Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory

Daniel Gruss, *Graz University of Technology, Graz, Austria;* Julian Lettner, *University of California, Irvine, USA;* Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa, *Microsoft Research, Cambridge, UK*

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

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

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory

Daniel Gruss[*], Julian Lettner[†], Felix Schuster, Olga Ohrimenko, Istvan Haller, Manuel Costa
Microsoft Research

Abstract

Cache-based side-channel attacks are a serious problem in multi-tenant environments, for example, modern cloud data centers. We address this problem with Cloak, a new technique that uses hardware transactional memory to prevent adversarial observation of cache misses on sensitive code and data. We show that Cloak provides strong protection against all known cache-based side-channel attacks with low performance overhead. We demonstrate the efficacy of our approach by retrofitting vulnerable code with Cloak and experimentally confirming immunity against state-of-the-art attacks. We also show that by applying Cloak to code running inside Intel SGX enclaves we can effectively block information leakage through cache side channels from enclaves, thus addressing one of the main weaknesses of SGX.

1 Introduction

Hardware-enforced isolation of virtual machines and containers is a pillar of modern cloud computing. While the hardware provides isolation at a logical level, physical resources such as caches are still shared amongst isolated domains, to support efficient multiplexing of workloads. This enables different forms of side-channel attacks across isolation boundaries. Particularly worrisome are cache-based attacks, which have been shown to be potent enough to allow for the extraction of sensitive information in realistic scenarios, e.g., between colocated cloud tenants [56].

In the past 20 years cache attacks have evolved from theoretical attacks [38] on implementations of cryptographic algorithms [4] to highly practical generic attack primitives [43,62]. Today, attacks can be performed in an automated fashion on a wide range of algorithms [24].

Many countermeasures have been proposed to mitigate cache side-channel attacks. Most of these countermeasures either try to eliminate resource sharing [12, 18, 42, 52, 58, 68, 69], or they try to mitigate attacks after detecting them [9, 53, 65]. However, it is difficult to identify all possible leakage through shared resources [34,55] and eliminating sharing always comes at the cost of efficiency. Similarly, the detection of cache side-channel attacks is not always sufficient, as recently demonstrated attacks may, for example, recover the entire secret after a single run of a vulnerable cryptographic algorithm [17,43,62]. Furthermore, attacks on singular sensitive events are in general difficult to detect, as these can operate at low attack frequencies [23].

In this paper, we present Cloak, a new efficient defensive approach against cache side-channel attacks that allows resource sharing. At its core, our approach prevents cache misses on sensitive code and data. This effectively conceals cache access-patterns from attackers and keeps the performance impact low. We ensure permanent cache residency of sensitive code and data using widely available hardware transactional memory (HTM), which was originally designed for high-performance concurrency.

HTM allows potentially conflicting threads to execute *transactions* optimistically in parallel: for the duration of a transaction, a thread works on a private memory snapshot. In the event of conflicting concurrent memory accesses, the transaction aborts and all corresponding changes are rolled back. Otherwise, changes become visible atomically when the transaction completes. Typically, HTM implementations use the CPU caches to keep track of transactional changes. Thus, current implementations like Intel TSX require that all accessed memory remains in the CPU caches for the duration of a transaction. Hence, transactions abort not only on real conflicts but also whenever transactional memory is evicted prematurely to DRAM. This behavior makes HTM a powerful tool to mitigate cache-based side channels.

The core idea of Cloak is to execute leaky algorithms in HTM-backed transactions while ensuring that *all* sensitive data and code reside in transactional memory for the duration of the execution. If a transaction succeeds, secret-dependent control flows and data accesses are guaranteed to stay within the CPU caches. Otherwise, the corresponding transaction would abort. As we show and discuss, this simple property can greatly raise the bar for contemporary cache side-channel attacks or even prevent them completely. The Cloak approach can be implemented on top of any HTM that provides the aforementioned basic properties. Hence, compared to other approaches [11, 42, 69] that aim to provide isola-

[*]Work done during internship at Microsoft Research; affiliated with Graz University of Technology.

[†]Work done during internship at Microsoft Research; affiliated with University of California, Irvine.

tion, Cloak does not require any changes to the operating system (OS) or kernel. In this paper, we focus on Intel TSX as HTM implementation for Cloak. This choice is natural, as TSX is available in many recent professional and consumer Intel CPUs. Moreover, we show that we can design a highly secure execution environment by using Cloak inside Intel SGX enclaves. SGX enclaves provide a secure execution environment that aims to protect against hardware attackers and attacks from malicious OSs. However, code inside SGX enclaves is as much vulnerable to cache attacks as normal code [7,20,46,57] and, when running in a malicious OS, is prone to other memory access-based leakage including page faults [10, 61]. We demonstrate and discuss how Cloak can reliably defend against such side-channel attacks on enclave code.

We provide a detailed evaluation of Intel TSX as available in recent CPUs and investigate how different implementation specifics in TSX lead to practical challenges which we then overcome. For a range of proof-of-concept applications, we show that Cloak's runtime overhead is small—between -0.8% and $+1.2\%$ for low-memory tasks and up to $+248\%$ for memory-intense tasks in SGX—while state-of-the-art cache attacks are effectively mitigated. Finally, we also discuss limitations of Intel TSX, specifically negative side effects of the aggressive and sparsely documented hardware prefetcher.

The key contributions of this work are:

- We describe Cloak, a universal HTM-based approach for the effective mitigation of cache attacks.
- We investigate the peculiarities of Intel TSX and show how Cloak can be implemented securely and efficiently on top of it.
- We propose variants of Cloak as a countermeasure against cache attacks in realistic environments.
- We discuss how SGX and TSX in concert can provide very high security in hostile environments.

Outline. The remainder of this paper is organized as follows. In Section 2, we provide background on software-based side-channel attacks and hardware transactional memory. In Section 3, we define the attacker model. In Section 4, we describe the fundamental idea of Cloak. In Section 5, we show how Cloak can be instantiated with Intel TSX. In Section 6, we provide an evaluation of Cloak on state-of-the-art attacks in local and cloud environments. In Section 7, we show how Cloak makes SGX a highly secure execution environment. In Section 8, we discuss limitations of Intel TSX with respect to Cloak. In Section 9, we discuss related work. Finally, we provide conclusions in Section 10.

2 Background

We now provide background on cache side-channel attacks and hardware transactional memory.

2.1 Caches

Modern CPUs have a hierarchy of caches that store and efficiently retrieve frequently used instructions and data, thereby, often avoiding the latency of main memory accesses. The first-level cache is the usually the smallest and fastest cache, limited to several KB. It is typically a private cache which cannot be accessed by other cores. The last-level cache (LLC), is typically unified and shared among all cores. Its size is usually limited to several MBs. On modern architectures, the LLC is typically inclusive to the lower-level caches like the L1 caches. That is, a cache line can only be in an L1 cache if it is in the LLC as well. Each cache is organized in *cache sets* and each cache set consists of multiple *cache lines* or *cache ways*. Since more addresses map to the same cache set than there are ways, the CPU employs a cache replacement policy to decide which way to replace. Whether data is cached or not is visible through the memory access latency. This is a root cause of the side channel introduced by caches.

2.2 Cache Side-Channel Attacks

Cache attacks have been studied for two decades with an initial focus on cryptographic algorithms [4, 38, 51]. More recently, cache attacks have been demonstrated in realistic cross-core scenarios that can deduce information about single memory accesses performed in other programs (*i.e.*, access-driven attacks). We distinguish between the following access-driven cache attacks: Evict+Time, Prime+Probe, Flush+Reload. While most attacks directly apply one of these techniques, there are many variations to match specific capabilities of the hardware and software environment.

In Evict+Time, the victim computation is invoked repeatedly by the attacker. In each run, the attacker selectively evicts a cache set and measures the victim's execution time. If the eviction of a cache set results in longer execution time, the attacker learns that the victim likely accessed it. Evict+Time attacks have been extensively studied on different cache levels and exploited in various scenarios [51, 60]. Similarly, in Prime+Probe, the attacker fills a cache set with their own lines. After waiting for a certain period, the attacker measures if all their lines are still cached. The attacker learns whether another process—possibly the victim—accessed the selected cache set in the meantime. While the first Prime+Probe attacks targeted the L1 cache [51, 54], more recent attacks have also been demonstrated on the LLC [43, 50, 56]. Flush+Reload [62] is a powerful but also constrained technique; it requires attacker and victim to share memory pages. The attacker selectively flushes a shared line from the cache and, after some waiting, checks if it was brought back through the victim's execution. Flush+Reload attacks have been studied extensively in different variations [2, 41, 66]. Apart from the CPU caches, the shared nature of other system resources has also been exploited in side-channel attacks. This includes different parts of the CPU's branch-prediction facility [1, 15, 40], the DRAM row buffer [5, 55], the page-translation caches [21, 28, 36] and other microarchitectural elements [14].

