure 2: Linearly-mapped TLB probing on Intel Broadwell, evidencing a 4-way, 16-set L1 dTLB.

responding *s* is 16—correctly probing for a 4-way 16-set L1 dTLB on Broadwell.

Complex-mapped TLB If our results prove inconsistent with the linear mapping hypothesis, we must reverse engineer a more complex hash function to collect eviction sets (**Q1**). This is, for instance, the case for the L2 sTLB (L2 shared TLB) on our Skylake machine. Reverse engineering this function is analogous to identifying its counterpart for CPU caches, which decides to which cache set a physical address maps [26, 60]. Thus, we assume that the TLB set number can be expressed as an XOR of a subset of bits of the virtual address, similar to the physical hash function for CPU caches.

To reverse engineer the hash, we first collect minimal eviction sets, following the procedure from [42]. From a large pool of virtual addresses, this procedure gives us minimal sets that each map to a single hash set. Second, we observe that every address from the same eviction set must map to the same hash set via the hash function, which we hypothesized to be a XOR of various bits from the virtual address. For each eviction set and address, this gives us many constraints that must hold. By calculating all possible subsets of XOR-ed bit positions that might make up this function, we arrive at a unique solution. For instance, Figure 5 shows the hash function for Skylake's L2 sTLB. We refer to it as *XOR-7*, as it XORs 7 consecutive virtual address bits to find the TLB set.

Table 1 summarizes the TLB properties that our reverse engineering methodology identified. As shown in the table, most TLB levels/types on recent Intel microarchitectures use linear mappings, but the L2 sTLB on Skylake and Broadwell are exceptions with complex, XOR-based hash functions.

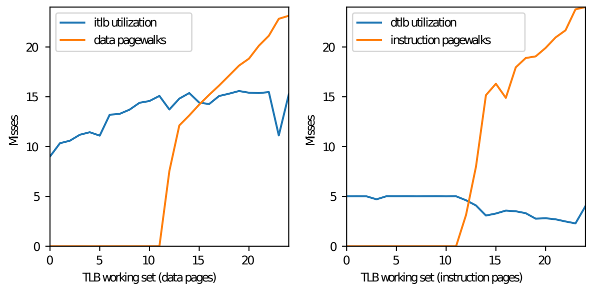

Figure 3: Skylake TLBs are not inclusive.

Interaction Between TLB Caches One of the central cache properties is inclusivity. If caches are inclusive, lower levels are guaranteed to be subsets of higher levels. If caches are not inclusive, cached items are guaranteed to be in at most one of the layers. To establish this property for TLBs, we conduct the following experiment:

1. Assemble a working set $S1$ that occupies part of a L1 TLB, and then the L2 TLB, until it is eventually too large for the L2 TLB. The pages should target only one particular L1 TLB (i.e., code or data).

2. Assemble a working set $S2$ of constant size that targets the other L1 TLB.

3. We access working sets $S1+S2$. We gradually grow $S1$ but not $S2$. We observe whether we see L1 misses of either type, and also whether we observe L2 misses.

4. If caches are inclusive, L2 evictions of one type will cause L1 evictions of the opposite type.

The result of our experiment is in Figure 3. We conclude TLBs on Skylake are not inclusive, as neither type of page can evict the other type from L1. This implicitly means that attacks that require L1 TLB evictions are challenging in absence of L1 TLB sharing, similar, in spirit, to the challenges faced by cache attacks in non-inclusive caching architectures [18].

With this analysis, we have addressed **Q1**. We now have a sufficient understanding of TLB internals on commodity platforms to proceed with our attack.

6 Cross-hyperthread TLB Monitoring

To verify the reverse engineered TLB partitions, and to determine how hyperthreads are exposed to each others' activity (addressing **Q2**), we run the following experiment for each TLB level/type:

1. Collect an eviction set that perfectly fills a TLB set.

Table 1: TLB properties per Intel microarchitecture as found by our reverse engineering methodology. hsh = hash function. w = number of ways. pn = miss penalty in cycles, shr indicates whether the TLB is shared between threads.

		L1 dTLB					L1 iTLB					L2 sTLB				
Name	year	set	w	pn	hsh	shr	set	w	pn	hsh	shr	set	w	pn	hsh	shr
Sandybridge	2011	16	4	7.0	lin	✓	16	4	50.0	lin	✗	128	4	16.3	lin	✓
Ivybridge	2012	16	4	7.1	lin	✓	16	4	49.4	lin	✗	128	4	18.0	lin	✓
Haswell	2013	16	4	8.0	lin	✓	8	8	27.4	lin	✗	128	8	17.1	lin	✓
HaswellXeon	2014	16	4	7.9	lin	✓	8	8	28.5	lin	✗	128	8	16.8	lin	✓
Skylake	2015	16	4	9.0	lin	✓	8	8	2.0	lin	✗	128	12	212.0	XOR-7	✓
BroadwellXeon	2016	16	4	8.0	lin	✓	8	8	18.2	lin	✗	256	6	272.4	XOR-8	✓
Coffeelake	2017	16	4	9.1	lin	✓	8	8	26.3	lin	✗	128	12	230.3	XOR-7	✓

2. For each pair of eviction sets, access one set on one hyperthread and the other set on another hyperthread running on the same core.

