Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX

Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen,
University of California, San Diego

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX

Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen
University of California, San Diego
{*cdisselk, dkohlbre*}*@cs.ucsd.edu, leporter@eng.ucsd.edu, tullsen@cs.ucsd.edu*

Abstract

Last-Level Cache (LLC) attacks typically exploit timing side channels in hardware, and thus rely heavily on timers for their operation. Many proposed defenses against such side-channel attacks capitalize on this reliance. This paper presents PRIME+ABORT, a new cache attack which bypasses these defenses by not depending on timers for its function. Instead of a timing side channel, PRIME+ABORT leverages the Intel TSX hardware widely available in both server- and consumer-grade processors. This work shows that PRIME+ABORT is not only invulnerable to important classes of defenses, it also outperforms state-of-the-art LLC PRIME+PROBE attacks in both accuracy and efficiency, having a maximum detection speed (in events per second) 3× higher than LLC PRIME+PROBE on Intel's Skylake architecture while producing fewer false positives.

1 Introduction

State-of-the-art cache attacks [35, 7, 11, 21, 25, 29, 33, 34, 43] leverage differences in memory access times between levels of the cache and memory hierarchy to gain insight into the activities of a victim process. These attacks require the attacker to frequently perform a series of timed memory operations (or cache management operations [7]) to learn if a victim process has accessed a critical address (e.g., a statement in an encryption library).

These attacks are highly dependent on precise and accurate timing, and defenses can exploit this dependence. In fact, a variety of defenses have been proposed which undermine these timing-based attacks by restricting access to highly precise timers [15, 27, 31, 39].

In this work, we introduce an alternate mechanism for performing cache attacks, which does not leverage timing differences (timing side channels) or require timed operations of any type. Instead, it exploits Intel's implementation of Hardware Transactional Memory, which is called TSX [19]. We demonstrate a novel cache attack based on this mechanism, which we will call PRIME+ABORT.

The intent of Transactional Memory (and TSX) is to both provide a simplified interface for synchronization and to enable optimistic concurrency: processes abort only when a conflict exists, rather than when a potential conflict may occur, as with traditional locks [14, 12]. Transactional memory operations require transactional data to be buffered, in this case in the cache which has limited space. Thus, the outcome of a transaction depends on the state of the cache, potentially revealing information to the thread that initiates the transaction. By exploiting TSX, an attacker can monitor the cache behavior of another process and receive an abort (call-back) if the victim process accesses a critical address. This work demonstrates how TSX can be used to trivially detect writes to a shared block in memory; to detect reads and writes by a process co-scheduled on the same core; and, most critically, to detect reads and writes by a process executing anywhere on the same processor. This latter attack works across cores, does not assume that the victim uses or even knows about TSX, and does not require any form of shared memory.

The advantages of this mechanism over conventional cache attacks are twofold. The first is that PRIME+ABORT does not leverage any kind of timer; as mentioned, several major classes of countermeasures against cache attacks revolve around either restricting access or adding noise to timers. PRIME+ABORT effectively bypasses these countermeasures.

The second advantage is in the efficiency of the attack. The TSX hardware allows for a victim's action to directly trigger the attacking process to take action. This means the TSX attack can bypass the detection phase required in conventional attacks. Direct coupling from event to handler allows PRIME+ABORT to provide over 3× the throughput of comparable state-of-the-art attacks.

The rest of this work is organized as follows. Section 2 presents background and related work; Section 3 introduces our novel attack, PRIME+ABORT; Section 4 describes experimental results, making comparisons with existing methods; in Section 5, we discuss potential countermeasures to our attack; Section 7 concludes.

2 Background and Related Work

2.1 Cache attacks

Cache attacks [35, 7, 11, 21, 25, 29, 33, 34, 43] are a well-known class of side-channel attacks which seek to gain information about which memory locations are accessed by some victim program, and at what times. In an excellent survey, Ge et al. [4] group such attacks into three broad categories: PRIME+PROBE, FLUSH+RELOAD, and EVICT+TIME. Since EVICT+TIME is only capable of monitoring memory accesses at the program granularity (whether a given memory location was accessed during execution or not), in this paper we focus on PRIME+PROBE and FLUSH+RELOAD, which are much higher resolution and have received more attention in the literature. Cache attacks have been shown to be effective for successfully recovering AES [25], ElGamal [29], and RSA [43] keys, performing keylogging [8], and spying on messages encrypted with TLS [23].

Figure 1 outlines all of the attacks which we will consider. At a high level, each attack consists of a pre-attack portion, in which important architecture- or runtime-specific information is gathered; and then an active portion which uses that information to monitor memory accesses of a victim process. The active portion of existing state-of-the-art attacks itself consists of three phases: an "initialization" phase, a "waiting" phase, and a "measurement" phase. The initialization phase prepares the cache in some way; the waiting phase gives the victim process an opportunity to access the target address; and then the measurement phase performs a timed operation to determine whether the cache state has changed in a way that implies an access to the target address has taken place.

Specifics of the initialization and measurement phases vary by cache attack (discussed below). Some cache attack implementations make a tradeoff in the length of the waiting phase between accuracy and resolution—shorter waiting phases give more precise information about the timing of victim memory accesses, but may increase the relative overhead of the initialization and measurement phases, which may make it more likely that a victim access could be "missed" by occurring outside of one of the measured intervals. In our testing, not all cache attack implementations and targets exhibited obvious experimental tradeoffs for the waiting phase duration. Nonetheless, fundamentally, all of these existing attacks can only gain temporal information at the waiting-interval granularity.

2.1.1 PRIME+PROBE

PRIME+PROBE [35, 21, 25, 34, 29] is the oldest and largest family of cache attacks, and also the most general. PRIME+PROBE does not rely on shared memory, unlike most other cache attacks (including FLUSH+RELOAD and its variants, described below). The original form of PRIME+PROBE [35, 34] targets the L1 cache, but recent work [21, 25, 29] extends it to target the L3 cache in Intel processors, enabling PRIME+PROBE to work across cores and without relying on hyperthreading (Simultaneous Multithreading [38]). Like all L3 cache attacks, L3 PRIME+PROBE can detect accesses to either instructions or data; in addition, L3 PRIME+PROBE trivially works across VMs.

PRIME+PROBE targets a single cache set, detecting accesses by any other program (or the operating system) to any address in that cache set. In its active portion's initialization phase (called "prime"), the attacker accesses enough cache lines from the cache set so as to completely fill the cache set with its own data. Later, in the measurement phase (called "probe"), the attacker reloads the same data it accessed previously, this time carefully observing how much time this operation took. If the victim did not access data in the targeted cache set, this operation will proceed quickly, finding its data in the cache. However, if the victim accessed data in the targeted cache set, the access will evict a portion of the attacker's primed data, causing the reload to be slower due to additional cache misses. Thus, a slow measurement phase implies the victim accessed data in the targeted cache set during the waiting phase. Note that this "probe" phase can also serve as the "prime" phase for the next repetition, if the monitoring is to continue.

Two different kinds of initial one-time setup are required for the pre-attack portion of this attack. The first is to establish a timing threshold above which the measurement phase is considered "slow" (i.e. likely suffering from extra cache misses). The second is to determine a set of addresses, called an "eviction set", which all map to the same (targeted) cache set (and which reside in distinct cache lines). Finding an eviction set is much easier for an attack targeting the L1 cache than for an attack targeting the L3 cache, due to the interaction between cache addressing and the virtual memory system, and also due to the "slicing" in Intel L3 caches (discussed further in Sections 2.2.1 and 2.2.2).