This paper focuses on mitigating Prime+Probe and Flush+Reload. However, Cloak conceptually also thwarts other memory-based side-channel attacks such as those that exploit the shared nature of the DRAM.

2.3 Hardware Transactional Memory

HTM allows for the efficient implementation of parallel algorithms [27]. It is commonly used to elide expensive software synchronization mechanisms [16, 63]. Informally, for a CPU thread executing a hardware transaction, all other threads appear to be halted; whereas, from the outside, a transaction appears as an atomic operation. A transaction fails if the CPU cannot provide this atomicity due to resource limitations or conflicting concurrent memory accesses. In this case, all transactional changes need to be rolled back. To be able to detect conflicts and revert transactions, the CPU needs to keep track of transactional memory accesses. Therefore, transactional memory is typically divided into a *read set* and a *write set*. A transaction's read set contains all read memory locations. Concurrent read accesses by other threads to the read set are generally allowed; however, concurrent writes are problematic and—depending on the actual HTM implementation and circumstances—likely lead to transactional aborts. Further, any concurrent accesses to the write set necessarily lead to a transactional abort. Figure 1 visualizes this exemplarily for a simple transaction with one conflicting concurrent thread.

Commercial Implementations. Implementations of HTM can be found in different commercial CPUs, among others, in many recent professional and consumer Intel CPUs. Nakaike et al. [48] investigated four commercial HTM implementations from Intel and other vendors. They found that all processors provide comparable functionality to begin, end, and abort transactions and that all implement HTM within the existing CPU cache hierarchy. The reason for this is that only caches can be held in a consistent state by the CPU itself. If data is

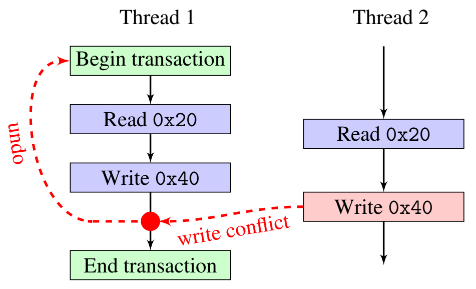

Figure 1: HTM ensures that no concurrent modifications influence the transaction, either by preserving the old value or by aborting and reverting the transaction.

evicted to DRAM, transactions necessarily abort in these implementations. Nakaike et al. [48] found that all four implementations detected access conflicts at cache-line granularity and that failed transactions were reverted by invalidating the cache lines of the corresponding write sets. Depending on the implementation, read and write set can have different sizes, and set sizes range from multiple KB to multiple MB of HTM space.

Due to HTM usually being implemented within the CPU cache hierarchy, HTM has been proposed as a means for optimizing cache maintenance and for performing security-critical on-chip computations: Zacharopoulos [64] uses HTM combined with prefetching to reduce the system energy consumption. Guan et al. [25] designed a system that uses HTM to keep RSA private keys encrypted in memory and only decrypt them temporarily inside transactions. Jang et al. [36] used hardware transaction aborts upon page faults to defeat kernel address-space layout randomization.

3 Attacker Model

We consider multi-tenant environments where tenants do not trust each other, including local and cloud environments, where malicious tenants can use shared resources to extract information about other tenants. For example, they can influence and measure the state of caches via the attacks described in Section 2.2. In particular, an attacker can obtain a high-resolution trace of its own memory access timings, which are influenced by operations of the victim process. More abstractly, the attacker can obtain a trace where at each time frame the attacker learns whether the victim has accessed a particular memory location. We consider the above attacker in three realistic environments which give her different capabilities:

Cloud We assume that the processor, the OS and the hypervisor are trusted in this scenario while other cloud tenants are not. This enables the attacker to launch cross-VM Prime+Probe attacks.

Local This scenario is similar to the Cloud scenario, but we assume the machine is not hosted in a cloud environment. Therefore, the tenants share the machine in a traditional time-sharing fashion and the OS is trusted to provide isolation between tenants. Furthermore, we assume that there are shared libraries between the victim and the attacker, since this is a common optimization performed by OSs. This enables the attacker to launch Flush+Reload attacks, in addition to Prime+Probe attacks.

SGX In this scenario, the processor is trusted but the adversary has full control over the OS, the hypervisor, and all other code running on the system, except the victim's code. This scenario models an SGX-enabled environment, where the victim's code runs inside an enclave. While the attacker has more control over the software running on the machine, the SGX protections prevent sharing of memory pages between the enclave and untrusted processes, which renders Flush+Reload attacks ineffective in this setting.

All other side-channels, including power analysis, and channels based on shared microarchitectural elements other than caches are outside our scope.

4 Hardware Transactional Memory as a Side-Channel Countermeasure

The foundation of all cache side-channel attacks are the timing differences between cache hits and misses, which an attacker tries to measure. The central idea behind Cloak is to instrument HTM to prevent any cache misses on the victim's sensitive code and data. In Cloak, all sensitive computation is performed in HTM. Crucially, in Cloak, all security-critical code and data is deterministically *preloaded* into the caches at the beginning of a transaction. This way, security-critical memory locations become part of the read or write set and all subsequent, possibly secret-dependent, accesses are guaranteed to be served from the CPU caches. Otherwise, in case any preloaded code or data is evicted from the cache, the transaction necessarily aborts and is reverted. (See Listing 1 for an example that uses the TSX instructions `xbegin` and `xend` to start and end a transaction.)

Given an *ideal* HTM implementation, Cloak thus prevents that an attacker can obtain a trace that shows whether the victim has accessed a particular memory location. More precisely, in the sense of Cloak, ideal HTM has the following properties:

R1 Both data and code can be added to a transaction as transactional memory and thus are included in the HTM atomicity guarantees.

Listing 1: A vulnerable crypto operation protected by Cloak instantiated with Intel TSX; the `AES_encrypt` function makes accesses into `lookup_tables` that depend on `key`. Preloading the tables and running the encryption code within a HTM transaction ensures that eviction of table entries from LLC will terminate the code before it may cause a cache miss.

```
if ((status = _xbegin ()) == _XBEGIN_STARTED) {
  for (auto p : lookup_tables)
    *(volatile size_t*)p;
  AES_encrypt(plaintext, ciphertext, &key);
  _xend ();
}
```

R2 A transaction aborts immediately when any part of transactional memory leaves the cache hierarchy.

R3 All pending transactional memory accesses are purged during a transactional abort.

R4 Prefetching decisions outside of transactions are not influenced by transactional memory accesses.

R1 ensures that all sensitive code and data can be added to the transactional memory in a deterministic and leakage-free manner. **R2** ensures that any cache line evictions are detected implicitly by the HTM and the transaction aborts before any non-cached access is performed. **R3** and **R4** ensure that there is no leakage after a transaction has succeeded or aborted.

Unfortunately, commercially available HTM implementations and specifically Intel TSX do not precisely provide **R1**–**R4**. In the following Section 5 we discuss how Cloak can be instantiated on commercially available (and not ideal) HTM, what leakage remains in practice, and how this can be minimized.

5 Cloak based on Intel TSX

Cloak can be built using an HTM that satisfies **R1**–**R4** established in the previous section. We propose Intel TSX as an instantiation of HTM for Cloak to mitigate the cache side channel. In this section, we evaluate how far Intel TSX meets **R1**–**R4** and devise strategies to address those cases where it falls short. All experiments we report on in this section were executed on Intel Core i7 CPUs of the Skylake generation (i7-6600U, i7-6700, i7-6700K) with 4MB or 8MB of LLC. The source code of these experiments will be made available at http://aka.ms/msr-cloak.

5.1 Meeting Requirements with Intel TSX

We summarize our findings and then describe the methodology.

R1 and **R2** hold for data. It supports read-only data that does not exceed the size of the LLC (several MB) and write data that does not exceed the size of the L1 cache (several KB);

R1 and **R2** hold for code that does not exceed the size of the LLC;

R3 and **R4** hold in the *cloud* and *SGX* attacker scenarios from Section 3, but not in general for *local* attacker scenarios.

5.1.1 Requirements 1&2 for Data

Our experiments and previous work [19] find the read set size to be ultimately constrained by the size of the LLC: Figure 2 shows the failure rate of a simple TSX transaction depending on the size of the read set. The abort rate reaches 100% as the read set size approaches the limits of the LLC (4MB in this case). In a similar experiment, we observed 100% aborts when the size of data written in a transaction exceeded the capacity of the L1 data cache (32 KB per core). This result is also confirmed in Intel's *Optimization Reference Manual* [30] and in previous work [19, 44, 64].