3. Measure the observed evictions to determine whether one given set interferes with the other set.

Figure 4 presents our results for Intel Skylake, with a heatmap depicting the number of evictions for each pair of TLB (and corresponding eviction) sets. The lighter colors indicate a higher number of TLB miss events in the performance counters, and so imply that the corresponding set was evicted. A diagonal in the heatmap shows interference between the hyperthreads. If thread 1 accesses a set and thread 2 accesses the same set, they interfere and increase the miss rate. The signals in the figure confirm our reverse engineering methodology was able to correctly identify the TLB sets for our Skylake testbed microarchitecture. Moreover, as shown in the figure, only the L1 dTLB and the L2 sTLB show a clear interference between matching pairs of sets, demonstrating that such TLB levels/types are shared between hyperthreads while the L1 iTLB does not appear to be shared. The signal on the diagonal in the L1 dTLB shows that a given set is shared with the exact same set on the other hyperthread. The signal on the diagonal in the L2 sTLB shows that sets are shared but with a 64-entry offset—the highest set number bit is XORred with the hyperthread ID when computing the set number. The spurious signals in the L1 dTLB and L1 iTLB charts are sets representing data and code needed by the instrumentation and do not reflect sharing between threads. This confirms statements in [11] that, since the Nehalem microarchitecture, "L1 iTLB page entries are statically allocated between two logical processors', and "DTLB0 and STLB" are a "competitively-shared resource." We verified that our results also extend to all other microarchitectures we considered (see Table 1).

With this analysis we have addressed **Q2**. We can now use the L1 dTLB and the L2 sTLB (but not the L1 iTLB) for our attack. In addition, we cannot easily use the L2 sTLB for code attacks, as with non-inclusive TLBs and non-shared L1 iTLB triggering L1 evictions is challenging, as discussed earlier. This leaves us with data attacks on the L1 dTLB or L2 sTLB.

7 Unprivileged TLB Monitoring

While performance counters can conveniently be used to reverse engineer the properties of the TLB, accessing them requires superuser access to the system by default on modern Linux distributions, which is incompatible with our unprivileged attacker model. To address **Q3**, we now look at how an attacker can monitor the TLB activity of a victim without any special privilege by just *timing memory accesses.*

We use the code in Figure 6, designed to monitor a 4-way TLB set, to exemplify our approach. As shown in the figure, the code simply measures the latency when accessing the target eviction set. This is similar, in spirit, to the PROBE phase of a classic PRIME+PROBE cache attack [42, 43, 45], which, after priming the cache, times the access to a cache eviction set to detect accesses of the victim to the corresponding cache set. In our TLB-based attack setting, a higher eviction set access latency indicates a likely TLB lookup performed by the victim on the corresponding TLB set.

To implement an efficient monitor, we time the accesses using the `rdtsc` and `rdtscp` instructions and serialize each memory access with the previous one. This is to ensure the latency is not hidden by parallelism, as each load is dependent on the previous one, a technique also seen in [43] and other previous efforts. This pointer chasing strategy allows us to access a full eviction set without requiring full serialization after every load. The `lfence` instructions on either side make it unnecessary to do a full pipeline flush with the `cpuid` instruction, which makes the operation faster.

With knowledge of the TLB structure, we can design an experiment that will tell us whether the latency reliably indicates a TLB hit or miss or not. We proceed as follows:

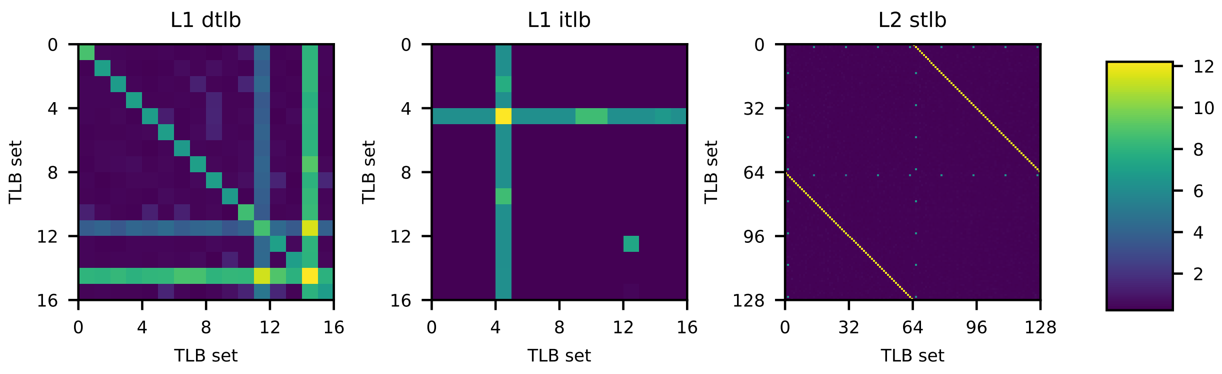

Figure 4: Interaction of TLB sets between hyperthreads on Intel Skylake. This shows that the L1 dTLB and the L2 sTLB are shared between hyperthreads, whereas this does not seem to be the case for the L1 iTLB.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Skylake L2 sTLB's hash function (H), which converts a virtual address VA to a L2 sTLB set with the matrix multiplication $H \cdot VA[26:12]$, where $VA[26:12]$ represent the next 14 lowest bits of VA after the 12 lowest bits of VA. We call this function *XOR-7*, because it XORs 7 consecutive virtual address bits. We have observed a similar *XOR-8* function on Broadwell.

Figure 6: Timed accesses used to monitor a 4-way TLB set with pointer chasing.

```
uint64_t probe; /* probe addr */
uint32_t time1,time2;

asm volatile (
"lfence\n"
"rdtsc\n"
"mov %%eax, %%edi\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"lfence\n"
"rdtscp\n"
"mov %%edi, %0\n"
"mov %%eax, %1\n"
  : "=r" (time1), "=r" (time2)
  : "r" (probe)
  : "rax", "rbx", "rcx",
    "rdx", "rdi");
```

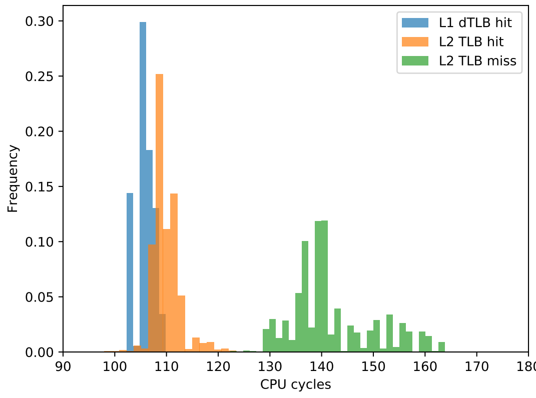

Figure 7: Memory access latency determining TLB hit or misses. The mapped physical page is always the same one, and so always in the cache, so the latency of the memory access purely depends on TLB latency.