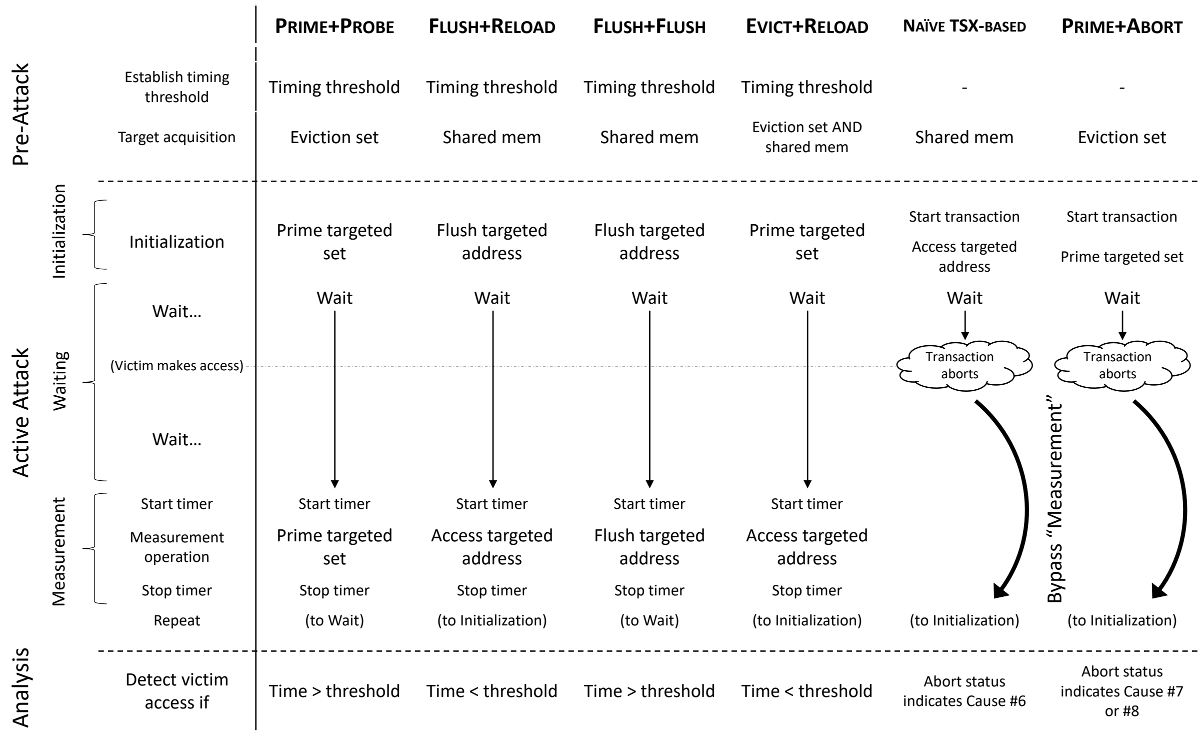

Figure 1: Comparison of the operation of various cache attacks, including our novel attacks.

2.1.2 FLUSH+RELOAD

The other major class of cache attacks is FLUSH+RELOAD [7, 11, 43]. FLUSH+RELOAD targets a specific address, detecting an access by any other program (or the operating system) to that exact address (or another address in the same cache line). This makes FLUSH+RELOAD a much more precise attack than PRIME+PROBE, which targets an entire cache set and is thus more prone to noise and false positives. FLUSH+RELOAD also naturally works across cores because of shared, inclusive, L3 caches (as explained in Section 2.2.1). Again, like all L3 cache attacks, FLUSH+RELOAD can detect accesses to either instructions or data. Additionally, FLUSH+RELOAD can work across VMs via the page deduplication exploit [43].

The pre-attack of FLUSH+RELOAD, like that of PRIME+PROBE, involves determining a timing threshold, but is limited to a single line instead of an entire "prime" phase. However, FLUSH+RELOAD does not require determining an eviction set. Instead, it requires the attacker to identify an exact target address; namely, an address in the attacker's virtual address space which maps to the physical address the attacker wants to monitor. Yarom and Falkner [43] present two ways to do this, both of which necessarily involve shared memory; one exploits shared libraries, and the other exploits page deduplication, which is how FLUSH+RELOAD can work across VMs. Nonetheless, this step's reliance on shared memory is a critical weakness in FLUSH+RELOAD, limiting it to only be able to monitor targets in shared memory.

In FLUSH+RELOAD's initialization phase, the attacker "flushes" the target address out of the cache using Intel's CLFLUSH instruction. Later, in the measurement phase, the attacker "reloads" the target address (by accessing it), carefully observing the time for the access. If the access was "fast", the attacker may conclude that another program accessed the address, causing it to be reloaded into the cache.

An improved variant of FLUSH+RELOAD, FLUSH+FLUSH [7], exploits timing variation in the CLFLUSH instruction itself; this enables the attack to combine its measurement and initialization phases, much like PRIME+PROBE. A different variant, EVICT+RELOAD [8], performs the initialization phase by evicting the cacheline with PRIME+PROBE's "prime" phase, allowing the attack to work without the CLFLUSH instruction at all—e.g., when the instruction has been disabled, as in Google Chrome's NaCl [6].

2.1.3 Timer-Free Cache Attacks

All of the attacks so far discussed—PRIME+PROBE, FLUSH+RELOAD, and variants—are still fundamentally timing attacks, exploiting timing differences as their underlying attack vector. One recent work which, like this work, proposes a cache attack without reference to timers is that of Guanciale et al. [10]. Instead of timing side channels, Guanciale et al. rely on the undocumented hardware behavior resulting from disobeying ISA programming guidelines, specifically with regards to virtual address aliasing and self-modifying code. However, they demonstrate their attacks only on the ARM architecture, and they themselves suggest that recent Intel x86-64 processors contain mechanisms that would render their attacks ineffective. In contrast, our attack exploits weaknesses specifically in recent Intel x86-64 processors, so in that respect our attack can be seen as complementary to Guanciale et al.'s work. We believe that our work, in addition to utilizing a novel attack vector (Intel's hardware transactional memory support), is the first timer-free cache attack to be demonstrated on commodity Intel processors.

2.2 Relevant Microarchitecture

2.2.1 Caches

[Basic Background] Caches in modern processors store data that is frequently or recently used, in order to reduce access time for that data on subsequent references. Data is stored in units of "cache lines" (a fixed architecture-dependent number of bytes). Caches are often organized hierarchically, with a small but fast "L1" cache, a medium-sized "L2" cache, and a large but comparatively slower "L3" cache. At each level of the hierarchy, there may either be a dedicated cache for each processor core, or a single cache shared by all processor cores.

Commonly, caches are "set-associative" which allows any given cacheline to reside in only one of N locations in the cache, where N is the "associativity" of the cache. This group of N locations is called a "cache set". Each cacheline is assigned to a unique cache set by means of its "set index", typically a subset of its address bits. Once a set is full (the common case) any access to a cacheline with the given set index (but not currently in the cache) will cause one of the existing N cachelines with the same set index to be removed, or "evicted", from the cache.

[Intel Cache Organization] Recent Intel processors contain per-core L1 instruction and data caches, per-core unified L2 caches, and a large L3 cache which is shared across cores. In this paper we focus on the Skylake architecture which was introduced in late 2015; important Skylake cache parameters are provided in Table 1.

Table 1: Relevant cache parameters in the Intel Skylake architecture.

	L1-Data	L1-Inst	L2	L3
Size	32 KB	32 KB	256 KB	2-8 MB[1]
Assoc	8-way	8-way	4-way	16-way
Sharing	Per-core	Per-core	Per-core	Shared
Line size	64 B	64 B	64 B	64 B

[1] depending on model. This range covers all Skylake processors (server, desktop, mobile, embedded) currently available as of January 2017 [20].

[Inclusive Caches] Critical to all cross-core cache attacks, the L3 cache is inclusive, meaning that everything in all the per-core caches must also be held in the L3. This has two important consequences which are key to enabling both L3-targeting PRIME+PROBE and FLUSH+RELOAD to work across cores. First, any data accessed by any core must be brought into not only the core's private L1 cache, but also the L3. If an attacker has "primed" a cache set in the L3, this access to a different address by another core necessarily evicts one of the attacker's cachelines, allowing PRIME+PROBE to detect the access. Second, any cacheline evicted from the L3 (e.g., in a "flush" step) must also be invalidated from all cores' private L1 and L2 caches. Any subsequent access to the cacheline by any core must fetch the data from main memory and bring it to the L3, causing FLUSH+RELOAD's subsequent "reload" phase to register a cache hit.

[Set Index Bits] The total number of cache sets in each cache can be calculated as (total number of cache lines) / (associativity), where the total number of cache lines is (cache size) / (line size). Thus, the Skylake L1 caches have 64 sets each, the L2 caches have 1024 sets each, and the shared L3 has from 2K to 8K sets, depending on the processor model.

In a typical cache, the lowest bits of the address (called the "line offset") determine the position within the cache line; the next-lowest bits of the address (called the "set index") determine in which cache set the line belongs, and the remaining higher bits make up the "tag". In our setting, the line offset is always 6 bits, while the set index will vary from 6 bits (L1) to 13 bits (L3) depending on the number of cache sets in the cache.

[Cache Slices and Selection Hash Functions] However, in recent Intel architectures (including Skylake), the situation is more complicated than this for the L3. Specifically, the L3 cache is split into several "slices" which can be accessed concurrently; the slices are connected on a ring bus such that each slice has a different latency depending on the core. In order to balance the load on these slices, Intel uses a proprietary and undocumented hash function, which operates on a physical address (except the line offset) to select which slice the address 'belongs' to. The output of this hash effectively serves as the top N bits of the set index, where 2^N is the number of slices in the system. Therefore, in the case of an 8 MB L3 cache with 8 slices, the set index consists of 10 bits from the physical address and 3 bits calculated using the hash function. For more details, see [25], [32], [44], [16], or [22].