Conflicts and Aborts. We always observed aborts when read or write set cache lines were actively evicted from the caches by concurrent threads. That is, evictions of write set cache lines from the L1 cache and read set cache lines from the LLC are sufficient to cause aborts. We also confirmed that transactions abort shortly after cache line evictions: using concurrent `clflush` instructions on the read set, we measured abort latencies in the order of a few hundred cycles (typically with an upper bound of around 500 cycles). In case varying abort times should prove to be an issue, the attacker's ability to measure them, e.g., via Prime+Probe on the abort handler, could be thwarted by *randomly* choosing one out of many possible abort handlers and rewriting the `xbegin` instruction accordingly,[1] before starting a transaction.

Tracking of the Read Set. We note that the data structure that is used to track the read set in the LLC is unknown. The Intel manual states that "an implementation-specific second level structure" may be available, which probabilistically keeps track of the addresses of read-set cache lines that were evicted from the L1 cache. This structure is possibly an on-chip bloom filter, which tracks the read-set membership of cache lines in a probabilistic manner that may give false positives but no false negatives.[2] There may exist so far unknown leaks through this data structure. If this is a concern, all sensitive data (including read-only data) can be kept in the write set in L1. However, this limits the working set to the L1 and also requires all data to be stored in writable memory.

[1]The 16-bit relative offset to a transaction's abort handler is part of the `xbegin` instruction. Hence, for each xbegin instruction, there is a region of 1 024 cache lines that can contain the abort handler code.

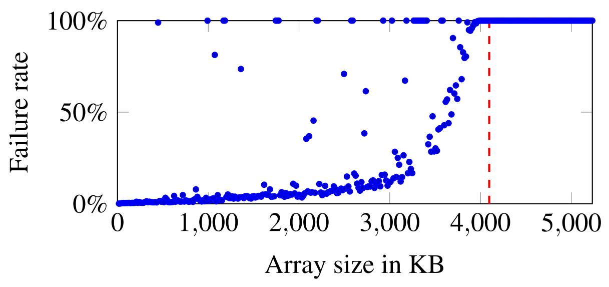

Figure 2: A TSX transaction over a loop reading an array of increasing size. The failure rate reveals how much data can be read in a transaction. Measured on an i7-6600U with 4 MB LLC.

L1 Cache vs. LLC. By adding all data to the write set, we can make **R1** and **R2** hold for data with respect to the L1 cache. This is important in cases where victim and attacker potentially share the L1 cache through hyper-threading.[3] Shared L1 caches are not a concern in the *cloud* setting, where it is usually ensured by the hypervisor that corresponding hyper-threads are not scheduled across different tenants. The same can be ensured by the OS in the *local* setting. However, in the *SGX* setting a malicious OS may misuse hyper-threading for an L1-based attack. To be not constrained to the small L1 in SGX nonetheless, we propose solutions to detect and prevent such attacks later on in Section 7.2.

We conclude that Intel TSX sufficiently fulfills **R1** and **R2** for data if the read and write sets are used appropriately.

5.1.2 Requirements 1&2 for Code

We observed that the amount of code that can be executed in a transaction seems not to be constrained by the sizes of the caches. Within a transaction with strictly no reads and writes we were reliably able to execute more

[2]In Intel's *Software Development Emulator* [29] the read set is tracked probabilistically using bloom filters.

[3]Context switches may also allow the attacker to examine the victim's L1 cache state "postmortem". While such attacks may be possible, they are outside our scope. TSX transactions abort on context switches.

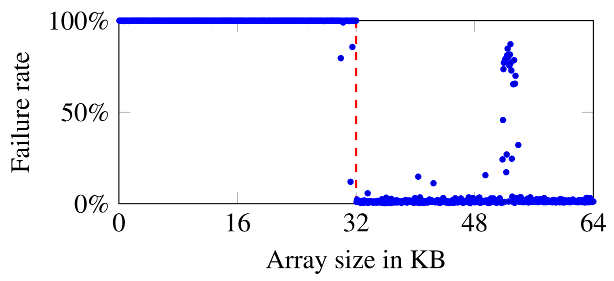

Figure 3: A TSX transaction over a nop-sled with increasing length. A second thread waits and then flushes the first cache line once before the transaction ends. The failure rate starts at 100% for small transaction sizes. If the transaction self-evicts the L1 instruction cache line, e.g., when executing more than 32 KB of instructions, the transaction succeeds despite of the flush. Measured on an i7-6600U with 32 KB L1 cache.

than 20 MB of nop instructions or more than 13 MB of arithmetic instructions (average success rate ~10%) on a CPU with 8 MB LLC. This result strongly suggests that executed code does not become part of the read set and is in general not explicitly tracked by the CPU.

To still achieve **R1** and **R2** for code, we attempted to make code part of the read or write set by accessing it through load/store operations. This led to mixed results: even with considerable effort, it does not seem possible to reliably execute cache lines in the write set without aborting the transaction.[4] In contrast, it is generally possible to make code part of the read set through explicit loads. This gives the same benefits and limitations as using the read set for data.

Code in the L1 Cache. Still, as discussed in the previous Section 5.1.1, it can be desirable to achieve **R1** and **R2** for the L1 cache depending on the attack scenario. Fortunately, we discovered undocumented microarchitectural effects that reliably cause transactional aborts in case a recently executed cache line is evicted from the cache hierarchy. Figure 3 shows how the transactional abort rate relates to the amount of code that is executed inside a transaction. This experiment suggests that a concurrent (hyper-) thread can cause a transactional abort by evicting a transactional code cache line currently in the L1 instruction cache. We verified that this effect exists for direct evictions through the `clflush` instruction as well as indirect evictions through cache set conflicts. However, self-evictions of L1 code cache lines (that is, when a transactional code cache line is replaced by another one) do not cause transactional aborts. Hence, forms of **R1** and **R2** can also be ensured for code in the L1 instruction cache without it being part of the write set.

In summary, we can fulfill requirements **R1** and **R2** by moving code into the read set or, using undocumented microarchitectural effects, by limiting the amount of code to the L1 instruction cache and preloading it via execution.

[4]In line with our observation, Intel's documentation [31] states that "executing self-modifying code transactionally may also cause transactional aborts".

5.1.3 Requirements 3&4

As modern processors are highly parallelized, it is difficult to guarantee that memory fetches outside a transaction are not influenced by memory fetches inside a transaction. For precisely timed evictions, the CPU may still enqueue a fetch in the memory controller, *i.e.*, a race condition. Furthermore, the hardware prefetcher is triggered if multiple cache misses occur on the same physical page within a relatively short time. This is known to introduce noise in cache attacks [24,62], but also to introduce side-channel leakage [6].

In an experiment with shared memory and a cycle-accurate alignment between attacker and victim, we investigated the potential leakage of Cloak instantiated with Intel TSX. To make the observable leakage as strong as possible, we opted to use Flush+Reload for the attack primitive. We investigated how a delay between the transaction start and the flush operation and a delay between the flush and the reload operations influence the probability that an attacker can observe a cache hit against code or data placed into transactional memory. The victim in this experiment starts the transaction, by placing data and code into transactional memory in a uniform manner (using either reads, writes or execution). The victim then simulates meaningful program flow, followed by an access to one of the sensitive cache lines and terminating the transaction. The attacker "guesses" which cache line the victim accessed and probes it. Ideally, the attacker should not be able to distinguish between correct and wrong guesses.

Figure 4 shows two regions where an attacker could observe cache hits on a correct guess. The left region corresponds to the preloading of sensitive code/data at the beginning of the transaction. As expected, cache hits in this region were observed to be identical to runs where the attacker had the wrong guess. On the other hand, the right region is unique to instances where the attacker made a correct guess. This region thus corresponds to a window of around 250 cycles, where an attacker could potentially obtain side-channel information. We explain the existence of this window by the fact that Intel did not design TSX to be a side-channel free primitive, thus **R3** and **R4** are not guaranteed to hold and a limited amount of leakage remains. We observed identical high-level re-

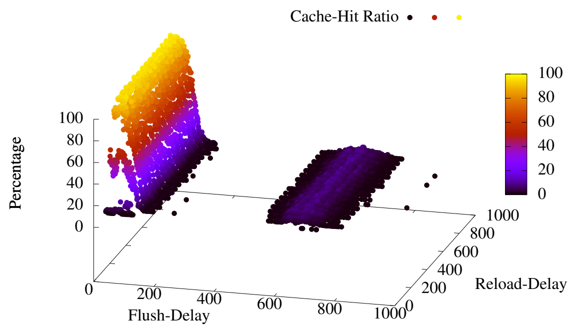

Figure 4: Cache hits observed by a Flush+Reload attacker with the ability to overlap the attack with different segments of the victim's transaction. Cache hits can be observed both in the region where the victim tries to prepare its transactional memory, as well as in a small window around a secret access. The Z axis represents the success rate of the attacker observing a cache hit.

sults for all forms of preloading (reading, writing, executing) and all forms of secret accesses (reading, writing, executing).