1. We assemble three working sets. The first stays entirely within L1 dTLB. The second misses L1 partially, but stays inside L2. The third set is larger than L2 and will so force a page table walk.

2. The eviction sets are virtual addresses, which we all map to the same physical page, thereby avoiding noise from the CPU data cache.

3. Using the assembly code we developed, we access these eviction sets. If the latency predicts the category, we should see a clear separation.

We take the Skylake platform as an example. The result of our experiment can be seen in Figure 7. We see a multi-modal distribution, clearly indicating that we can use unprivileged instructions to profile TLB activity.

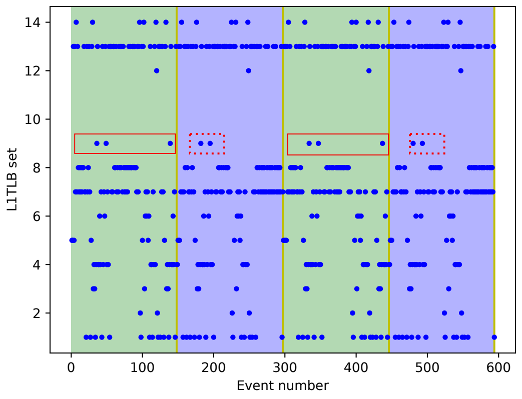

Figure 8: Page-level access patterns of data of an ECC point multiplication routine. The background is the ground truth of the cases we wish to distinguish. The rectangles show temporally unique patterns that make this possible.

Figure 9: Elliptic curve point multiplication in libgcrypt. We attack the non-constant-time half of the branch.

```
void
_gcry_mpi_ec_mul_point (mpi_point_t result,
 gcry_mpi_t scalar, mpi_point_t point,
 mpi_ec_t ctx)
{
 gcry_mpi_t x1, y1, z1, k, h, yy;
 unsigned int i, loops;
 mpi_point_struct p1, p2, p1inv;
...
 if (mpi_is_secure (scalar)) {
/* If SCALAR is in secure memory we assume that it
   is the secret key we use constant time operation.
*/
     ...
 } else {
  for (j=nbits-1; j >= 0; j--) {
   _gcry_mpi_ec_dup_point (result, result, ctx);
   if (mpi_test_bit (scalar, j))
    _gcry_mpi_ec_add_points(result,result,point,ctx);
  }
 }
}
```

Our analysis here addresses **Q3**. We can now rely on unprivileged memory access latency measurements to reliably distinguish TLB misses from TLB hits and hence monitor the activity of the victim over shared TLBs in practical settings.

8 Temporal Analysis

Given the monitoring logic we developed in Section 5, we now turn to **Q4**—how can we leak information with a page-granular signal for data pages only? When targeting sensitive cryptographic applications, previous work on controlled channels focused on leaking the secret using code pages due to the difficulty of extracting secrets using page-granular data accesses [58]. Data pages are only used for synchronization purposes in the attack. In other words, this is a non-trivial challenge, especially given our side-channel rather than controlled-channel attack scenario.

To investigate the extent of this challenge, we pick an example target, `libgcrypt`, and target its elliptic curve cryptography (ECC) multiplication function, shown in Figure 9. This function will be used in a signing operation, where `scalar` is a secret. We use the non-constant-time version in this work. We instrument the code with the Intel Pin Dynamic Binary Instrumentation framework [39].

Figure 8 shows the observed activity in each of the 16 L1 dTLB sets over time. The two background colors differentiate between data accesses of the two different functions, namely the function that performs a duplication operation and one that performs an addition operation depending on a single bit in the private key as shown in a code snippet taken from `libgcrypt`. If we can differentiate between the TLB operations of these two functions, we can leak the secret private key. It is clear that the same sets are always active in both sides of the branch, making it impossible to leak bits of the key by just monitoring which sets are active a la PRIME+PROBE. Hence, due to (page-level) side-channel coarsening, TLB attacks cannot easily rely on traditional spatial access information to leak secrets in real-world attack settings.

Looking more carefully at Figure 8, it is clear that some sets are accessed at *different times* within the execution of each side of the branch. For example, it is clear that the data variables that map to TLB set 9 are being accessed at different times in the different sides of the branch. The question is whether we can use such timings as distinguishing features for leaking bits of data from `libgcrypt`'s ECC multiplication function. In other words, we have to rely on temporal accesses to the TLB sets instead of the commonly-used spatial accesses for the purposes of leaking information.

To investigate this approach, we now look at signal classification for the activity in the TLB sets. Furthermore, in the presence of address-space layout randomization (ASLR), target data may map to different TLB sets. We discuss how we can detect the TLB sets of interest using a similar technique.

Signal classification Assuming availability of latency measurements from a target TLB set, we want to distinguish the execution of different functions that access the target TLB set at different times. For this purpose, we train a classifier that can distinguish which function is being executed by the victim, as a function of observed TLB latencies. We find that, due to the high resolution of our channel, a simple classification and feature extraction strategy is sufficient to leak our target functions' temporal traces with a high accuracy. We discuss what more may be possible with more advanced learning techniques and the implications for future cache attacks and defenses in Section 10. We now discuss how we trained our classifier.

To collect the ground truth, we instrument the victim with statements that record the state of the victim's functions, that is how the classifier should classify the current state. This information is written to memory and shared with our TLB monitoring code developed in Section 5. We run the monitoring code on the sibling hyperthread of the one that executes the instrumented victim. Our monitoring code uses the information provided by the instrumented victim to measure the activity of the target TLB set for each of the two functions that we wish to differentiate.