This hash function has been reverse-engineered for many different processors in Intel's Sandy Bridge [25, 32, 44], Ivy Bridge [16, 22, 32], and Haswell [22, 32] architectures, but to our knowledge has not been reverse-engineered for Skylake yet. Not knowing the precise hash function adds additional difficulty to determining eviction sets for PRIME+PROBE—that is, finding sets of addresses which all map to the same L3 cache set. However, our attack (following the approach of Liu et al. [29]) does not require knowledge of the specific hash function, making it more general and more broadly applicable.

2.2.2 Virtual Memory

In a modern virtual memory system, each process has a set of *virtual addresses* which are mapped by the operating system and hardware to *physical addresses* at the granularity of pages [2]. The lowest bits of an address (referred to as the page offset) remain constant during address translation. Pages are typically 4 KB in size, but recently larger pages, for instance of size 2 MB, have become widely available for use at the option of the program [25, 29]. Crucially, an attacker may choose to use large pages regardless of whether the victim does or not [29].

Skylake caches are physically-indexed, meaning that the physical address of a cache line (and not its virtual address) determines the cache set which the line is mapped into. Like the slicing of the L3 cache, physical indexing adds additional difficulty to the problem of determining eviction sets for PRIME+PROBE, as it is not immediately clear which virtual addresses may have the same set index bits in their corresponding physical addresses. Pages make this problem more manageable, as the bottom 12 bits (for standard 4 KB pages) of the address remain constant during translation. For the L1 caches, these 12 bits contain the entire set index (6 bits of line offset + 6 bits of set index), so it is easy to choose addresses with the same set index. This makes the problem of determining eviction sets trivial for L1 attacks. However, L3 attacks must deal with both physical indexing and cache slicing when determining eviction sets. Using large pages helps, as the 21-bit large-page offset completely includes the set index bits (meaning they remain constant during translation), leaving only the problem of the hash function. However, the hash function is not only an unknown function itself, but it also incorporates bits from the entire physical address, including bits that are still translated even when using large pages.

Table 2: Availability of Intel TSX in recent Intel CPUs, based on data drawn from Intel ARK [20] in January 2017. Since Broadwell, all server CPUs and a majority of i7/i5 CPUs support TSX.

Series (Release[1])	Server[2]	i7/i5	i3/m/etc[3]
Kaby Lake (Jan 2017)	3/3 (100%)	23/32 (72%)	12/24 (50%)
Skylake (Aug 2015)	23/23 (100%)	27/42 (64%)	4/34 (12%)
Broadwell (Sep 2014)	77/77 (100%)	11/22 (50%)	2/18 (11%)
Haswell (Jun 2013)	37/85 (44%)	2/87 (2%)	0/82 (0%)

[1] for the earliest available processors in the series
[2] Xeon and Pentium-D
[3] (i3/m/Pentium/Celeron)

2.3 Transactional Memory and TSX

Transactional Memory (TM) has received significant attention from the computer architecture and systems community over the past two decades [14, 13, 37, 45]. First proposed by Herlihy and Moss in 1993 as a hardware alternative to locks [14], TM is noteworthy for its simplification of synchronization primitives and for its ability to provide optimistic concurrency.

Unlike traditional locks which require threads to wait if a conflict is possible, TM allows multiple threads to proceed in parallel and only abort in the event of a conflict [36]. To detect a conflict, TM tracks each thread's read and write sets and signals an abort when a conflict is found. This tracking can be performed either by special hardware [14, 13, 45] or software [37].

Intel's TSX instruction set extension for x86 [12, 19] provides an implementation of hardware TM and is widely available in recent Intel CPUs (see Table 2).

TSX allows any program to identify an arbitrary section of its code as a 'transaction' using explicit `XBEGIN` and `XEND` instructions. Any transaction is guaranteed to either: (1) **complete**, in which case all memory changes which happened during the transaction are made visible *atomically* to other processes and cores, or (2) **abort**, in which case all memory changes which happened during the transaction, as well as all other changes (e.g. to registers), are discarded. In the event of an abort, control is transferred to a fallback routine specified by the user, and a status code provides the fallback routine with some information about the cause of the abort.

From a security perspective, the intended uses of hardware transactional memory (easier synchronization

Table 3: Causes of transactional aborts in Intel TSX

1.	Executing certain instructions, such as CPUID or the explicit XABORT instruction
2.	Executing system calls
3.	OS interrupts[1]
4.	Nesting transactions too deeply
5.	Access violations and page faults
6.	Read-Write or Write-Write memory conflicts with other threads or processes (including other cores) at the cacheline granularity—whether those other processes are using TSX or not
7.	A cacheline which has been written during the transaction (i.e., a cacheline in the transaction's "write set") is evicted from the L1 cache
8.	A cacheline which has been read during the transaction (i.e., a cacheline in the transaction's "read set") is evicted from the L3 cache

[1] This means that any transaction may abort, despite the absence of memory conflicts, through no fault of the programmer. The periodic nature of certain interrupts also sets an effective maximum time limit on any transaction, which has been measured at about 4 ms [41].

or optimistic concurrency) are unimportant, so we will merely note that we can place arbitrary code inside both the transaction and the fallback routine, and whenever the transaction aborts, our fallback routine will immediately be given a callback with a status code. There are many reasons a TSX transaction may abort; important causes are listed in Table 3. Most of these are drawn from the Intel Software Developer's Manual [19], but the specifics of Causes #7 and #8—in particular the asymmetric behavior of TSX with respect to read sets and write sets—were suggested by Dice et al. [3]. Our experimental results corroborate their suggestions about these undocumented implementation details.

While a transaction is in process, an arbitrary amount of data must be buffered (hidden from the memory system) or tracked until the transaction completes or aborts. In TSX, this is done in the caches—transactionally written lines are buffered in the L1 data cache, and transactionally read lines marked in the L1–L3 caches. This has the important ramification that the cache size and associativity impose a limit on how much data can be buffered or tracked. In particular, if cache lines being buffered or tracked by TSX must be evicted from the cache, this necessarily causes a transactional abort. In this way, details about cache activity may be exposed through the use of transactions.

TSX has been addressed only rarely in a security context; to the best of our knowledge, there are only two works on the application of TSX to security to date [9, 24]. Guan et al. use TSX as part of a defense against memory disclosure attacks [9]. In their system, operations involving the plaintext of sensitive data necessarily occur inside TSX transactions. This structurally ensures that this plaintext will never be accessed by other processes or written back to main memory (in either case, a transactional abort will roll back the architectural state and invalidate the plaintext data).

Jang et al. exploit a timing side channel in TSX itself in order to break kernel address space layout randomization (KASLR) [24]. Specifically, they focus on Abort Cause #5, access violations and page faults. They note that such events inside a transaction trigger an abort but not their normal respective handlers; this means the operating system or kernel are *not* notified, so the attack is free to trigger as many access violations and page faults as it wants without raising suspicions. They then exploit this property and the aforementioned timing side channel to determine which kernel pages are mapped and unmapped (and also which are executable).

Neither of these works enable new attacks on memory accesses, nor do they eliminate the need for timers in attacks.

3 Potential TSX-based Attacks

We present three potential attacks, all of which share their main goal with cache attacks—to monitor which cachelines are accessed by other processes and when. The three attacks we will present leverage Abort Causes #6, 7, and 8 respectively. Figure 1 outlines all three of the attacks we will present, as the PRIME+ABORT entry in the figure applies to both PRIME+ABORT–L1 and PRIME+ABORT–L3.

All of the TSX-based attacks which we will propose have the same critical structural benefit in common. This benefit, illustrated in Figure 1, is that these attacks have no need for a "measurement" phase. Rather than having to conduct some (timed) operation to determine whether the cache state has been modified by the victim, they simply receive a hardware callback through TSX immediately when a victim access takes place. In addition to the reduced overhead this represents for the attack procedure, this also means the attacker can be actively waiting almost indefinitely until the moment a victim access occurs—the attacker does not need to break the attack into predefined intervals. This results in a higher resolution attack, because instead of only coarse-grained knowledge of when a victim access occurred (i.e. which predefined interval), the attacker gains precise estimates of the relative timing of victim accesses.