To exploit the leakage we found, the attacker has to be able to determine whether the CPU reloaded a secret-dependent memory location. This is only possible if the attacker shares the memory location with the victim, *i.e.*, only in *local* attacks but not in other scenarios. Furthermore, it is necessary to align execution between attacker and victim to trigger the eviction in exactly the right cycle range in the transaction. While these properties might be met in a Flush+Reload attack with fast eviction using `clflush` and shared memory, it is rather unlikely that an attack is possible using Prime+Probe, due to the low frequency of the attack [22] and the cache replacement policy. Thus, we conclude that requirements **R3** and **R4** are likely fulfilled in all scenarios where the attacker can only perform Prime+Probe, but not Flush+Reload, *i.e.*, cloud and SGX scenarios. Furthermore, requirements **R3** and **R4** are likely to be fulfilled in scenarios where an attacker can perform Flush+Reload, but not align with a victim on a cycle base nor measure the exact execution time of a TSX transaction, *i.e.*, the local scenario.

5.2 Memory Preloading

Using right the memory preloading strategy is crucial for the effectiveness of Cloak when instantiated on top of TSX. In the following, we describe preloading techniques for various scenarios. The different behavior for read-only data, writable data, and code, makes it necessary to preload these memory types differently.

5.2.1 Data Preloading

As discussed, exclusively using the write set for preloading has the benefit that sensitive data is guaranteed to stay within the small L1 cache, which is the most secure option. To extend the working set beyond L1, sensitive read-only data can also be kept in the LLC as described in Section 5.1.1. However, when doing so, special care has to be taken. For example, naïvely preloading a large ($> 32\,$KB) sequential read set after the write set leads to assured abortion during preloading, as some write set cache-lines are inevitably evicted from L1. Reversing the preloading order, *i.e.*, read set before write set, partly alleviates this problem, but, depending on the concrete read set access patterns, one is still likely to suffer from aborts during execution caused by read/write set conflicts in the L1 cache. In the worst case, such self-eviction aborts may leak information.

To prevent such conflicts, in Cloak, we reserve certain cache sets in L1 entirely for the write set. This is possible as the L1 cache-set index only depends on the virtual address, which is known at runtime. For example, reserving the L1 cache sets with indexes 0 and 1 gives a conflict-free write set of size $2 \cdot 8 \cdot 64B = 1KB$. For this allocation, it needs to be ensured that the same 64 B cache lines of *any* 4 KB page are not part of the read set (see Figure 5 for an illustration). Conversely, the write set is placed in the same 64 B cache lines in up to eight different 4 KB pages. (Recall that an L1 cache set comprises eight ways.) Each reserved L1 cache set thus blocks $1/64^{th}$ of the entire virtual memory from being used in the read set.

While this allocation strategy plays out nicely in theory, we observed that apparently the CPU's data prefetcher [30] often optimistically pulled-in unwanted cache lines that were conflicting with our write set. This can be mitigated by ensuring that sequentially accessed read cache lines are separated by a page boundary from write cache lines and by adding "safety margins" between read and write cache lines on the same page.

In general, we observed benefits from performing preloading similar to recent Prime+Probe attacks [22, 45], where a target address is accessed multiple times and interleaved with accesses to other addresses. Further, we observed that periodic "refreshing" of the write set, e.g., using the `prefetchw` instruction, reduced the chances of write set evictions in longer transactions.

5.2.2 Code Preloading

As described in Section 5.1.2, we preload code into the read set and optionally into the L1 instruction cache. To preload it into the read set, we use the same approach as for data. However, to preload the code into the L1 instruction cache we cannot simply execute the function,

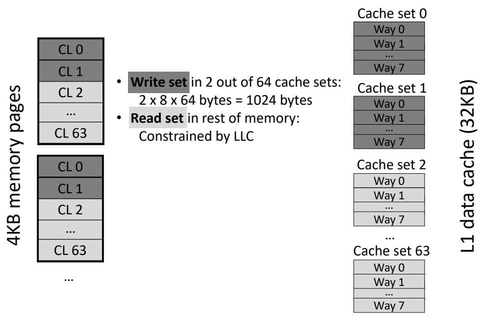

Figure 5: Allocation of read and write sets in memory to avoid conflicts in the L1 data cache

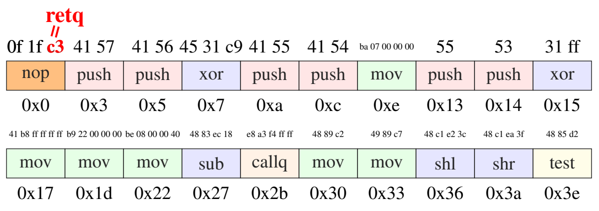

Figure 6: Cache lines are augmented with a multi-byte `nop` instruction. The `nop` contains a byte `c3` which is the opcode of `retq`. By jumping directly to the `retq` byte, we preload each cache line into the L1 instruction cache.

as this would have unwanted side effects. Instead, we insert a multi-byte `nop` instruction into every cache line, as shown in Figure 6. This `nop` instruction does not change the behavior of the code during actual function execution and only has a negligible effect on execution time. However, the multi-byte `nop` instruction allows us to incorporate a byte `c3` which is the opcode of `retq`. Cloak jumps to this return instruction, loading the cache line into the instruction L1 cache but not executing the actual function. In the preloading phase, we perform a call to each such `retq` instruction in order to load the corresponding cache lines into the L1 instruction cache. The `retq` instruction immediately returns to the preloading function. Instead of `retq` instructions, equivalent `jmp reg` instructions can be inserted to avoid touching the stack.

5.2.3 Splitting Transactions

In case a sensitive function has greater capacity requirements than those provided by TSX, the function needs to be split into a series of smaller transactional units. To prevent leakage, the control flow between these units and their respective working sets needs to be input-independent. For example, consider a function `f()` that iterates over a fixed-size array, e.g., in order to update certain elements. By reducing the number of loop iterations in `f()` and invoking it separately on fixed parts of the target array, the working set for each individual transaction is reduced and chances for transactional aborts decrease. Ideally, the splitting would be done in an automated manner by a compiler. In a context similar to ours though not directly applicable to Cloak, Shih et al. [59] report on an extension of the Clang compiler that automatically splits transactions into smaller units with TSX-compatible working sets. Their approach is discussed in more detail in Section 9.

5.3 Toolset

We implemented the read-set preloading strategy from Section 5.2.1 in a small C++ container template library. The library provides generic read-only and writable arrays, which are allocated in "read" or "write" cache lines respectively. The programmer is responsible for arranging data in the specialized containers before invoking a Cloak-protected function. Further, the programmer decides which containers to preload. Most local variables and input and output data should reside in the containers. Further, all sub-function calls should be inlined, because each `call` instruction performs an implicit write of a return address. Avoiding this is important for large read sets, as even a single unexpected cache line in the write set can greatly increase the chances for aborts.

We also extended the Microsoft C++ compiler version 19.00. For programmer-annotated functions on Windows, the compiler adds code for starting and ending transactions, ensures that all code cache lines are preloaded (via read or execution according to Section 5.2.2) and, to not pollute the write set, refrains from unnecessarily spilling registers onto the stack after preloading. Both library and compiler are used in the SGX experiments in Section 7.1.

6 Retrofitting Leaky Algorithms

To evaluate Cloak, we apply it to existing weak implementations of different algorithms. We demonstrate that in all cases, in the *local* setting (Flush+Reload) as well as the *cloud* setting (Prime+Probe), Cloak is a practical countermeasure to prevent state-of-the-art attacks. All experiments in this section were performed on a mostly idle system equipped with a Intel i7-6700K CPU with 16 GB DDR4 RAM, running a default-configured Ubuntu Linux 16.10. The applications were run as regular user programs, not pinned to CPU cores, but sharing CPU cores with other threads in the system.

6.1 OpenSSL AES T-Tables

As a first application of Cloak, we use the AES T-table implementation of OpenSSL which is known to be sus-

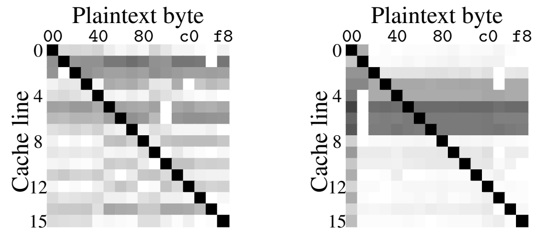

Figure 7: Color matrix showing cache hits on an AES T-table. Darker means more cache hits. Measurement performed over roughly 2 billion encryptions. Prime+Probe depicted on the left, Flush+Reload on the right.