To extract suitable features from the TLB signal, we simply encode information about the activity in the targeted TLB set using a vector of normalized latencies. We then use a number of such feature vectors to train a Support Vector Machine (SVM) classifier, widely used nowadays for general-purpose classification tasks [12]. We use our SVM classifier to solve a three-class classification problem: distinguishing accesses to two different functions (class-1 and class-2) and other arbitrary functions (class-3) based on the collected TLB signals. The training set consists of a fixed number (300) of observed TLB latencies starting at a function boundary (based on the ground truth). We find the normalizing the amplitude of the latencies prior to training and classification to be critical for the performance of our classifier. For each training sample, we normalize the latencies by subtracting the mean latency and dividing by the standard deviation of the 300 latencies in the training sample.

We use 8 executions to train our SVM classifier. On average, this results in 249 executions of the target duplication function, and 117 executions of the target addition function, leading to 2,928 training samples of function boundaries. After training, the classifier can be used on target executions to extract function signatures and reconstruct the target private key. We report on the performance of the classifier and its effect on the end-to-end TLBleed attack on `libgcrypt` in Section 9.2.

As an example of the classifier in action on the raw signal, see Figure 10. It has been trained on the latency values, and can reliably detect the 2 different function boundaries. We use a peak detection algorithm to derive the bit stream from the classification output. The moving average is not used by the classifier, but is shown to make the signal discrepancy more apparent to human inspection. The peak detection merges spurious peaks/-valleys into one as seen in the first valley, and turns the continuous classification into a discrete bitstream.

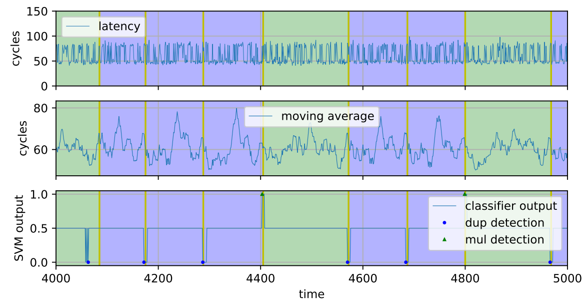

Figure 10: SVM signal classification on raw latency data. The background shade represents ground truth; either the execution of the 'dup' function (0) or the 'mul' function (1). The classifier properly classifies signal boundaries from raw latency data; either the start of a dup (0), mul (1) or not a boundary (0.5). The peak detection converts the continuous classifications into discrete single detections.

Identifying the Target TLB Set For the `libgcrypt` target, we only need to use a single TLB set for training and testing. For the purpose of training our classifier, we assume that this information is known. During a real-world attack, however, we cannot know the target TLB set beforehand, due to virtual address randomization performed by ASLR.

Nonetheless, our hypothesis is that each of the TLB sets behave differently during the execution of our target program. Hence, we can follow the same approach of classifying behavior based on the temporal activity of each of the sets to distinguish the target set. In other words, in a preliminary step, we can now use our SVM classifier to solve a s-class classification problem, where each class represents TLB signals for a particular TLB set and we want to identify TLB signals that belong to the "target" class of interest. To validate our hypothesis, we run this step for the same period as we do for the attack, when the ECC point multiplication occurs. We find that this simple strategy already results in a classifier that can distinguish the TLB sets. Section 9.1 evaluates the reliability and performance of our target TLB set detection technique.

We can now mount an end-to-end attack using a simple classification and feature extraction strategy, as well as a preliminary step to identify the victim TLB set in spite of ASLR.

9 Evaluation

In this section we select a challenging case study, and evaluate the reliability of TLBleed.

Testbed To gain insights on different recent micro-architectures, we evaluated TLBleed on three different systems: (i) a workstation with an Intel Skylake Core i7-6700K CPU and 16 GB of DDR4 memory, (ii) a server with an Intel Broadwell Xeon E5-2620 v4 and 16 GB of DDR4 memory, and (iii) a workstation with an Intel Coffeelake Core i7-8700 and 16 GB of DDR4 memory. We mention which system(s) we use for each experiment.

Overview of the results We first target libgcrypt's Curve 25519 EdDSA signature implementation. We use a version of the code that is not written to be constant-time. We first show that our classifier can successfully distinguish the TLB set of interest from other TLB sets (Section 9.1). We then evaluate the reliability of the TLBleed attack (Section 9.2). On average, TLBleed can reconstruct the private key in 97% of the case using *only a single signature generation capture and in only 17 seconds*. In the remaining cases, TLBleed significantly compromises the private key. Next we perform a similar evaluation on RSA code implemented in libgcrypt, that was written to be constant-time in order to mitigate FLUSH+RELOAD [59], but nevertheless leaves a secret-dependent data trace. The implementation has since been improved, already before our work. We then evaluate the security of state-of-the-art cache defenses in face of TLBleed. We find that TLBleed is able to leak information even in presence of strong, hardware-based cache defenses (Section 9.5 and Section 9.6). Finally, we construct a covert channel using the TLB, to evaluate the resistance of TLBleed to noise (Section 9.7).

9.1 TLB set identification

To show all TLB sets behave in a sufficiently unique way for TLBleed to reliably differentiate them, we show our classifier trained on all the different TLB sets recognizing test samples near-perfectly. After training a classifier on samples from each of the 16 L1 dTLB access patterns in *libgcrypt*, we are able to distinguish all TLB sets from each other with an F1-score of 0.54, as shown in a reliability matrix in Figure 11. We observe no false positives or false negatives to find the desired TLB set across repeated runs. We hence conclude that TLBleed is effective against ASLR in our target application. We further discuss the implications of TLB set identification on weakening ASLR in Section 10.

