Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think

Stephan van Schaik
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Kaveh Razavi
Vrije Universiteit Amsterdam

Abstract

Cache attacks have increasingly gained momentum in the security community. In such attacks, attacker-controlled code sharing the cache with a designated victim can leak confidential data by timing the execution of cache-accessing operations. Much recent work has focused on defenses that enforce cache access isolation between mutually distrusting software components. In such a landscape, many software-based defenses have been popularized, given their appealing portability and scalability guarantees. All such defenses prevent attacker-controlled CPU instructions from accessing a cache partition dedicated to a different security domain.

In this paper, we present a new class of attacks (*indirect cache attacks*), which can bypass all the existing software-based defenses. In such attacks, rather than accessing the cache directly, attacker-controlled code lures an external, trusted component into indirectly accessing the cache partition of the victim and mount a confused-deputy side-channel attack. To demonstrate the viability of these attacks, we focus on the MMU, demonstrating that indirect cache attacks based on translation operations performed by the MMU are practical and can be used to bypass all the existing software-based defenses.

Our results show that the isolation enforced by existing defense techniques is imperfect and that generalizing such techniques to mitigate arbitrary cache attacks is much more challenging than previously assumed.

1 Introduction

Cache attacks are increasingly being used to leak sensitive information from a victim software component (e.g., process) running on commodity CPUs [8, 11, 12, 15, 19, 21, 22, 26, 29, 31, 32, 33, 42]. These attacks learn about the secret operations of a victim component by observing changes in the state of various CPU caches. Since such attacks exploit fundamental hardware properties (i.e., caching), commodity software operating on security-sensitive data is inherently vulnerable. Constant-time software implementations are an exception, but generating them manually is error-prone and automated approaches incur impractical performance costs [34]. In response to these attacks, state-of-the-art defenses use software- or hardware-enforced mechanisms to partition CPU caches between mutually distrusting components.

Given the lack of dedicated hardware support for the mitigation of cache attacks, current hardware-enforced mechanisms re-purpose *other* CPU features, originally intended for *different* applications, to partition the shared caches. For example, Intel CAT, originally designed to enforce quality-of-service between virtual machines [18], can be re-purposed to coarsely partition the shared last level cache [30]. As an another example, Intel TSX, originally designed to support hardware transactional memory, can be re-purposed to pin the working set of a secure transaction inside the cache. By probing the cache partitions used by protected software running in a transaction, attackers will cause transaction aborts that can signal an on-going attack. While effective, these defenses rely on features available only on specific (recent Intel) architectures and, due to their limited original scope, cannot alone scale to provide whole-system protection against cache attacks. For instance, Intel CAT-based defenses can only support limited security partitions or secure pages. In another direction, Intel TSX-based defenses can only protect a limited working set.

In comparison, software-based cache defenses do not suffer from these limitations and in recent years have become increasingly popular. Given the knowledge of how memory is mapped to the CPU caches, these defenses can freely allocate memory in a way that partitions the cache to isolate untrusted software components from one another. This can be done at a fine granularity to guarantee scalability [25, 44], while remaining portable across different architectures. The main question with these defenses, however, is whether they perform this partition-

ing sufficiently well without hardware support.

The answer is no. In this paper we present a new class of attacks, *indirect cache attacks*, which demonstrate that an attacker can mount practical cache attacks by piggybacking on external, trusted components, for instance on existing hardware components. Recent side-channel attacks have already targeted hardware components as *victims*, for instance by side channeling CPU cores [21, 31, 33, 42], memory management units (MMU) [12], transactions [8, 22], or speculative execution functionality [26, 29].

Unlike such attacks, indirect cache attacks abuse hardware components as *confused deputies* to access the cache on the attacker's behalf and leak information from victim software components. We show this strategy bypasses the imperfect partitioning of all state-of-the-art software-based defenses, which implicitly assume hardware components other than the CPU are trusted.

To substantiate our claims, we focus on MMU-based indirect cache attacks and show how such attacks can bypass existing software-based defenses in practical settings. Our focus on the MMU is motivated by (i) the MMU being part of the standard hardware equipment on commodity platforms exposed to side-channel attacks, and (ii) the activity of the MMU being strongly dependent on the operations performed by the CPU, making it an appealing target for practical indirect cache attacks.

In detail, we show how our concrete attack implementation, named XLATE, can program the MMU to replace the CPU as the active actor, mounting attacks such as FLUSH + RELOAD and PRIME + PROBE. Performing XLATE attacks is challenging due to the unknown internal architecture of the MMU, which we explore as part of this paper. XLATE attacks show that the translation structures (i.e., page tables) and any other data structures used by other cache-enabled trusted hardware/software components should be subject to the same partitioning policy as regular code/data pages in existing and future cache defenses. We show that retrofitting this property in existing defenses is already challenging for XLATE attacks, let alone for future, arbitrary indirect cache attacks, which we anticipate can target a variety of other trusted hardware/software components.