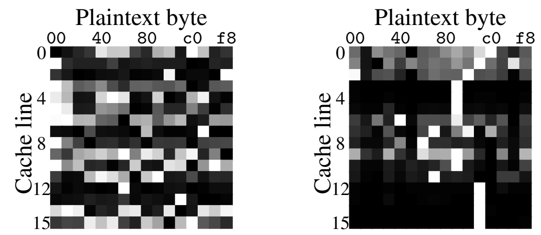

Figure 8: Color matrix showing cache hits on an AES T-table. The implementation is protected by Cloak. Darker means more cache hits. Measurement performed over roughly 3 billion transactions (500 million encryptions) for Prime+Probe (left) and 4.9 billion transactions (1.5 million encryptions) for Flush+Reload (right). The side-channel leakage is not visible in both cases.

ceptible to cache attacks [4, 24, 26, 33, 35, 51]. In this implementation, AES performs 16 lookups to 4 different T-tables for each of the 10 rounds and combines the values using `xor`. The table lookups in the first round of AES are $T_j[x_i = p_i \oplus k_i]$ where p_i is a plaintext byte, k_i a key byte, and $i \equiv j \mod 4$. A typical attack scenario is a known-plaintext attack. By learning the cache line of the lookup index x_i an attacker learns the upper 4 bits of the secret key byte $x_i \oplus p_i = k_i$. We wrap the entire AES computation together with the preloading step into a single TSX transaction. The preloading step fetches the 4 T-Tables, *i.e.*, it adds 4 KB of data to the read set.

We performed roughly 2 billion encryptions in an asynchronous attack and measured the cache hits on the T-table cache lines using Prime+Probe and Flush+Reload. Figure 7 is a color matrix showing the number of cache hits per cache line and plaintext-byte value. When protecting the T-tables with Cloak (cf. Figure 8), the leakage from Figure 7 is not present anymore.

We fixed the time for which the fully-asynchronous known-plaintext attack is run. The amount of time corresponds to roughly 2 billion encryptions in the baseline implementation. For the AES T-Table implementation protected with Cloak we observed a significant performance difference based on whether or not an attack is running simultaneously. This is due to the TSX transaction failing more often if under attack.

While not under attack the implementation protected with Cloak started 0.8% more encryptions than the baseline implementation (*i.e.*, with preloading) and less than 0.1% of the transactions failed. This is not surprising, as the execution time of the protected algorithm is typically below 500 cycles. Hence, preemption or interruption of the transaction is very unlikely. Furthermore, cache evictions are unlikely because of the small read set size and optimized preloading (cf. Section 5.2.1). Taking the execution time into account, the implementation protected with Cloak was 0.8% faster than the baseline implementation. This is not unexpected, as Zacharopoulos [64] already found that TSX can improve performance.

Next, we measured the number of transactions failing under Prime+Probe and Flush+Reload. We observed 82.7% and 99.97% of the transactions failing for each attack, respectively. Failing transactions do not consume the full amount of time that one encryption would take as they abort earlier in the function execution. Thus, the protected implementation started over 37% more encryptions as compared to the baseline implementation when under attack using Prime+Probe and 2.53 times the encryptions when under attack using Flush+Reload. However, out of these almost 3 billion transactions only 500 million transactions succeeded in the case of Prime+Probe. In the case of Flush+Reload only 1.4 million out of 4.9 billion transactions succeeded. Thus, in total the performance of our protected implementation under a Prime+Probe attack is only 23.7% of the performance of the baseline implementation and only 0.06% in the case of a Flush+Reload attack.

The slight performance gain of Cloak while not being actively attacked shows that deploying our countermeasure for this use case does not only eliminate cache side-channel leakage but it can also be beneficial. The lower performance while being attacked is still sufficient given that leakage is eliminated, especially as the attacker has to keep one whole CPU core busy to perform the attack and this uses up a significant amount of hardware resources whether or not Cloak is deployed.

6.2 Secret-dependent execution paths

Powerful attacks allow to recover cryptographic keys [62] and generated random numbers by monitoring execution paths in libraries [67]. In this example, we model such a scenario by executing one of 16 functions based on a secret value that the attacker tries to learn—adapted from the AES example. Like in previous Flush+Reload attacks [62, 67], the attacker

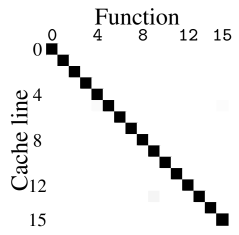

Figure 9: Color matrix showing cache hits on function code. Darker means more cache hits. Measurement performed over roughly 100 million function executions for Prime+Probe (left) and 10 million function executions for Flush+Reload (right).

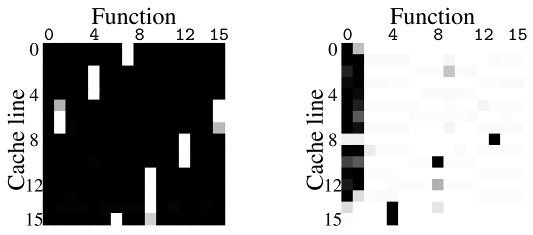

Figure 10: Color matrix showing cache hits on function code protected using Cloak. Darker means more cache hits. Measurement performed over roughly 1.5 billion transactions (77 314 function executions) for Prime+Probe (left) and 2 billion transactions (135 211 function executions) for Flush+Reload (right). Side-channel leakage is not visible in both cases.

monitors the function addresses for cache hits and thus derives which function has been called. Each of the 16 functions runs only a small code snippet consisting of a loop counting from 0 to 10 000. We wrap the entire switch-case together with the preloading step into a single TSX transaction. The preloading step fetches the 16 functions, each spanning two cache lines, *i.e.*, 2 KB of code are added to the read set.

As in the previous example, Cloak eliminates all leakage (cf. Figure 10). While not under attack the victim program protected with Cloak started 0.7% more function executions than the baseline implementation. Less than 0.1% of the transactions failed, leading to an overall performance penalty of 1.2%. When under attack using Prime+Probe, 11.8 times as many function executions were started and with Flush+Reload, 19 times as many. However, only 0.005% of the transactions succeeded in the case of Prime+Probe and only 0.0006% in the case of Flush+Reload. Thus, overall the performance is reduced to 0.03% of the baseline performance when under a Prime+Probe attack and 0.14% when under a Flush+Reload attack. The functions are 20 times slower than

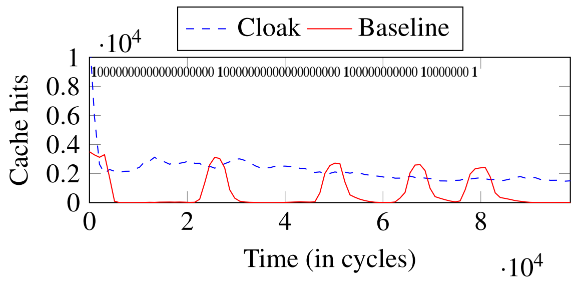

Figure 11: Cache traces for the multiply routine (as used in RSA) over 10 000 exponentiations. The secret exponent is depicted as a bit sequence. Measurement performed over 10 000 exponentiations. The variant protected with Cloak does not have visual patterns that correspond to the secret exponent.

the AES encryptions from the previous example. Thus, the high failure rate is not unexpected, as there is more time for cache evictions caused by other processes.

It is important to note that the performance under attack is not essential as the attacker simultaneously keeps one or more CPU cores on full load, accounting for a significant performance loss with and without Cloak.

6.3 RSA Square-and-Multiply example

We now demonstrate an attack against a square-and-multiply exponentiation and how Cloak allows to protect it against cache side-channel attacks. Square-and-multiply is commonly used in cryptographic implementations of algorithms such as RSA and is known to be vulnerable to side-channel attacks [54, 62]. Though cryptographic libraries move to constant-time exponentiations that are intended to not leak any information through the cache, we demonstrate our attack and protection on a very vulnerable schoolbook implementation. A square-and-multiply algorithm takes 100 000 cycles to complete. Thus, wrapping the whole algorithm in one TSX transaction has only a very low chance of success by itself. Instead we split the loop of the square-and-multiply algorithm into one small TSX transaction per exponent bit, *i.e.*, adding the `xbegin` and `xend` instructions and the preloading step to the loop. This way, we increase the success rate of the TSX transactions significantly, while still leaking no information on the secret exponent bits. The preloading step fetches 1 cache line per function, *i.e.*, 128 B of code are added to the read set.