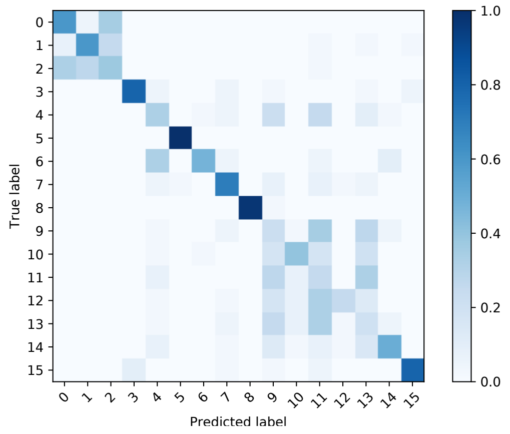

Figure 11: Classification reliability for distinguishing TLB sets using temporal access patterns. For all active TLB sets during our target operation, we can reliably determine where they are mapped in the virtual address space.

9.2 Compromising EdDSA

Curve 25519 EdDSA signature algorithm in libgcrypt v1.6.3 is a high-performance elliptic curve algorithm [6]. To demonstrate TLBleed determining a key by just monitoring the TLB, we attack the non-constant-time version of this code. This would still be safe when cache isolation is deployed.

As shown previously in Figure 9, we are interested in distinguishing between the duplication (i.e., _gcry_mpi_ec_dup_point) and addition (i.e., _gcry_mpi_ec_add_points) operations, so that we can distinguish key bits in the secret used in the signature. There will always be a dup invocation for every bit position in the execution trace, plus an average of 128 add invocations somewhere for every '1' bit in the secret value. As keys are 256 bits in Curve 25519, on average we observe 384 of these operations.

Hence, we must be able to distinguish the two operations with high reliability. Errors in the classification require additional bruteforcing on the attacker's side to compensate. As misclassification errors translate to arbitrary bit edit operations in the secret key, bruteforcing quickly becomes intractable with insufficient reliability.

We follow a two step approach in evaluating TLBleed on libgcrypt. We first collect the activities in the TLB for *only 2 ms* during a single signing operation. Our classifier then uses the information in this trace to find the TLB set of interest and to classify the duplication and addition operations for leaking the private key. In the second step, we try to compensate for classification errors using a number of heuristics to guide bruteforcing in exhausting the residual entropy. We first discuss the

Table 2: Success rate of TLBleed on various microarchitectures. The success rate is a count of the number of successful full key recoveries, with some brute forcing (BF) attempts. Unsuccessful cases were out of reach of bruteforcing.

Micro-architecture	Trials	Success	Median BF
Skylake	500	0.998	$2^{1.6}$
Broadwell	500	0.982	$2^{3.0}$
Coffeelake	500	0.998	$2^{2.6}$
Total	1500	0.993	

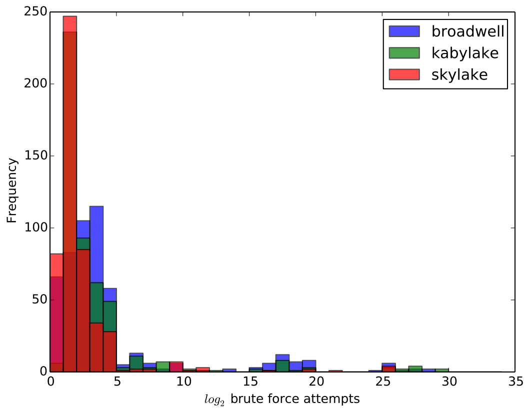

Figure 12: Required number of bruteforcing attempts for compromising 256-bit EdDSA encryption keys with TLBleed.

results and then elaborate on the bruteforcing heuristics that we use.

Table 2 shows the results of our attack on all testbeds. With a small number of measurements-guided bruteforcing, TLBleed can successfully leak the key in 99.8% of the cases in the Skylake system, in 98.2% of the cases on the Broadwell system, and 99.8% on Coffeelake. In the remaining cases, while the key is significantly compromised, bruteforcing was still out of reach with our available computing resources. The end-to-end attack time is composed of: 2 ms of capture time; 17 seconds of signals analysis with the trained classifier; and a variable amount of brute-force guessing with a negligible median work factor of 2^3 at worst, taking a fraction of a second. Thus, in the most common case, the end-to-end attack time is dominated by the signals analysis phase of 17 seconds and can be trivially reduced with more computing resources. Given that TLBleed requires a very small capture time, existing re-randomization techniques (e.g., Shuffler [57]) do not provide adequate protection against TLBleed, even if they re-randomized both code and data.

Figure 13: Sketched representation of SIMPLE_EXPONENTIATION variant of modular exponentiation in `libgcrypt`, in an older version.

```
void
_gcry_mpi_powm (gcry_mpi_t res,
   gcry_mpi_t base, gcry_mpi_t expo, gcry_mpi_t mod)
{
   mpi_ptr_t rp, xp; /* pointers to MPI data */
   mpi_ptr_t tp;
   ...
   for(;;) {
     ...
     /* For every exponent bit in expo: */
     _gcry_mpih_sqr_n_basecase(xp, rp);
     if(secret_exponent || e_bit_is1) {
      /* Unconditional multiply if exponent is
       * secret to mitigate FLUSH+RELOAD.
       */
      _gcry_mpih_mul (xp, rp);
     }
     if(e_bit_is1) {
       /* e bit is 1, use the result */
       tp = rp; rp = xp; xp = tp;
       rsize = xsize;
     }
   }
}
```

Figure 12 provides further information on the frequency of bruteforcing attempts required after classification. We rely on two heuristics based on the classification results to guide our bruteforcing attempts. Due to the streaming nature of our classifier, sometimes it does not properly recognize a 1 or a 0, leaving a blank (i.e., *skipping*), and sometimes it classifies two 1s or two 0s instead of only one (i.e., *duplicating*). By looking at the length of periods in which the classifier makes decision, we can find cases where the period is too long for a single classification (skipping) and cases where the period is too short for two classifications (duplicating). In the case of skipping, we try to insert a guess bit and in the case of duplicating, we try to remove the duplicate. As evidenced by our experimental results, these heuristics work quite well for dealing with misclassifications in the case of the TLBleed attack.