All of our proposed TSX-based attacks also share a structural weakness when compared to PRIME+PROBE and FLUSH+RELOAD. Namely, they are unable to monitor multiple targets (cache sets in the case of PRIME+PROBE, addresses in the case of FLUSH+RELOAD) simultaneously while retaining the ability to distinguish accesses to one target from accesses to another. PRIME+PROBE and FLUSH+RELOAD are able to do this at the cost of increased overhead; effectively, a process can monitor multiple targets concurrently by performing multiple initialization stages, having a common waiting stage, and then performing multiple measurement stages, with each measurement stage revealing the activity for the corresponding target. In contrast, although our TSX-based attacks could monitor multiple targets at once, they would be unable to distinguish events for one target from events for another without additional outside information. Some applications of PRIME+PROBE and FLUSH+RELOAD rely on this ability (e.g. [33]), and adapting them to rely on PRIME+ABORT instead would not be trivial. However, others, including the attack presented in Section 4.4, can be straightforwardly adapted to utilize PRIME+ABORT as a drop-in replacement for PRIME+PROBE or FLUSH+RELOAD.

We begin by discussing the simplest, but also least generalizable, of our TSX-based attacks, ultimately building to our proposed primary attack, PRIME+ABORT–L3.

3.1 Naïve TSX-based Attack

Abort Cause #6 enables a potentially powerful, but limited attack.

From Cause #6, we can get a transaction abort (which for our purposes is an immediate, fast hardware callback) whenever there is a read-write or write-write conflict between our transaction and another process. This leads to a natural and simple attack implementation, where we simply open a transaction, access our target address, and wait for an abort (with the proper abort status code); on abort, we know the address was accessed by another process.

The style of this attack is reminiscent of FLUSH+RELOAD [43] in several ways. It targets a single, precise cacheline, rather than an entire cache set as in PRIME+PROBE and its variants. It does not require a (comparatively slow) "prime eviction set" step, providing fast and low-overhead monitoring of the target cacheline. Also like FLUSH+RELOAD, it requires the attacker to acquire a specific address to target, for instance exploiting shared libraries or page deduplication.

Like the other attacks using TSX, it benefits in performance by not needing the "measurement" phase to detect a victim access. In addition to the performance benefit, this attack would also be harder to detect and defend against. It would execute without any kind of timer, mitigating several important classes of defenses (see Section 5). It would also be resistant to most types of cache-based defenses; in fact, this attack has so little to do with the cache at all that it could hardly be called a cache attack, except that it happens to expose the same information as standard cache attacks such as FLUSH+RELOAD or PRIME+PROBE do.

However, in addition to only being able to monitor target addresses in shared memory (the key weakness shared by all variants of FLUSH+RELOAD), this attack has another critical shortcoming. Namely, it can only detect read-write or write-write conflicts, not read-read conflicts. This means that one or the other of the processes—either the attacker or the victim—must be issuing a write command in order for the access to be detected, i.e. cause a transactional abort. Therefore, the address being monitored must not be in read-only memory. Combining this with the earlier restriction, we find that this attack, although powerful, can only monitor addresses in writable shared memory. We find this dependence to render it impractical for most real applications, and for the rest of the paper we focus on the other two attacks we will present.

3.2 PRIME+ABORT–L1

The second attack we will present, called PRIME+ABORT–L1, is based on Abort Cause #7. Abort Cause #7 provides us with a way to monitor evictions from the L1 cache in a way that is precise and presents us with, effectively, an immediate hardware callback in the form of a transactional abort. This allows us to build an attack in the PRIME+PROBE family, as the key component of PRIME+PROBE involves detecting cacheline evictions. This attack, like all attacks in the PRIME+PROBE family, does not depend in any way on shared memory; but unlike other attacks, it will also not depend on timers.

Like other PRIME+PROBE variants, our attack requires a one-time setup phase where we determine an eviction set for the cache set we wish to target; but like early PRIME+PROBE attacks [35, 34], we find this task trivial because the entire L1 cache index lies within the page offset (as explained earlier). Unlike other PRIME+PROBE variants, for PRIME+ABORT this is the sole component of the setup phase; we do not need to find a timing threshold, as we do not rely on timing.

The main part of PRIME+ABORT–L1 involves the same "prime" phase as a typical PRIME+PROBE attack, except that it opens a TSX transaction first. Once the "prime" phase is completed, the attack simply waits for an abort (with the proper abort status code). Upon receiving an abort, the attacker can conclude that some other program has accessed an address in the target cache set. This is similar to the information gleaned by ordinary PRIME+PROBE.

The reason this works is that, since we will hold an entire cache set in the write set of our transaction, any access to a different cache line in that set by another process will necessarily evict one of our cachelines and cause our transaction to abort due to Cause #7. This gives us an immediate hardware callback, obviating the need for any "measurement" step as in traditional cache attacks. This is why we call our method PRIME+ABORT—the abort replaces the "probe" step of traditional PRIME+PROBE.

3.3 PRIME+ABORT–L3

PRIME+ABORT–L1 is fast and powerful, but because it targets the (core-private) L1 cache, it can only spy on threads which share its core; and since it must execute simultaneously with its victim, this means it and its victim must be in separate hyperthreads on the same core. In this section we present PRIME+ABORT–L3, an attack which overcomes these restrictions by targeting the L3 cache. The development of PRIME+ABORT–L3 from PRIME+ABORT–L1 mirrors the development of L3-targeting PRIME+PROBE [29, 21, 25] from L1-targeting PRIME+PROBE [35, 34], except that we use TSX. PRIME+ABORT–L3 retains all of the TSX-provided advantages of PRIME+ABORT–L1, while also (like L3 PRIME+PROBE) working across cores, easily detecting accesses to either instructions or data, and even working across virtual machines.

PRIME+ABORT–L3 uses Abort Cause #8 to monitor evictions from the L3 cache. The only meaningful change this entails to the active portion of the attack is performing reads rather than writes during the "prime" phase, in order to hold the primed cachelines in the read set of the transaction rather than the write set. For the pre-attack portion, PRIME+ABORT–L3, like other L3 PRIME+PROBE attacks, requires a much more sophisticated setup phase in which it determines eviction sets for the L3 cache. This is described in detail in the next section.

3.4 Finding eviction sets

The goal of the pre-attack phase for PRIME+ABORT is to determine an eviction set for a specified target address. For PRIME+ABORT–L1, this is straightforward, as described in Section 2.2.2. However, for PRIME+ABORT–L3, we must deal with both physical indexing and cache slicing in order to find L3 eviction sets. Like [29] and [21], we use large (2 MB) pages in this process as a convenience. With large pages, it becomes trivial to choose virtual addresses that have the same physical set index (i.e. agree in bits 6 to N, for some processor-dependent N, perhaps 15), again as explained in Section 2.2.2. We will refer to addresses which agree in physical set index (and in line offset, i.e. bits 0 to 5) as *set-aligned* addresses.

Algorithm 1: Dynamically generating a prototype eviction set for each cache slice, as implemented in [42]

```
Input: a set of potentially conflicting cachelines lines, all
       set-aligned
Output: a set of prototype eviction sets, one eviction set for each
        cache slice; that is, a "prototype group"
group ← {};
workingSet ← {};
while lines is not empty do
    repeat forever :
        line ← random member of lines;
        remove line from lines;
        if workingSet evicts line then // Algorithm 2 or 3
            c ← line;
            break;
        end
        add line to workingSet;
    end
    foreach member in workingSet do
        remove member from workingSet;
        if workingSet evicts c then    // Algorithm 2 or 3
            add member back to lines;
        else
            add member back to workingSet;
        end
    end
    foreach line in lines do
        if workingSet evicts line then // Algorithm 2 or 3
            remove line from lines;
        end
    end
    add workingSet to group;
    workingSet ← {};
end
return group;
```

We generate eviction sets dynamically using the algorithm from Mastik [42] (inspired by that in [29]), which is shown as Algorithm 1. However, for the subroutine where Mastik uses timing methods to evaluate potential eviction sets (Algorithm 2), we use TSX methods instead (Algorithm 3).

Algorithm 3, a subroutine of Algorithm 1, demonstrates how Intel TSX is used to determine whether a candidate eviction set can be expected to consistently evict a given target cacheline. If "priming" the eviction set (accessing all its lines) inside a transaction followed by accessing the target cacheline consistently results in an immediate abort, we can conclude that a transaction cannot hold both the eviction set and the target cacheline in its read set at once, which means that together they contain at least ($associativity + 1$, or 17 in our case) lines which map to the same cache slice and cache set.