Figure 11 shows a Flush+Reload cache trace for the multiply routine as used in RSA. The plot is generated over 10 000 exponentiation traces. Each trace is aligned by the first cache hit on the multiply routine that was measured per trace. The traces are then summed to produce the functions that are plotted. The baseline imple-

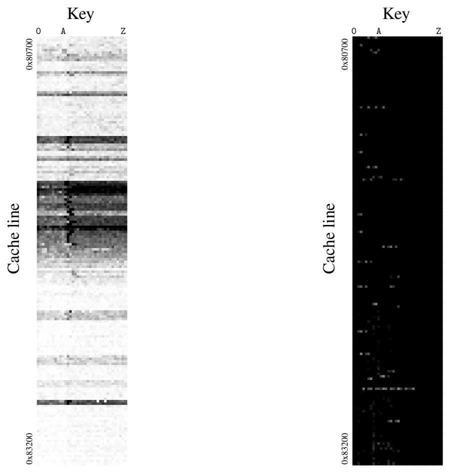

Figure 12: Cache template matrix showing cache hits on the binary search in `_gdk_keyval_from_name` without protection (left) and with Cloak (right). Darker means more cache hits. All measurements were performed with Flush+Reload. The pattern of the binary search is clearly visible for the unprotected implementation and not visible anymore when protected with Cloak.

mentation has a clear peak for each 1 bit in the secret exponent. The same implementation protected with Cloak shows no significant changes in the cache hits over the full execution time.

While not under attack, the performance of the implementation protected with Cloak is only slightly lower than the performance of the unprotected baseline implementation. To evaluate the performance in this case we performed 1 million exponentiations. During these 1 million exponentiations, only 0.5% of the transactions failed. The total runtime overhead we observed while not under attack was 1.1%. Unsurprisingly, while under attack we observed a significantly higher overhead of factor 982. This is because 99.95% of the transactions failed, *i.e.*, the transactions for almost every single bit failed and had to be repeated.

6.4 GTK keystroke example

We investigated leakage in the GTK framework, which performs a binary search to translate raw keyboard inputs to platform-independent key names and key values. Gruss et al. [24] demonstrated that this leaks significant information on single keys typed by a user, in an automated cache template attack. Their attack on GDK library version 3.10.8 has partially been resolved on current Linux systems with GDK library version 3.18.9. Instead of multiple binary searches that leak information we only identified one binary search that is still performed upon every keystroke.

In order to demonstrate the general applicability of Cloak, we reproduced the attack by Gruss et al. [24] on a recent version of the GDK library (3.18.9) which comes with Ubuntu 16.10. We attack the binary search in `_gdk_keyval_from_name` which is executed upon every keystroke in a GTK window. As shown in Figure 12, the cache template matrix of the unprotected binary search reveals the search pattern, narrowing down on the darker area where the letter keys are and thus the search ends. In case of the implementation protected by Cloak, the search pattern is disguised. With the keystroke information protected by Cloak, we could neither measure a difference in the perceived latency when typing through a keyboard, nor measure and overall increase of the system load or execution time of processes. The reason for this is that keystroke processing involves hundreds of thousands of CPU cycles spent in drivers and other functions. Furthermore, keystrokes are rate-limited by the OS and constrained by the speed of the user typing. Thus, the overhead we introduce is negligible for the overall latency and performance.

We conclude that Cloak can be used as a practical countermeasure to prevent cache template attacks on fine-grained information such as keystrokes.

7 Side-Channel Protection for SGX

Intel SGX provides an isolated execution environment called *enclave*. All code and data inside an enclave is shielded from the rest of the system and is even protected against hardware attacks by means of strong memory encryption. However, SGX enclaves use the regular cache hierarchy and are thus vulnerable to cache side-channel attacks. Further, as enclaves are meant to be run on untrusted hosts, they are also susceptible to a range of other side-channel attacks such as OS-induced page faults [61] and hardware attacks on the memory bus. In this section, we first retrofit a common machine learning algorithm with Cloak and evaluate its performance in SGX. Afterwards, we explore the special challenges that enclave code faces with regard to side channels and design extended countermeasures on top of Cloak. Specifically, we augment sensitive enclave code with Cloak and require that the potentially malicious OS honors a special *service contract* while this code is running.

7.1 Secure Decision Tree Classification

To demonstrate Cloak's applicability to the SGX environment and its capability to support larger working sets, we adapted an existing C++ implementation of a *decision tree classification* algorithm [49] using the toolset described in Section 5.3. The algorithm traverses a decision tree for an input record. Each node of the tree contains a predicate which is evaluated on features of the input record. As a result, observations of unprotected tree traversal can leak information about the tree and the input record. In this particular case, several trees in a so-called *decision forest* are traversed for each input record.

Our Cloak-enhanced implementation of the algorithm contains three programmer-annotated functions, which translates into three independent transactions. The most complex of these traverses a preloaded tree for a batch of preloaded input records. The batching of input records is crucial here for performance, as it amortizes the cost of preloading a tree. We give a detailed explanation and a code sample of tree traversal with Cloak in Appendix A.

Evaluation. We compiled our implementation for SGX enclaves using the extended compiler and a custom SGX software stack. We used a pre-trained decision forest for the *Covertype* data set from the *UCI Machine Learning Repository*[5]. Each tree in the forest consists of 30 497—32 663 nodes and has a size of 426 KB–457 KB. Each input record is a vector of 54 floating point values. We chose the *Covertype* data set because it produces large trees and was also used in previous work by Ohrimenko et al. [49], which also mitigates side channel leakage for enclave code.

We report on experiments executed on a mostly idle system equipped with a TSX and SGX-enabled Intel Core i7-6700 CPU and 16 GB DDR4 RAM running Windows Server 2016 Datacenter. In our container library, we reserved eight L1 cache sets for writable arrays, resulting in an overall write set size of 4 KB. Figure 13 shows the cycles spent inside the enclave (including entering and leaving the enclave) per input record averaged over ten runs for differently sized input batches. These batches were randomly drawn from the data set. The sizes of the batches ranged from 5 to 260. For batches larger than 260, we observed capacity aborts with high probability. Nonetheless, seemingly random capacity aborts could also be observed frequently even for small batch sizes. The number of aborts also increased with higher system load. The cost for restarting transactions on aborts is included in Figure 13.

As baseline, we ran inside SGX the same algorithm without special containers and preloading and without transactions. The baseline was compiled with the unmodified version of the Microsoft C++ compiler at the highest optimization level. As can be seen, the number of cycles per query greatly decreases with the batch size. Batching is particularly important for Cloak, because it enables the amortization of cache preloading costs. Overall, the overhead ranges between +79% (batch size 5) and +248% (batch size 260). The overhead increases

[5]https://archive.ics.uci.edu/ml

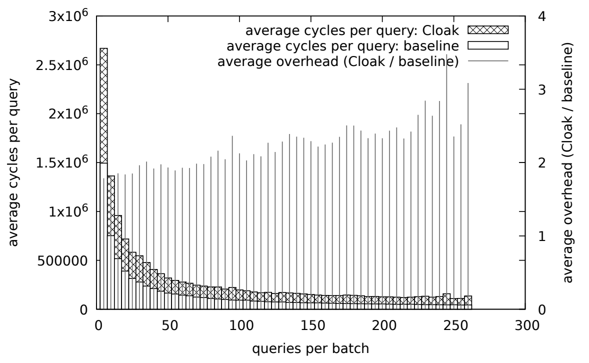

Figure 13: Average number of cycles per query for decision forest batch runs of different sizes.

with the batch size, because the baseline also profits from batching (*i.e.*, "cache warming" effects and amortization of costs for entering/leaving the enclave), while the protected version experiences more transactional aborts for larger batches. We also ran a similar micro-benchmark outside SGX with more precise timings. Here, the effect of batching was even clearer: for a batch size of 5, we observed a very high overhead of +3 078%, which gradually decreased to +216% for a batch size of 260.

Even though the experimental setting in Ohrimenko et al. [49] is not the same as ours (for instance they used the official Intel SGX SDK, an older version of the compiler, and their input data was encrypted) and they provide different guarantees, we believe that their reported overhead of circa +6 200% for a single query to SGX highlights the potential efficiency of Cloak.

7.2 Service Contracts with the OS

Applying the basic Cloak techniques to sensitive enclave code reduces the risk of side-channel attacks. However, enclave code is especially vulnerable as the corresponding attacker model (see Section 3) includes malicious system software and hardware attacks. In particular, malicious system software, *i.e.*, the OS, can amplify side-channel attacks by concurrently (**A1**) interrupting and resuming enclave threads [40], (**A2**) unmapping enclave pages [61], (**A3**) taking control of an enclave thread's sibling hyper-thread (HT) [11], or (**A4**) repeatedly resetting an enclave. **A3** is of particular concern in Cloak as TSX provides requirement **R2** (see Section 4) only for the LLC. Hence, code and data in the read set are not protected against a malicious HT which can perform attacks over the L1 and L2 caches from outside the enclave. In the following, we describe how Cloak-protected enclave code can ensure that the OS is honest and does not mount attacks **A1**–**A4**.