9.3 Compromising RSA

We next show that an RSA implemenetation, written to mitigate FLUSH+RELOAD [59], nevertheless leaves a secret-dependent data trace in the TLB that TLBleed can detect. This finding is not new to our work and this version has since been improved. Nevertheless we show TLBleed can detect secret key bits from such an RSA implementation, even when protected with cache isolations deployed, as well as code hardening against FLUSH+RELOAD.

Listing 13 shows our target RSA implemenatation.

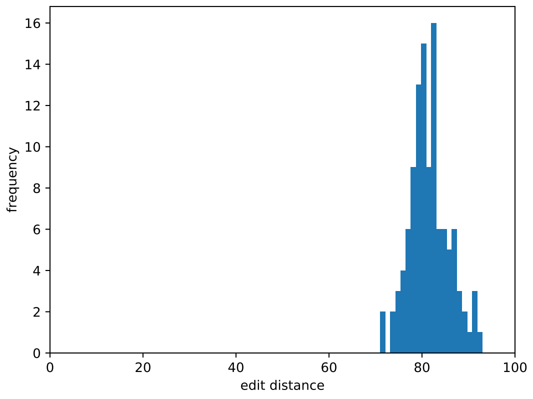

Figure 14: TLBleed accuracy in computing RSA 1024-bit secret exponent bits. Shown is the histogram of the number of errors the reconstructed RSA exponent contained from a single capture, expressed as the Levenshtein edit distance.

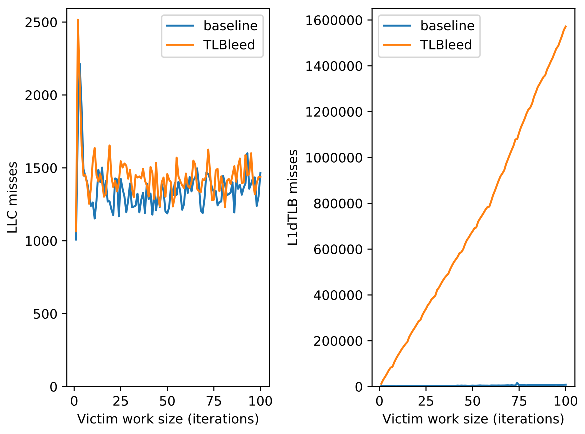

Figure 15: TLBleed compromising software defenses, as demonstrated by the substantial number of TLB rather than cache misses required by our `libgcrypt` attack.

The code maintains pointers to the result data (rp) and working data (xp). This is a schematic representation of modular exponentiation code as it existed in older versions of `libgcrypt`, following a familiar square-and-multiply algorithm to compute the modular exponentiation. The multiplication should only be done if the corresponding exponent bit is 1. Conditionally executing this code leaks information about the secret exponent, as shown in [59]. To mitigate this, the code unconditionally executes the multiplication but conditionally uses the result, by swapping the rp and xp pointers if the bit is 1. Whenever these pointers fall in different TLB sets, TLBleed can detect whether or not this swapping operation has happened, by distinguishing the access activity in the swapped and unswapped cases, directly leaking information about the secret exponent.

We summarize the accuracy of our key reconstruction results in Figure 14, a histogram of the edit distance of the reconstructed RSA keys showing that on average we recover more than 92% of RSA keys with a single capture. While we have not upgraded these measurements to a full key recovery, prior work [61] has shown that it is trivial to reconstruct the full key from 60% of the recovered key by exploiting redundancies in the storage of RSA public keys [22].

9.4 Compromising Software Defenses

Software-implemented cache defenses all seek to prevent an attacker to operate cache evictions for the victim's cachelines. Since TLBleed only relies on TLB evictions and is completely oblivious to cache activity, our attack strategy trivially bypasses such defenses. To confirm this assumption, we repeat our `libgcrypt` attack for an increasing number of iterations to study the dependency between victim activity and cache vs. TLB misses.

Table 3: TLBleed compromising Intel CAT.

Microarchitecture	Trials	Success	Median BF
Broadwell (CAT)	500	0.960	$2^{2.6}$
Broadwell	500	0.982	$2^{3.0}$

Figure 15 presents our results. As shown in the figure, the TLBleed has no impact on the cache behavior of the victim (LLC shown in figure, but we observed similar trends for the other CPU caches). The only slight increase in the number of cache misses is a by-product of the fast-growing number of TLB misses required by TLBleed and hence the MMU's page table walker more frequently accessing the cache. Somewhat counter-intuitively, the increase in the number of cache misses in Figure 15 is still constant regardless of the number of TLB misses reported. This is due to high virtual address locality in the victim, which translates to a small, constant cache working set for the MMU when handling TLB misses. This experiment confirms our assumption that TLBleed is oblivious to the cache activity of the victim and can trivially leak information in presence of state-of-the-art software-implemented cache defenses.

9.5 Compromising Intel CAT

We now want to assess whether TLBleed can compromise strong, hardware-based cache defenses based on hardware cache partitioning. Our hypothesis is that such

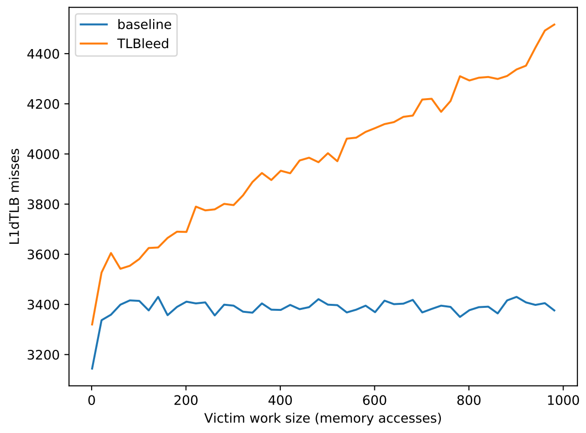

Figure 16: TLBleed detects TLB activity of a victim process running inside an Intel TSX transaction by stealthily measuring TLB misses.