Conceptually, the algorithm for dynamically generating an eviction set for any given address has two phases: first, creating a "prototype group", and second, specializing it to form an eviction set for the desired target ad-

Algorithm 2: PRIME+PROBE (timing-based) method for determining whether an eviction set evicts a given cacheline, as implemented in [42]

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line
times ← {};
repeat 16 times :
    access line;
    repeat 20 times :
        foreach member in es do
            access member;
        end
    end
    timed access to line;
    times ← times + {elapsed time};
end
if median of times > predetermined threshold then return true;
else return false;
```

Algorithm 3: PRIME+ABORT (TSX-based) method for determining whether an eviction set evicts a given cacheline

```
Input: a candidate eviction set es and a cacheline line
Output: true if es can be expected to consistently evict line
aborts ← 0;
commits ← 0;
while aborts < 16 and commits < 16 do
    begin transaction;
    foreach member in es do
        access member;
    end
    access line;
    end transaction;
    if transaction committed then increment commits;
    else if transaction aborted with appropriate status code then
      increment aborts;
end
if aborts >= 16 then return true;
else return false;
```

dress. The algorithms shown (Algorithms 1, 2, and 3) together constitute the first phase of this larger algorithm. In this first phase, we use only set-aligned addresses, noting that all such addresses, after being mapped to an L3 cache slice, necessarily map to the same cache set inside that slice. This phase creates one eviction set for each cache slice, targeting the cache set inside that slice with the given set index. We call these "prototype" eviction sets, and we call the resulting group of one "prototype" eviction set per cache slice a "prototype group".

Once we have a prototype group generated by Algorithm 1, we can obtain an eviction set for any cache set in any cache slice by simply adjusting the set index of each address in one of the prototype eviction sets. Not knowing the specific cache-slice-selection hash function, it will be necessary to iterate over all prototype eviction sets (one per slice) in order to find the one which collides with the target on the same cache slice. If we do not know the (physical) set index of our target, we can also iterate through all possible set indices (with each prototype eviction set) to find the appropriate eviction set, again following the procedure from Liu et al. [29].

4 Results

4.1 Characteristics of the Intel Skylake Architecture

Our test machine has an Intel Skylake i7-6600U processor, which has two physical cores and four virtual cores. It is widely reported (e.g., in all of [16, 22, 25, 29, 32, 44]) that Intel processors have one cache slice per physical core, based on experiments conducted on Sandy Bridge, Ivy Bridge, and Haswell processors. However, our testing on the Skylake dual-core i7-6600U leads us to believe that it has four cache slices, contrary to previous trends which would predict it has only two. We validate this claim by using Algorithm 1 to produce four distinct eviction sets for large-page-aligned addresses. Then we test our four distinct eviction sets on many additional large-page-aligned addresses not used in Algorithm 1. We find that each large-page-aligned address conflicts with exactly one of the four eviction sets (by Algorithm 3), and further, that the conflicts are spread relatively evenly over the four sets. This convinces us that each of our four eviction sets represents set index 0 on a different cache slice, and thus that there are indeed four cache slices in the i7-6600U.

Having determined the number of cache slices, we can now calculate the number of low-order bits in an address that must be fixed to create groups of set-aligned addresses. For our i7-6600U, this is 16. Henceforth we can use set-aligned addresses instead of large-page-aligned addresses, which is an efficiency gain.

4.2 Dynamically Generating Eviction Sets

In the remainder of the Results section we compare PRIME+ABORT–L3 to L3 PRIME+PROBE as implemented in [42]. We begin by comparing the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1 for dynamically generating prototype eviction sets.

Table 4 compares the runtimes of the PRIME+ABORT and PRIME+PROBE versions of Algorithm 1. The PRIME+ABORT-based method is over 5× faster than the PRIME+PROBE-based method in the median case, over 15× faster in the best case, and over 40% faster in the worst case.

Next, we compare the "coverage" of prototype groups (sets of four prototype eviction sets) derived and tested

Table 4: Runtimes of PRIME+ABORT- and PRIME+PROBE-based versions of Algorithm 1 to generate a "prototype group" of eviction sets (data based on 1000 runs of each version of Algorithm 1)

	PRIME+ABORT	PRIME+PROBE
Min	4.5 ms	68.3 ms
1Q	10.1 ms	76.6 ms
Median	15.0 ms	79.3 ms
3Q	21.3 ms	82.0 ms
Max	64.7 ms	91.0 ms

with the two methods. We derive 10 prototype groups with each version of Algorithm 1; then, for each prototype group, we use either timing-based or TSX-based methods to test 1000 additional set-aligned addresses not used for Algorithm 1 (a total of 10,000 additional set-aligned addresses for PRIME+ABORT and 10,000 for PRIME+PROBE). The testing procedure is akin to a single iteration of the outer loop in Algorithm 2 or 3 respectively. Using this procedure, each of the 10,000 set-aligned addresses is tested 10,000 times against each of the four prototype eviction sets in the prototype group. This produces four "detection rates" for each set-aligned address (one per prototype eviction set). We assume that the highest of these four detection rates corresponds to the prototype eviction set from the same cache slice as the tested address, and we call this detection rate the "max detection rate" for the set-aligned address. Both PRIME+ABORT and PRIME+PROBE methods result in "max detection rates" which are consistently indistinguishable from 100%. However, we note that out of the 100 million trials in total, 13 times we observed the PRIME+PROBE-based method fail to detect the access (resulting in a "max detection rate" of 99.99% in 13 cases), whereas with the PRIME+ABORT-based method, all 100 million trials were detected, for perfect max detection rates of 100.0%. This result is due to the structural nature of transactional conflicts—it is impossible for a transaction with a read set of size $(1 + \textit{associativity})$ to ever successfully commit; it must always abort.

Since each address maps to exactly one cache slice, and ideally each eviction set contains lines from only one cache slice, we expect that any given set-aligned address conflicts with only one out of the four prototype eviction sets in a prototype group. That is, we expect that out of the four detection rates computed for each line (one per prototype eviction set), one will be very high (the "max detection rate"), and the other three will be very low. Figure 2 shows the "second-highest detection rate" for each line—that is, the maximum of the remaining three detection rates for that line, which is a measure of false positives. For any given detection rate on the x-axis, the figure shows what percentage of the 10,000 set-aligned addresses had a false-positive detection rate at or above that level. Whenever the "second-highest detection rate" is greater than zero, it indicates that the line appeared to be detected by a prototype eviction set meant for an entirely different cache slice (i.e. a false positive detection). In Figure 2, we see that with the PRIME+PROBE-based method, around 22% of lines have "second-highest detection rates" over 5%, around 18% of lines have "second-highest detec-

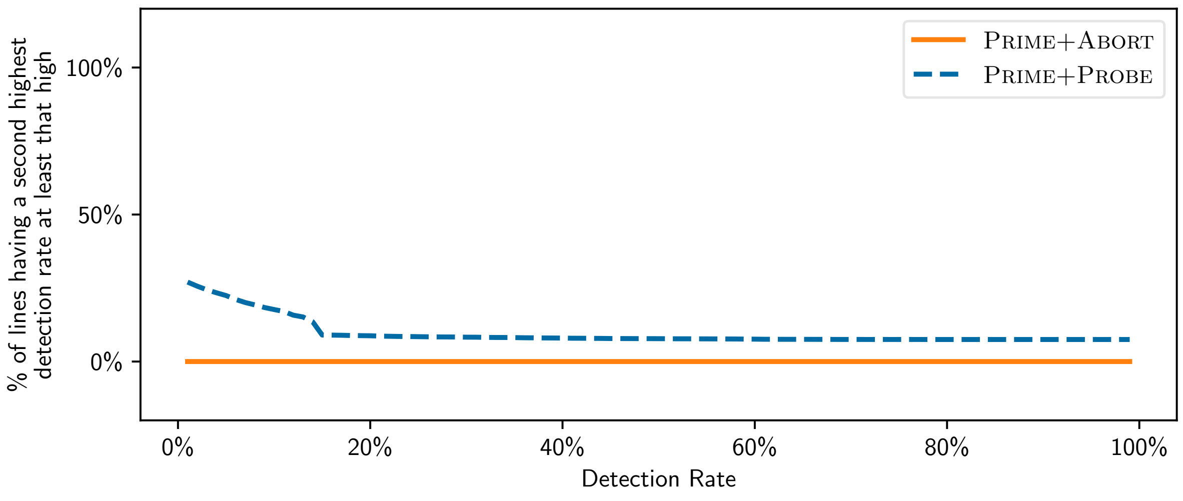

Figure 2: "Double coverage" of prototype groups generated by PRIME+ABORT- and PRIME+PROBE-based versions of Algorithm 1. With PRIME+PROBE, some tested cachelines are reliably detected by more than one prototype eviction set. In contrast, with PRIME+ABORT each tested cacheline is reliably detected by only one prototype eviction set.