7.2.1 Checking the Honesty of the OS

While SGX does not provide functionality to directly check for **A1** and **A2** or to prevent them, it is simple with Cloak: our experiments showed in line with Intel's documentation [31] that transactions abort with code OTHER (no bits set in the abort code) in case of interrupts or exceptions. In case unexpected aborts of this type occur, the enclave may terminate itself as a countermeasure.

Preventing **A3** is more involved and requires several steps: before executing a transaction, we demand that (i) both HTs of a CPU core enter the enclave and (ii) remain there. To enforce (ii), the two threads write a unique marker to each thread's *State Save Area* (SSA) [32] inside the enclave. Whenever a thread leaves an enclave asynchronously (e.g., because of an interrupt), its registers are saved in its SSA [32]. Hence, every unexpected exception or interrupt necessarily overwrites our markers in the SSAs. By inspecting the markers, we can thus ensure that neither of the threads was interrupted (and potentially maliciously migrated to a different core by the OS). One thread now enters a Cloak transaction and verifies the two markers, making them part of its read set. Thus, as we confirmed experimentally, any interruption of the threads would overwrite an SSA marker in the read set and cause an immediate transactional abort with code CONFLICT (bit three set in the abort code).

Unfortunately, for (i), there is no direct way for enclave code to tell if two threads are indeed two corresponding HTs. However, after writing the SSA markers, before starting the SSA transaction, the enclave code can initially conduct a series of experiments to check that, with a certain confidence, the two threads indeed share an L1 cache. One way of doing so is to transmit a secret (derived using the `rdrand` instruction inside the enclave) over a timing-less L1-based TSX *covert channel*: for each bit in the secret, the receiver starts a transaction and fills a certain L1 cache set with write-set cache lines and busy-waits within the transaction for a certain time; if the current bit is 1, the sender aborts the receiver's transaction by touching conflicting cache lines of the same cache set. Otherwise, it touches non-conflicting cache lines. After the transmission, both threads compare their versions of the secret. In case bit-errors are below a certain threshold, the two threads are assumed to be corresponding HTs. In our experiments, the covert channel achieved a raw capacity of 1 MB/s at an error rate of 1.6% between two HTs. For non-HTs, the error rate was close to 50% in both cases, showing that no cross-core transmission is possible.[6] While a malicious OS could attempt to eavesdrop on the sender and replay for the receiver to spoil the check, a range of additional countermeasures exists that would mitigate this attack. For example, the two threads could randomly choose a different L1 cache set (out of the 64 available) for each bit to transmit.

To protect against **A4**, the enclave may use SGX's trusted monotonic counters [3] or require an online connection to its owner on restart.

Finally, the enclave may demand a private LLC partition, which could be provided by the OS via Intel's recent *Cache Allocation Technology* (CAT) feature [32] or "cache coloring" [11,37,58]. A contract violation would become evident to the enclave through increased numbers of aborts with code CONFLICT.

[6]Using the read set instead yields a timing-less cross-core covert channel with a raw capacity of 335 KB/s at an error rate of 0.4%.

8 Limitations and Future Work

Cache attacks are just one of many types of side-channel attacks and Cloak naturally does not mitigate all of them. Especially an adversary able to measure the execution time of a transaction might still derive secret information. Beyond this, Cloak instantiated with Intel TSX may be vulnerable to additional side channels that have not yet been explored. We identified five potential side channels that should be investigated in more detail: First, the interaction of the read set and the "second level structure" (*i.e.*, the bloom filter) is not documented. Second, other caches, such as translation-lookaside buffers and branch-prediction tables, may still leak information. Third, the Intel TSX abort codes may provide side-channel information if accessible to an attacker. Fourth, variants of Prime+Probe that deliberately evict read set cache lines from L1 to the LLC but not to DRAM could potentially obtain side-channel information without causing transaction aborts. Fifth, the execution time of transactions including in particular the timings of aborts may leak information. Finally, it is important to note that Cloak is limited by the size of the CPU's caches, since code and data that have secret-dependent accesses must fit in the caches. TSX runtime behavior can also be difficult to predict and control for the programmer.

9 Related Work

Using HTM for Security and Safety. The Mimosa system [25] uses TSX to protect cryptographic keys in the Linux kernel against different forms of memory disclosure attacks. Mimosa builds upon the existing TRESOR system [47], which ensures that a symmetric master key is always kept in the CPU's debug registers. Mimosa extends this protection to an arbitrary number of (asymmetric) keys. Mimosa always only writes protected keys to memory within TSX transactions. It ensures that these keys are wiped before the corresponding transaction commits. This way, the protected keys are never written to RAM. However, Mimosa does not prevent cache side-channel attacks. Instead, for AES computations it uses AES-NI, which does not leak information through the cache. However, a cache attack on the square-and-multiply routine of RSA in the presence of Mimosa would still be possible. To detect hardware faults, the HAFT system [39] inserts redundant instructions into programs and compares their behavior at runtime. HAFT uses TSX to efficiently roll-back state in case a fault was encountered.

Probably closest related to Cloak is the recent T-SGX approach [59]. It employs TSX to protect SGX enclave code against the page-fault side channel [61], which can be exploited by a malicious OS that unmaps an enclave's memory pages (cf. Section 7). At its core, T-SGX leverages the property that exceptions within TSX transactions cause transactional aborts and are not delivered to the OS. T-SGX ensures that virtually all enclave code is executed in transactions. To minimize transactional aborts, e.g., due to cache-line evictions, T-SGX's extension of the Clang compiler automatically splits enclave code into small *execution blocks* according to a static over-approximation of L1 usage. At runtime, a *springboard* dispatches control flow between execution blocks, wrapping each into a separate TSX transaction. Thus, only page faults related to the springboard can be (directly) observed from the outside. All transactional aborts are handled by the springboard, which may terminate the enclave when an attack is suspected. For T-SGX, Shih et al. [59] reported performance overheads of 4%–108% across a range of algorithms and, due to the strategy of splitting code into small execution blocks, caused only very low rates of transactional aborts.

The strategy employed by T-SGX cannot be generally transferred to Cloak, as—for security—one would need to reload the code and data of a sensitive function whenever a new block is executed. Hence, this strategy is not likely to reduce cache conflicts, which is the main reason for transactional aborts in Cloak, but rather increase performance overhead. Like T-SGX, the recent Déjà Vu [8] approach also attempts to detect page-fault side-channel attacks from within SGX enclaves using TSX: an enclave thread emulates an non-interruptible clock through busy waiting within a TSX transaction and periodically updating a counter variable. Other enclave threads use this counter for approximate measuring of their execution timings along certain control-flow paths. In case these timings exceed certain thresholds, an attack is assumed. Both T-SGX and Déjà Vu conceptually do not protect against common cache side-channel attacks.

Prevention of Resource Sharing. One branch of defenses against cache attacks tries to reduce resource sharing in multi-tenant systems. This can either be implemented through hardware modifications [12, 52], or by dynamically separating resources. Shi et al. [58] and Kim et al. [37] propose to use cache coloring to isolate different tenants in cloud environments. Zhang et al. [68] propose cache cleansing as a technique to remove information leakage from time-shared caches. Godfrey et al. [18] propose temporal isolation through scheduling and resource isolation through cache coloring. More recently Zhou et al. [69] propose a more dynamic approach where pages are duplicated when multiple processes access them simultaneously. Their approach can make attacks significantly more difficult to mount, but not impossible. Liu et al. [42] propose to use Intel CAT to split the LLC, avoiding the fundamental resource sharing that is exploited in many attacks. In contrast to Cloak, all these approaches require changes on the OS level.

Detecting Cache Side-Channel Leakage. Other defenses aim at detecting potential side-channel leakage and attacks, e.g., by means of static source code analysis [13] or by performing dynamic anomaly detection using CPU performance counters. Gruss et al. [23] explore the latter approach and devise a variant of Flush+Reload that evades it. Chiappetta et al. [9] combine performance counter-based detection with machine learning to detect yet unknown attacks. Zhang et al. [65] show how performance counters can be used in cloud environments to detect cross-VM side-channel attacks. In contrast, Cloak follows the arguably stronger approach of mitigating attacks before they happen. Many attacks require only a small number of traces or even work with single measurements [17, 43, 62]. Thus, Cloak can provide protection where detection mechanisms fail due to the inherent detection delays or too coarse heuristics. Further, reliable performance counters are not available in SGX enclaves.