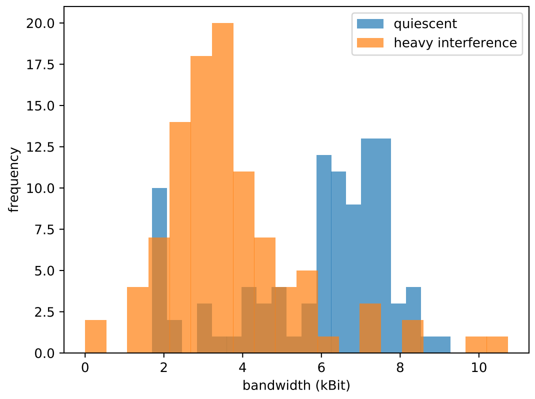

Figure 17: TLB covert channel bandwidth without and with a heavy interference load. The undetected frame error rate is low in both cases: $2.0 \cdot 10^{-5}$ and $3.2 \cdot 10^{-4}$ respectively.

hardware mechanisms do not extend their partitioning to the TLB. Our Broadwell processor, for example, is equipped with the Intel CAT extension, which can partition the shared cache between distrusting processes [37]. To validate our hypothesis, our goal is to show that TLBleed can still leak information even when Intel CAT is in effect.

We repeat the same experiment we used to attack `libgcrypt`, but this time with Intel CAT enabled. We isolate the victim `libgcrypt` process from the TLBleed process using Intel `rdtset` tool by perfectly partitioning the cache between the two processes (using the `0xf0` mask for the victim, and `0x0f` for TLBleed). Table 3 shows that the hardware cache partitioning strategy implemented by Intel CAT does not stop TLBleed, validating our hypothesis. This demonstrates TLBleed can bypass state-of-art defenses that rely on Intel CAT (or similar mechanisms) [37].

9.6 Compromising Intel TSX

We now want to assess whether TLBleed can compromise strong, hardware-based cache defenses that protect the cache activity of the victim with hardware transactional memory features such as Intel TSX. In such defenses, attacker-induced cache evictions induce Intel TSX capacity aborts, detecting the attack [21]. Our hypothesis is that such hardware mechanisms do not extend their abort strategy to TLB evictions. To validate our hypothesis, our goal is to show that TLBleed can still detect the victim's activity with successful transactions and leak information even when Intel TSX is in effect.

Porting `libgcrypt`'s EdDSA algorithm to run inside a TSX transaction requires major source changes since its working set does not fit inside the CPU cache. We instead experiment with a synthetic but representative example, where a victim process accesses a number of memory addresses in a loop for a given number of times inside a transaction.

Figure 16 shows the number of TLB misses with and without TLBleed. Increasing the duration of victim's execution allows TLBleed to detect more and more TLB miss due to the victim's activity. Each additional miss provides TLBleed with information about the secret operation of a victim without aborting the transaction, validating our hypothesis. This demonstrates TLBleed can also bypass recent defenses that rely on Intel TSX (or similar mechanisms) [21] and, ultimately, all the state-of-the-art cache defenses.

9.7 TLB Covert Channel

To further prove the correct reverse engineering of TLB properties, and to do a basic quantification of the noise resistance properties of this channel, we use our new TLB architecture knowledge to construct a covert channel. This allows communication between mutually co-operating parties that are not authorized to communicate, e.g. to exfiltrate data. We exclusively use the TLB and no other micro-architectural state for this channel. For the purposes of this design, TLB sets and cache sets serve the same purpose: accessing the set gives the other party a higher latency in the same set, which we use as a communication primitive. We borrow design ideas from [41].

We implement this covert channel and do two experiments. The first we run the protocol with a transmitter and receiver on two co-resident hyperthreads on an otherwise quiescent machine. The second we do the same, but generate two heavy sources of interference: one, we run the libgcrypt signing binary target in a tight loop on the same core; and two, we run `stress -m 5` to generate a high rate of memory activity throughout the machine.

We find the usable bandwidth under intense load is roughly halved, and the rate of errors that was not caught by the framing protocol does increase, but remains low. We see an undetected frame error rate of $2.0 \cdot 10^{-5}$ for a quiescent machine, and $3.2 \cdot 10^{-4}$ for the heavily loaded machine. These results are summarized in Figure 17 and show robust behaviour in the presence of heavy interference. We believe that, given the raw single TLB set probe rate of roughly $30 \cdot 10^{7}$, with additional engineering effort the bandwidth of this channel could be significantly improved.

10 Discussion

Leaking cryptographic keys and bypassing cache side-channel defenses are not the only possible targets for TLBleed. Moreover, mitigating TLBleed without support from future hardware is challenging. We discuss these topics in this section.

10.1 Other targets

TLBleed can potentially leak other information whenever TLBs are shared with a victim process. We expect that our TLB set classification technique can very quickly reduce the entropy of ASLR, either that of the browser [8, 17] or kernel [20, 24, 29]. The L2 TLB in our Broadwell system has 256 sets, allowing us to reduce up to 8 bits of entropy. Note that since the TLB is shared, separating address spaces [19] will not protect against TLBleed.

Other situations where TLBleed may leak information stealthily are from Intel SGX enclaves or ARM TrustZone processes. We intend to pursue this avenue of research in the future.

10.2 Mitigating TLBleed

The simplest way to mitigate TLBleed is by disabling hyperthreads or by ensuring in the operating system that sensitive processes execute in isolation on a core. However, this strategy inevitably wastes resources. Furthermore, in cloud environments, customers cannot trust that their cloud provider's hardware or hypervisor has deployed a (wasteful) mitigation. Hence, it is important to explore other mitigation strategies against TLBleed.