tion rates" over 10%, and around 7.5% of lines even have "second-highest detection rates" of 100%, meaning that more than one of the "prototype eviction sets" each detected that line in 100% of the $10,000$ trials. In contrast, with the PRIME+ABORT-based method, none of the $10,000$ lines tested had "second-highest detection rates" over 1%. PRIME+ABORT produces very few false positives and cleanly monitors exactly one cache set in exactly one cache slice.

4.3 Detecting Memory Accesses

Figures 3, 4, and 5 show the success of PRIME+ABORT and two variants of PRIME+PROBE in detecting the memory accesses of an artificial victim thread running on a different physical core from the attacker. The victim thread repeatedly accesses a single memory location for the duration of the experiment—in the "treatment" condition, it accesses the target (monitored) location, whereas in the "control" condition, it accesses an unrelated location. We introduce delays (via busy-wait) of varying lengths into the victim's code in order to vary the frequency at which it accesses the target location (or unrelated location for control). Figures 3, 4, and 5 plot the number of events observed by the respective attackers, vs. the actual number of accesses by the victim, in "control" and "treatment" scenarios. Data were collected from 100 trials per attacker, each entailing separate runs of Algorithm 1 and new targets. The $y = x$ line is shown for reference in all figures; it indicates perfect performance for the "treatment" condition, with all events detected but no false positives. Perfect performance in the "control" condition, naturally, is values as low as possible in all cases.

We see in Figure 3 that PRIME+ABORT detects a large fraction of the victim's accesses at frequencies up to several hundred thousand accesses per second, scaling up smoothly and topping out at a maximum detection speed (on our test machine) of around one million events per second. PRIME+ABORT exhibits this performance while also displaying relatively low false positive rates of around 200 events per second, or one false positive every 5000 μs. The close correlation between number of detected events and number of victim accesses indicates PRIME+ABORT's low overheads—in fact, we measured its transactional abort handler as executing in 20-40 ns—which allow it to be essentially "always listening" for victim accesses. Also, it demonstrates PRIME+ABORT's ability to accurately count the number of victim accesses, despite only producing a binary output (access or no access) in each transaction. Its high speed and low overheads allow it to catch each victim access in a separate transaction.

Figure 4 shows the performance of unmodified PRIME+PROBE as implemented in Mastik [42][1]. We see false positive rates which are significantly higher than those observed for PRIME+ABORT—over 2000 events per second, or one every 500 μs. Like PRIME+ABORT, this implementation of PRIME+PROBE appears to have a top speed around one million accesses detected per second under our test conditions. But most interestingly, we observe significant "oversampling" at low frequencies—PRIME+PROBE reports many more events than actually occurred. For instance, when the victim thread performs 2600 accesses per second, we expect to observe 2600 events per second, plus around 2000 false positives per second as before. However, we actually observe over $18,000$ events per second in the median case. Likewise, when the victim thread provides $26,000$ accesses per second, we observe over $200,000$ events per second in the median case. Analysis shows that for this implementation of PRIME+PROBE on our hardware, single accesses can cause long streaks of consecutive observed events, sometimes as long as hundreds of observed events. We believe this to be due to the interaction between this PRIME+PROBE implementation and our hardware's L3 cache replacement policy. One plausible explanation for why PRIME+ABORT is not similarly afflicted, is that the replacement policy may prioritize keeping lines that are part of active transactions, evicting everything else first. This would be a sensible policy for Intel to implement, as it would minimize the number of unwanted/unnecessary aborts. In our setting, it benefits PRIME+ABORT by ensuring that a "prime" step inside a transaction cleanly evicts all other lines.

To combat the oversampling behavior observed in PRIME+PROBE, we investigate a modified implementation of PRIME+PROBE which "collapses" streaks of observed events, meaning that a streak of any length is simply counted as a single observed event. Results with this modified implementation are shown in Figure 5. We see that this strategy is effective in combating oversampling, and also reduces the number of false positives to around 250 per second or one every 4000 μs. However, this implementation of PRIME+PROBE performs more poorly at high frequencies, having a top speed around $300,000$ events per second compared to the one million per second of the other two attacks. This effect can be explained by the fact that as the victim access frequency increases, streaks of observed events become more and more likely to "hide" real events (multiple real events occur in the same streak)—in the limit, we expect to observe an event

[1]We make one slight modification suggested by the maintainer of Mastik: every probe step, we actually perform multiple probes, "counting" only the first one. In our case we perform five probes at a time, still alternating between forwards and backwards probes. All of the results which we present for the "unmodified" implementation include this slight modification.

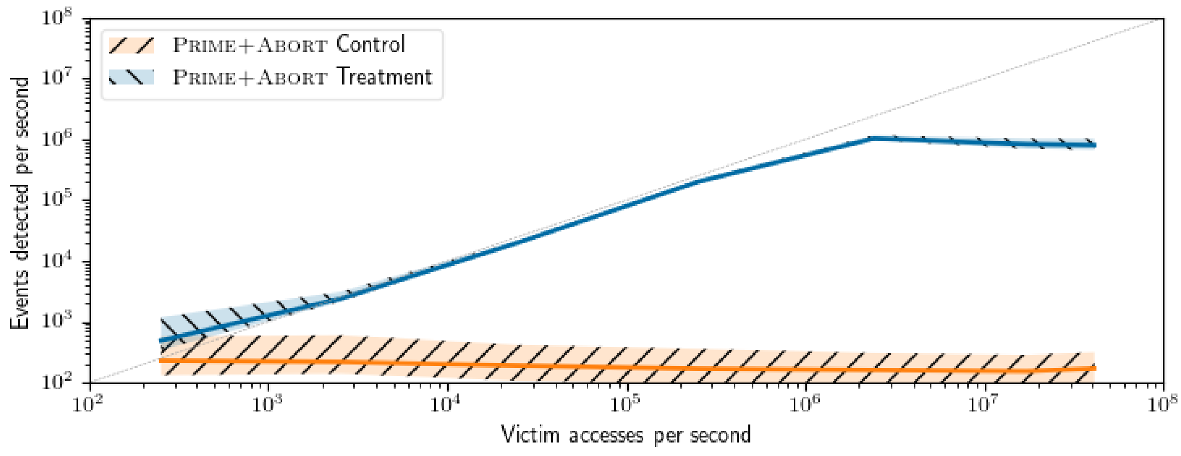

Figure 3: Access detection rates for PRIME+ABORT in the "control" and "treatment" conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

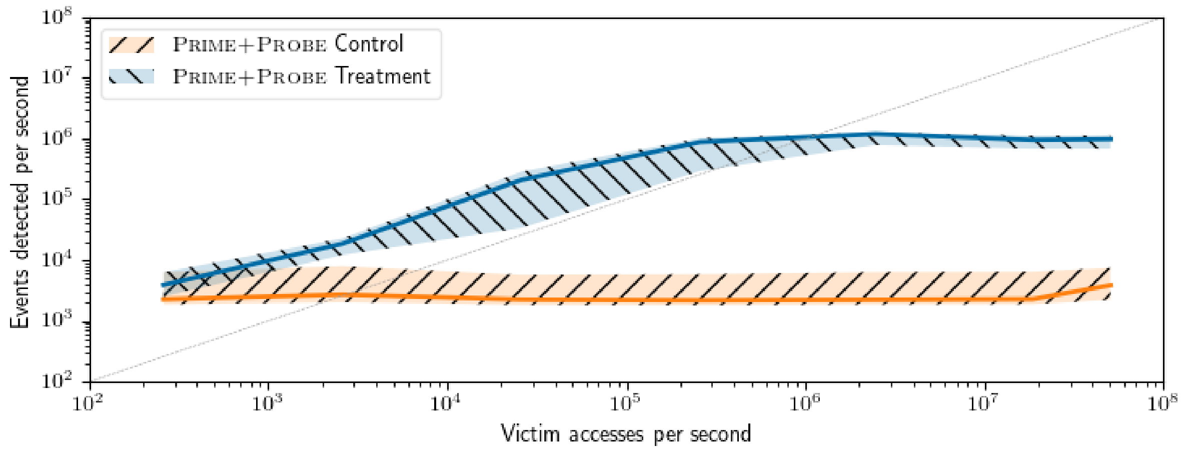

Figure 4: Access detection rates for unmodified PRIME+PROBE in the "control" and "treatment" conditions. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

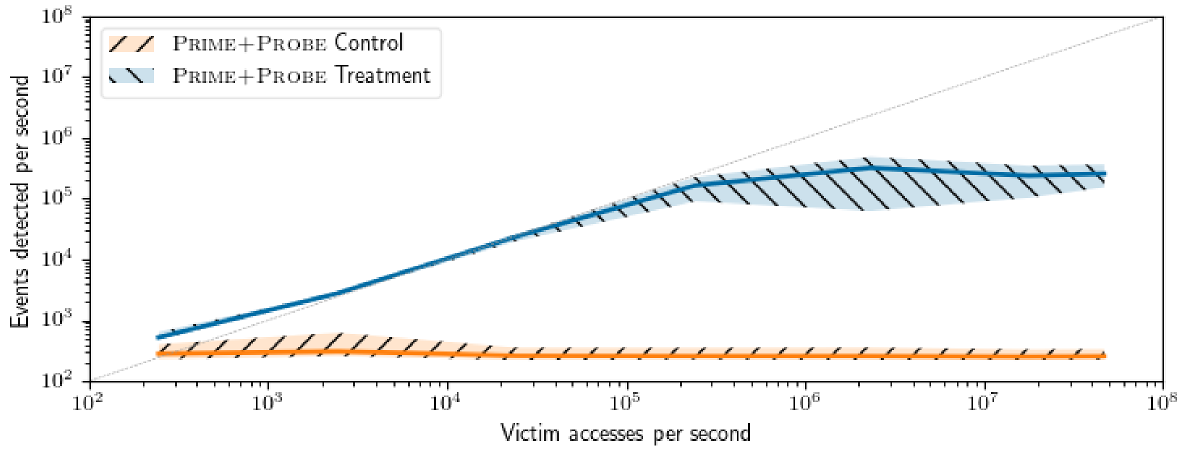

Figure 5: Access detection rates for our modified implementation of PRIME+PROBE which "collapses" streaks. Data were collected over 100 trials, each involving several different victim access speeds. Shaded regions indicate the range of the middle 75% of the data; lines indicate the medians. The $y = x$ line is added for reference and indicates perfect performance for the "treatment" condition (all events detected but no false positives or oversampling).

during every probe, but this approach will observe only a single streak and indicate a single event occurred.

Observing the two competing implementations of PRIME+PROBE on our hardware reveals an interesting tradeoff. The original implementation has good high frequency performance, but suffers from both oversampling and a high number of false positives. In contrast, the modified implementation has poor high frequency performance, but does not suffer from oversampling and exhibits fewer false positives. For the remainder of this paper we consider the modified implementation of PRIME+PROBE only, as we expect that its improved accuracy and fewer false positives will make it more desirable for most applications. Finally, we note that PRIME+ABORT combines the desirable characteristics of both PRIME+PROBE implementations, as it exhibits the fewest false positives, does not suffer from oversampling, and has good high frequency performance, with a top speed around one million events per second.

4.4 Attacks on AES

In this section we evaluate the performance of PRIME+ABORT in an actual attack by replicating the attack on OpenSSL's T-table implementation of AES, as conducted by Gruss et al. [7]. As those authors acknowledge, this implementation is no longer enabled by default due to its susceptibility to these kinds of attacks. However, as with their work, we use it for the purpose of comparing the speed and accuracy of competing attacks. Gruss et al. compared PRIME+PROBE, FLUSH+RELOAD, and FLUSH+FLUSH [7]; we have chosen to compare PRIME+PROBE and PRIME+ABORT, as these attacks do not rely on shared memory. Following their methods, rather than using previously published results directly, we rerun previous attacks alongside ours to ensure fairness, including the same hardware setup.