10 Conclusions

We presented Cloak, a new technique that defends against cache side-channel attacks using hardware transactional memory. Cloak enables the efficient retrofitting of existing algorithms with strong cache side-channel protection. We demonstrated the efficacy of our approach by running state-of-the-art cache side-channel attacks on existing vulnerable implementations of algorithms. Cloak successfully blocked all attacks in every attack scenario. We investigated the imperfections of Intel TSX and discussed the potentially remaining leakage. Finally, we showed that one of the main limitations of Intel SGX, the lack of side-channel protections, can be overcome by using Cloak inside Intel SGX enclaves.

References

[1] Aciiçmez, O., Gueron, S., and Seifert, J.-p. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding* (2007).

[2] Allan, T., Brumley, B. B., Falkner, K., van de Pol, J., and Yarom, Y. Amplifying side channels through performance degradation. In *Anual Computer Security Applications Conference (ACSAC)* (2016).

[3] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[4] Bernstein, D. J. Cache-timing attacks on AES. Tech. rep., Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.

[5] Bhattacharya, S., and Mukhopadhyay, D. Curious case of Rowhammer: Flipping secret exponent bits using timing analysis. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2016).

[6] Bhattacharya, S., Rebeiro, C., and Mukhopadhyay, D. Hardware prefetchers leak : A revisit of SVF for cache-timing attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2012).

[7] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A.-R. Software Grand Exposure: SGX cache attacks are practical. *arXiv:1702.07521* (2017).

[8] Chen, S., Zhang, X., Reiter, M. K., and Zhang, Y. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2017).

[9] Chiappetta, M., Savas, E., and Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034, 2015.

[10] Costan, V., and Devadas, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086.

[11] Costan, V., Lebedev, I., and Devadas, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016).

[12] Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., and Ponomarev, D. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* (2011).

[13] Doychev, G., Köpf, B., Mauborgne, L., and Reineke, J. CacheAudit: a tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* (2015).

[14] Evtyushkin, D., and Ponomarev, D. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[15] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[16] Ferri, C., Bahar, R. I., Loghi, M., and Poncino, M. Energy-optimal synchronization primitives for single-chip multi-processors. In *ACM Great Lakes Symposium on VLSI* (2009).

[17] 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).

[18] Godfrey, M. M., and Zulkernine, M. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing* (2014).

[19] Goel, B., Titos-Gil, R., Negi, A., McKee, S. A., and Stenstrom, P. Performance and energy analysis of the restricted transactional memory implementation on Haswell. In *International Conference on Parallel Processing (ICPP)* (2014).

[20] Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on intel sgx. In *European Workshop on System Security (EuroSec)* (2017).

[21] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch side-channel attacks: Bypassing SMAP and Kernel ASLR. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[22] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[23] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A fast and stealthy cache attack. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2016).

[24] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium* (2015).

[25] Guan, L., Lin, J., Luo, B., Jing, J., and Wang, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[26] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy (S&P)* (2011).

[27] Herlihy, M., Eliot, J., and Moss, B. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)* (1993).

[28] Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[29] Intel Corp. Software Development Emulator v. 7.49. `https://software.intel.com/en-us/articles/intel-software-development-emulator/` (retrieved 19/01/2017).

[30] Intel Corp. Intel 64 and IA-32 architectures optimization reference manual, June 2016.

[31] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, September 2016.

[32] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, September 2016.

[33] 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 *IEEE Symposium on Security and Privacy (S&P)* (2015).

[34] Irazoqui, G., Eisenbarth, T., and Sunar, B. Cross processor cache attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2016).

[35] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! a fast, cross-VM attack on AES. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2014).

[36] Jang, Y., Lee, S., and Kim, T. Breaking kernel address space layout randomization with intel TSX. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

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

[38] KOCHER, P. C. Timing attacks on implementations of Diffe-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO* (1996).

[39] KUVAISKII, D., FAQEH, R., BHATOTIA, P., FELBER, P., AND FETZER, C. HAFT: hardware-assisted fault tolerance. In *European Conference on Computer Systems (EuroSys)* (2016).

[40] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *arXiv preprint arXiv:1611.06952* (2016).

[41] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache attacks on mobile devices. In *USENIX Security Symposium* (2016).

[42] 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 *International Symposium on High Performance Computer Architecture (HPCA)* (2016).

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

[44] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *International Symposium on High Performance Computer Architecture (HPCA)* (2014).

[45] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[46] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. CacheZoom: How SGX amplifies the power of cache attacks. *arXiv:1703.06986* (2017).

[47] MÜLLER, T., FREILING, F. C., AND DEWALD, A. TRESOR runs encryption securely outside RAM. In *USENIX Security Symposium* (2011).

[48] NAKAIKE, T., ODAIRA, R., GAUDET, M., MICHAEL, M. M., AND TOMARI, H. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *International Symposium on Computer Architecture (ISCA)* (2015).

[49] OHRIMENKO, O., SCHUSTER, F., FOURNET, C., MEHTA, A., NOWOZIN, S., VASWANI, K., AND COSTA, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016).

[50] 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 *ACM Conference on Computer and Communications Security (CCS)* (2015).

[51] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *RSA Conference Cryptographer's Track (CT-RSA)* (2006).

[52] PAGE, D. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280.

[53] PAYER, M. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems (ESSoS)* (2016).

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

[55] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security Symposium* (2016).

[56] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[57] SCHWARZ, M., GRUSS, D., WEISER, S., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to conceal cache attacks. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2017).

[58] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2011).

[59] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[60] SPREITZER, R., AND PLOS, T. Cache-access pattern attack on disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design (COSADE)* (2013).

[61] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[62] YAROM, Y., AND FALKNER, K. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014).

[63] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[64] ZACHAROPOULOS, G. Employing hardware transactional memory in prefetching for energy efficiency. Uppsala Universitet (report), 2015. `http://www.diva-portal.org/smash/get/diva2:847611/FULLTEXT01.pdf` (retrieved 20/02/2017).

[65] ZHANG, T., ZHANG, Y., AND LEE, R. B. CloudRadar: A real-time side-channel attack detection system in clouds. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2016).

[66] ZHANG, X., XIAO, Y., AND ZHANG, Y. Return-oriented flush-reload side channels on ARM and their implications for Android devices. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[67] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[68] ZHANG, Y., AND REITER, M. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[69] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A software approach to defeating side channels in last-level caches. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

Listing 2: Decision tree classification before and after Cloak: the code in black is shared by both versions, the code before Cloak is in dark gray(lines 1–3), and Cloak-specific additions are in blue (lines 5–7, 11, 12, 15).

```
 1  using Nodes = nelem_t*;
 2  using Queries = Matrix<float>;
 3  using LeafIds = uint16_t*;
 4
 5  using Nodes = ReadArray<nelem_t, NCS_R>;
 6  using Queries = ReadMatrix<float, NCS_R>;
 7  using LeafIds = WriteArray<uint16_t, NCS_W>;
 8
 9  void _tsx_protected_lookup_leafids(
10  Nodes& nodes, Queries& queries, LeafIds&
        leafids) {
11    nodes.preload();
12    queries.preload();
13
14    for (size_t q=0; q < queries.entries();
         q++) {
15      if (!(q % 8)) leafids.preload();
16      size_t idx = 0, left, right;
17      for(;;) {
18        auto &node = nodes[idx];
19        left = node.left;
20        right = node.right_or_leafid;
21        if (left == node) {
22          leafids[q] = right;
23          break;
24        }
25        if (queries.item(q, node.fdim) <=
      node.fthresh)
26          idx = left;
27        else
28          idx = right;
29      }
30    }
31  }
```

A Cloak Code Example

Listing 2 gives an example of the original code for tree traversal and its Cloak-protected counterpart. In the original code, a tree is stored in a `Nodes` array where each node contains a feature, `fdim`, and a threshold, `fthres`. Access to a node determines which feature is used to make a split and its threshold on the value of this feature indicates whether the traversal continues left or right. For every record batched in `Queries`, the code traverses the tree according to feature values in the record. Once a leaf is reached its value is written as the output of this query in `LeafIds`. The following features of Cloak are used to protect code and data accesses of the tree traversal. First, it uses Cloak data types to allocate `Nodes` and `Queries` in the read set and `LeafIds` in the write sets. This ensures that data is allocated as described in Section 5.2.1, oblivious from the programmer. The parameters `NCS_R` and `NCS_W` indicate the number of cache sets to be used for the read and write sets. Second, the programmer indicates to the compiler which function should be run within a transaction by using `_tsx_protected` annotation. The programmer calls preload (lines 11, 12, and 15) on sensitive data structures. The repeated preloading of the writable array `leafids` in line 15 refreshes the write set to prevent premature evictions.