In software, it may be possible to partition the TLB between distrusting processes by partitioning the virtual address space. This is, however, challenging since almost all applications rely on contiguous virtual addresses for correct operations, which is no longer possible if certain TLB sets are not accessible due to partitioning.

It is easier to provide adequate protection against TLBleed in hardware. Intel CAT, for example, can be extended to provide partitioning of TLB ways on top of partitioning cache ways. Existing defenses such as CATalyst [37] can protect themselves against TLBleed by partitioning the TLB in hardware. Another option is to extend hardware transactional memory features such as Intel TSX to cause capacity aborts if a protected transaction observes unexpected TLB misses similar to CPU caches. Existing defenses such as Cloak [21] can then protect themselves against TLBleed, since an ongoing TLBleed attack will cause unexpected aborts.

11 Related Work

We focus on closely related work on TLB manipulation and side-channel exploitation over shared resources.

11.1 TLB manipulation

There is literature on controlling TLB behavior in both benign and adversarial settings. In benign settings, controlling the impact of the TLB is particularly relevant in real-time systems [27, 44]. This is to make the execution time more predictable while keeping the benefits of a TLB. In adversarial settings, the TLB has been previously used to facilitate exploitation of SGX enclaves. In particular, Wang et al. [56] showed that it is possible to bypass existing defenses [51] against controlled channel attacks [58] by flushing the TLB to force page table walks without trapping SGX enclaves. In contrast, TLBleed leaks information by directly observing activity in the TLB sets.

11.2 Exploiting shared resources

Aside from the cache attacks and defenses extensively discussed in Section 2.1, there is literature on other microarchitectural attacks exploiting shared resources. Most recently, Spectre [32] exploits shared Branch Target Buffers (BTBs) to mount "speculative" control-flow hijacking attacks and control the speculative execution of the victim to leak information. Previously, branch prediction has been attacked to leak data or ASLR information [2, 3, 14, 35]. In [4], microarchitectural properties of execution unit sharing between hyperthreads is analyzed. Finally, DRAMA exploits the DRAM row buffer to mount (coarse-grained) cross-CPU side-channel attacks [46].

11.3 Temporal side-channel analysis

A number of previous efforts have observed that temporal information can be used to mount side-channel attacks over shared caches or similar fine-grained channels [4, 15, 25, 38, 45, 61]. With TLBleed, we introduce a machine learning-based analysis framework that exploits (only) high-resolution temporal features to leak information even in (page-level) side-channel coarsening scenarios. Nonetheless, our approach is generic and hence applicable to other attack settings, where an attacker targets either fine-grained (e.g., cache) or even more coarse-grained (e.g., DRAM) channels.

12 Conclusion

TLBleed, a powerful and fundamentally new side channel attack via the TLB, shows that the problem of microarchitectural side channels goes much deeper than previously assumed. So far, much of the community has implicitly assumed that practical, fine-grained side-channel attacks are limited to the CPU data and instruction caches, leaving most other shared resources out of the threat model. In this paper, we have shown that TLB activity monitoring not only offers a practical new side channel, but also that it bypasses all the state-of-the-art cache side-channel defenses. Since the operation of the TLB is a fundamental hardware property, mitigating TLBleed is challenging. It requires novel research to design efficient yet flexible mechanisms that isolate TLB partitions based on the corresponding security domains. However, it is not unlikely that as new mitigations are developed, new side channels amenable to practical attacks emerge. As a more general lesson, TLBleed demonstrates that comprehensive side-channel protection should carefully consider *all* shared resources.

Acknowledgements

The authors would like to thank the anonymous reviewers for their thoughtful feedback. We would also like to thank Colin Percival, Yuval Yarom, and Taylor 'Riastradh' Campbell for feedback on early versions of this paper. The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 786669 and was supported in part by the MALPAY project and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax".

References

[1] Amazon ec2 instance types: Each vcpu is a hyperthread of an intel xeon core except for t2. `https://aws.amazon.com/ec2/instance-types/`, Accessed on 28.06.2018., 2016.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, pages 185–203. Springer, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[4] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[5] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[6] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[7] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices*, volume 29, pages 158–170. ACM, 1994.

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. *S&P (May. 2016)*, 2016.

[9] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[10] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–178. ACM, 2017.

[11] Intel Coorporation. Intel 64 and ia-32 architectures optimization reference manual, 2016.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx. 2017.

[14] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[15] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[16] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858. ACM, 2017.

[17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)*, 2017.

[18] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *USENIX Security Symposium*, 2017.

[19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In *Engineering Secure Software and Systems*, pages 161–176, 2017.

[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[21] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. 2017.

[22] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *Proceedings of Crypto 2009*, volume 5677 of *LNCS*, pages 1–17. Springer-Verlag, August 2009.

[23] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer, 2001.

[24] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[25] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 629–636. IEEE, 2015.

[27] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. Investigation and improvement on the impact of tlb misses in real-time systems. *Proc. of OSPERT*, 2013.

[28] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, 2016.

[30] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[31] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204, 2012.

[32] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[33] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480, 2016.

[35] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[37] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[40] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[41] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. *NDSS, San Diego, CA, US*, 2017.

[42] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[44] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[45] Colin Percival. Cache missing for fun and profit, 2005.

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*.

[47] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[49] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[50] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199. IEEE, 2011.

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

[52] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds. *arXiv preprint arXiv:1708.09538*, 2017.

[53] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned aes t-tables. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.

[54] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[55] Venkatanathan Varadarajan and Yinqian Zhang. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Security Symposium*.

[56] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. *arXiv preprint arXiv:1705.07289*, 2017.

[57] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 367–382, 2016.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[59] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[60] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[61] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[62] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882. ACM, 2016.