Figures 6 and 7 provide the results of this experiment. In this chosen-plaintext attack, we listen for accesses to the first cacheline of the first T-Table (`Te0`) while running encryptions. We expect that when the first four bits of our plaintext match the first four bits of the key, the algorithm will access this cacheline one extra time over the course of each encryption compared to when the bits do not match. This will manifest as causing more events to be detected by PRIME+ABORT or PRIME+PROBE respectively, allowing the attacker to predict the four key bits. The attack can then be continued for each byte of plaintext (monitoring a different cacheline of `Te0` in each case) to reveal the top four bits of each key byte.

In our experiments, we used a key whose first four bits were arbitrarily chosen to be `1110`, and for each method we performed one million encryptions with each possible 4-bit plaintext prefix (a total of sixteen million encryptions for PRIME+ABORT and sixteen million for PRIME+PROBE). As shown in Figures 6 and 7, both methods correctly predict the first four key bits to be `1110`, although the signal is arguably cleaner and stronger when using PRIME+ABORT.

5 Potential Countermeasures

Many countermeasures against side-channel attacks have already been proposed; Ge et al. [4] again provide an

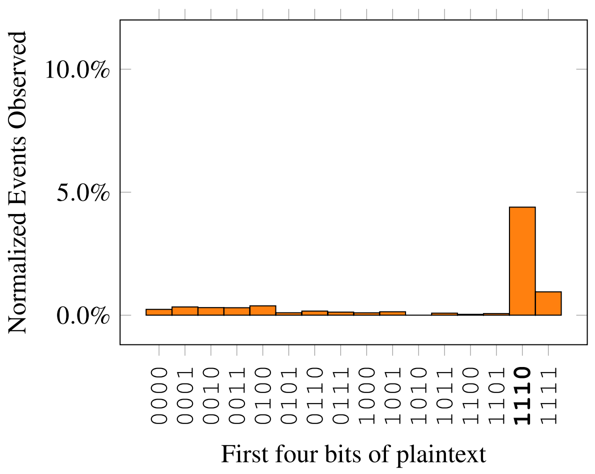

Figure 6: PRIME+ABORT attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

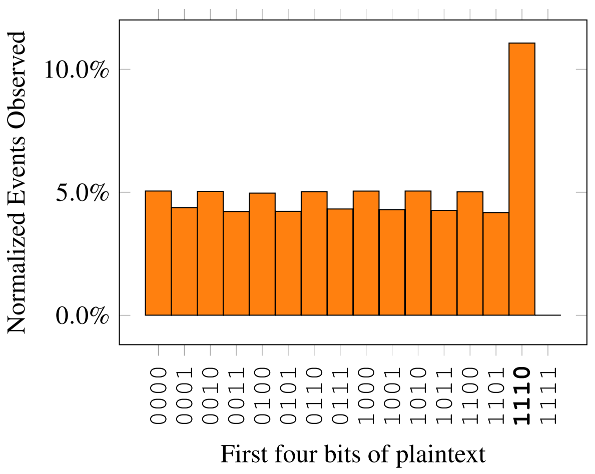

Figure 7: PRIME+PROBE attack against AES. Shown is, for each condition, the percentage of additional events that were observed compared to the condition yielding the fewest events.

excellent survey. Examining various proposed defenses in the context of PRIME+ABORT reveals that some are effective against a wide variety of attacks including PRIME+ABORT, whereas others are impractical or ineffective against PRIME+ABORT. This leads us to advocate for the prioritization and further development of certain approaches over others.

We first examine classes of side-channel countermeasures that are impractical or ineffective against PRIME+ABORT and then move toward countermeasures which are more effective and practical.

Timer-Based Countermeasures: A broad class of countermeasures ineffective against PRIME+ABORT are approaches that seek to limit the availability of precise timers, either by injecting noise into timers to make them less precise, or by restricting access to timers in general. There are a wide variety of proposals in this vein, including [15], [27], [31], [39], and various approaches which Ge et al. classify as “Virtual Time” or “Black-Box Mitigation”. PRIME+ABORT should be completely immune to all timing-related countermeasures.

Partitioning Time: Another class of countermeasures that seems impractical against PRIME+ABORT is the class Ge et al. refer to as Partitioning Time. These countermeasures propose some form of “time-sliced exclusive access” to shared hardware resources. This would technically be effective against PRIME+ABORT, because the attack is entirely dependent on running simultaneously with its victim process; any context switch causes a transactional abort, so the PRIME+ABORT process must be active in order to glean any information. However, since PRIME+ABORT targets the LLC and can monitor across cores, implementing this countermeasure against PRIME+ABORT would require providing each user process time-sliced exclusive access to the LLC. This would mean that processes from different users could never run simultaneously, even on different cores, which seems impractical.

Disabling TSX: A countermeasure which would ostensibly target PRIME+ABORT’s workings in particular would be to disable TSX entirely, similarly to how hyperthreading has been disabled entirely in cloud environments such as Microsoft Azure [30]. While this is technically feasible—in fact, due to a hardware bug, Intel already disabled TSX in many Haswell CPUs through a microcode update [17]—TSX’s growing prevalence (Table 2), as well as its adoption by applications such as `glibc` (pthreads) and the JVM [24], indicates its importance and usefulness to the community. System administrators are probably unlikely to take such a drastic step.

Auditing: More practical but still not ideal is the class of countermeasures Ge et al. refer to as Auditing, which is based on behavioral analysis of running processes. Hardware performance counters in the target systems can be used to monitor LLC cache misses or miss rates, and thus detect when a PRIME+PROBE- or FLUSH+RELOAD-style attack is being conducted [1, 7, 46] (as any attack from those families will introduce a large number of cache misses—at least in the victim process). As a PRIME+PROBE-style attack, PRIME+ABORT would be just as vulnerable to these countermeasures as other cache attacks are. However, any behavioral auditing scheme is necessarily imperfect and subject to misclassification errors in both directions. Furthermore, any auditing proposal targeting PRIME+ABORT which specifically monitors TSX-related events, such as transactions opened or transactions aborted, seems less likely to be effective, as many benign programs which utilize TSX generate a large number of both transactions and aborts, just as PRIME+ABORT does. This makes it difficult to distinguish PRIME+ABORT from benign TSX programs based on these statistics.

Constant-Time Techniques: The class of countermeasures referred to as "Constant-Time Techniques" includes a variety of approaches, some of which are likely to be effective against PRIME+ABORT. These countermeasures are generally software techniques to ensure important invariants are preserved in program execution regardless of (secret) input data, with the aim of mitigating side channels of various types. Some "Constant-Time Techniques" merely ensure that critical functions in a program always execute in constant time regardless of secret data. This is insufficient to defend against PRIME+ABORT, as PRIME+ABORT can track cache accesses without relying on any kind of timing side-channel. However, other so-called "Constant-Time Techniques" are actually more powerful than their name suggests, and ensure that no data access or control-flow decision made by the program ever depends on any secret data. This approach is effective against PRIME+ABORT, as monitoring cache accesses (either for instructions or data) would not reveal anything about the secret data being processed by the program.

Randomizing Hardware Operations: Another interesting class of defenses proposes to insert noise into hardware operations so that side-channel measurements are more difficult. Although PRIME+ABORT is immune to such efforts related to timers, other proposals aim to inject noise into other side-channel vectors, such as cache accesses. For instance, RPcache [40] proposes to randomize the mapping between memory address and cache set, which would render PRIME+ABORT and other cache attacks much more difficult. Other proposals aim to, for instance, randomize the cache replacement policy. Important limitations of this kind of noise injection (noted by Ge et al.) include that it generally can only make side-channel attacks more difficult or less efficient (not completely impossible), and that higher levels of mitigation generally come with higher performance costs. However, these kinds of schemes seem to be promising, providing relatively lightweight countermeasures against a quite general class of side-channel attacks.

Cache Set Partitioning: Finally, a very promising class of countermeasures proposes to partition cache sets between processes, or disallow a single process to use all of the ways in any given LLC cache set. This would be a powerful defense against PRIME+ABORT or any other PRIME+PROBE variant. Some progress has been made towards implementing these defenses, such as CATalyst [28], which utilizes Intel's "Cache Allocation Technology" [18]; or "cache coloring" schemes such as STEALTHMEM [26] or that proposed by [5]. One undesirable side effect of this approach is that it would reduce the maximum size of TSX transactions, hindering legitimate users of the hardware transactional memory functionality. However, the technique is still promising as an effective defense against a wide variety of cache attacks. For more examples and details of this and other classes of side-channel countermeasures, we again refer the reader to Ge et al. [4].

Our work with PRIME+ABORT leads us to recommend the further pursuit of those classes of countermeasures which are effective against all kinds of cache attacks including PRIME+ABORT, specifically so-called "Constant-Time Techniques" (in their strict form), randomizing cache operations, or providing mechanisms for partitioning cache sets between processes.

6 Disclosure

We disclosed this vulnerability to Intel on January 30, 2017, explaining the basic substance of the vulnerability and offering more details. We also indicated our intent to submit our research on the vulnerability to USENIX Security 2017 in order to ensure Intel was alerted before it became public. We did not receive a response.

7 Conclusion

PRIME+ABORT leverages Intel TSX primitives to yield a high-precision, cross-core cache attack which does not rely on timers, negating several important classes of defenses. We have shown that leveraging TSX improves the efficiency of algorithms for dynamically generating eviction sets; that PRIME+ABORT has higher accuracy and speed on Intel's Skylake architecture than previous L3 PRIME+PROBE attacks while producing fewer false positives; and that PRIME+ABORT can be successfully employed to recover secret keys from a T-table implementation of AES. Additionally, we presented new evidence useful for all cache attacks regarding Intel's Skylake architecture: that it may differ from previous architectures in number of cache slices, and that it may use different cache replacement policies for lines involved in TSX transactions.

8 Acknowledgments

We thank our anonymous reviewers for their helpful advice and comments. We also especially thank Yuval

Yarom for his assistance in improving the quality of this work.

This material is based in part upon work supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] CHIAPPETTA, M., SAVAS, E., AND YILMAZ, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing 49* (2016), 1162–1174.

[2] DENNING, P. J. Virtual memory. *ACM Computing Surveys (CSUR) 2*, 3 (1970), 153–189.

[3] DICE, D., HARRIS, T., KOGAN, A., AND LEV, Y. The influence of malloc placement on TSX hardware transactional memory, 2015. https://arxiv.org/pdf/1504.04640.pdf.

[4] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2016).

[5] GODFREY, M. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, 2013.

[6] GOOGLE. Google Chrome Native Client SDK release notes. https://developer.chrome.com/native-client/sdk/release-notes.

[7] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Proceedings of the 13th Conference on* (2016).

[8] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium* (2015).

[9] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[10] GUANCIALE, R., NEMATI, H., BAUMANN, C., AND DAM, M. Cache storage channels: alias-driven attacks and verified countermeasures. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016).

[11] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games - bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011).

[12] HAMMARLUND, P., MARTINEZ, A. J., BAJWA, A. A., HILL, D. L., HALLNOR, E., JIANG, H., DIXON, M., DERR, M., HUNSAKER, M., KUMAR, R., ET AL. Haswell: The fourth-generation intel core processor. *IEEE Micro 34*, 2 (2014), 6–20.

[13] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News* (2004), vol. 32, IEEE Computer Society, p. 102.

[14] HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[15] HU, W.-M. Reducing timing channels with fuzzy time. *Journal of Computer Security 1*, 3-4 (1992), 233–254.

[16] İNCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *Cryptographic Hardware and Embedded Systems (CHES), Proceedings of the 18th International Conference on* (2016).

[17] INTEL. Desktop 4th generation Intel Core processor family, desktop Intel Pentium processor family, and desktop Intel Celeron processor family: specification update. Revision 036US, page 67.

[18] INTEL. Improving real-time performance by utilizing Cache Allocation Technology. Tech. rep., Intel Corporation, 2015.

[19] INTEL. *Intel 64 and IA-32 architectures software developer's manual*. September 2016.

[20] INTEL. ARK — your source for Intel product specifications, Jan 2017. https://ark.intel.com.

[21] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[22] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Systematic reverse engineering of cache slice selection in Intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on* (2015).

[23] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *Information, Computer, and Communications Security, Proceedings of the 10th ACM Symposium on* (2015).

[24] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with Intel TSX. In *Computer and Communcications Security, Proceedings of the 23rd ACM Conference on* (2016).

[25] KAYAALP, M., ABU-GHAZALEH, N., PONOMAREV, D., AND JALEEL, A. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference (DAC), Proceedings of the 53rd* (2016).

[26] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTH-MEM: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium* (2012).

[27] KOHLBRENNER, D., AND SHACHAM, H. Trusted browsers for uncertain times. In *Proceedings of the 25th USENIX Security Symposium* (2016).

[28] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High-Performance Computer Architecture (HPCA), IEEE Symposium on* (2016).

[29] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[30] MARSHALL, A., HOWARD, M., BUGHER, G., AND HARDEN, B. Security best practices for developing Windows Azure applications. Tech. rep., Microsoft Corp., 2010.

[31] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture (ISCA), Proceedings of the 39th Annual* (2012).

[32] MAURICE, C., LE SCOUARNEC, N., NEUMANN, C., HEEN, O., AND FRANCILLON, A. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses (RAID), Proceedings of the 18th Symposium on* (2015).

[33] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. 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* (2015).

[34] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2006).

[35] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan 2005* (2005).

[36] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002).

[37] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

[38] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News* (1995), vol. 23, ACM, pp. 392–403.

[39] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating fine-grained timers in Xen. In *Cloud Computing Security Workshop (CCSW), Proceedings of the 3rd ACM* (2011).

[40] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA), Proceedings of the 34th* (2007).

[41] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *European Conference on Computer Systems (EuroSys), Proceedings of the Ninth* (2014).

[42] YAROM, Y. Mastik: a micro-architectural side-channel toolkit. `http://cs.adelaide.edu.au/~yval/Mastik`. Version 0.02.

[43] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: a high-resolution, low-noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[44] YAROM, Y., GE, Q., LIU, F., LEE, R. B., AND HEISER, G. Mapping the Intel last-level cache, 2015. http://eprint.iacr.org.

[45] YEN, L., BOBBA, J., MARTY, M. R., MOORE, K. E., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), IEEE, pp. 261–272.

[46] ZHANG, T., ZHANG, Y., AND LEE, R. B. Cloudradar: a real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses (RAID), Proceedings of the 19th Symposium on* (2016).