Summarizing, we make the following contributions:

- The reverse engineering of the internal architecture of the MMU, including translation and page table caches in a variety of CPU architectures.
- A new class of cache attacks, which we term indirect cache attacks and instantiate for the first time on the MMU. Our XLATE attack implementation can program the MMU to indirectly perform a variety of existing cache attacks in practical settings.
- An evaluation of XLATE attacks, showing how they compromise all known software-based cache defenses, and an analysis of possible mitigations.
- An open-source test-bed for all the existing and new cache attacks considered in this paper, the corresponding covert-channel implementations, and applicable cache defenses, which can serve as a framework to foster future research in the area. The source code and further information about this project can be found here:

  `https://vusec.net/projects/xlate`

The remainder of the paper is organized as follows. Section 3 provides background on existing cache attacks, while Section 4 provides background on existing cache defenses both in hardware and software. Section 5 and Section 6 present the design and implementation of XLATE family of indirect cache attacks. Section 7 compares the XLATE attacks against existing attacks and show that they break state-of-the-art software-based defenses. Finally, Section 8 discusses possible mitigations against these attacks, Section 9 covers related work, and Section 10 concludes the paper.

2 Threat Model

We assume an attacker determined to mount a cache attack such as PRIME + PROBE and leak information from a co-located victim on the same platform. In practical settings, the victim is typically a virtual machine in a multi-tenant cloud or a user process in an unprivileged code-based exploitation scenario. We also assume the attacker shares hardware resources such as the last-level cache (LLC) with the victim. Furthermore, we assume the victim is protected with state-of-the-art software-based defenses against cache attacks, either deployed standalone or complementing existing hardware-based solutions for scalability reasons. In such a setting, the goal of the attacker is to escape from the containing security domain (cache partition) enforced by the software-based defenses and mount a successful cache attack.

3 Cache Side-Channel Attacks

To overcome the performance gap between processors and memory, multiple caches in the processor store recently-accessed memory locations to hide the memory's high latency. While these CPU caches are an important performance optimization deployed universally, they can also be abused by attackers to leak information from a victim process. Recently accessed memory locations by the victim process will be in the cache and

Table 1: An overview of existing cache side-channel attacks.

Name	Same-Core	Cross-Core	Shared Memory	Measurement
EVICT + TIME [27]	✓	✓	✗	time
PRIME + PROBE [21, 31]	✓	✓	✗	time
PRIME + ABORT [8]	✗	✓	✗	TSX
FLUSH + RELOAD [42]	✓	✓	✓	time
FLUSH + FLUSH [16]	✓	✓	✓	time

attackers can probe for this information by observing the state of the caches to leak sensitive information about the secret operation of the victim process. This prevalent class of side-channel attacks is known as cache attacks. We now briefly explain the high-level architecture of CPU caches before discussing how attackers can perform different variants of these cache attacks.

3.1 Cache Architecture

In the Intel Core architecture, there are three levels of CPU caches. The caches closer to the CPU are smaller and faster, and the caches further away are larger and slower. At the first level, there are two caches, L1i and L1d, to store code and data respectively, while the L2 cache unifies code and data. Where these caches are private to each core, all cores share the L3 which is the last-level cache (LLC). One important property of the LLC is that it is inclusive of the lower level caches—data stored in the lower levels is always present in the LLC. Furthermore, because of its size, the LLC is always set-associative, i.e., it is divided into multiple cache sets where part of the physical address is used to index into the corresponding cache set. These two properties are important for state-of-the-art cache attacks on the LLC.

3.2 Existing Attacks

Table 1 illustrates existing cache attacks. Some of the attacks only work if the attacker executes them on the same core that also executes the victim, while others can leak information across cores through the shared LLC. Furthermore, to measure the state of the cache, these attacks rely either on timing memory accesses to detect if they are cached, or on other events such as transaction aborts. We provide further detail about these attacks in the remainder of this section.

EVICT + TIME In an EVICT + TIME attack, the attacker evicts certain cache sets and then measures the execution time of the victim's code to determine whether the victim used a memory location that maps to the evicted cache sets. While EVICT + TIME attacks provide a lower bandwidth than PRIME + PROBE attacks [33], they are effective in high-noise environments such as JavaScript [12].

PRIME + PROBE and PRIME + ABORT In a PRIME + PROBE attack, the attacker builds an eviction set of memory addresses to fill a specific cache set. By repeatedly measuring the time it takes to refill the cache set, the attacker can monitor memory accesses to that cache set. Furthermore, as part of the memory address determines the cache set to which the address maps, the attacker can infer information about the memory address used to access the cache set. Thus, by monitoring different cache sets, an attacker can determine, for example, which part of a look-up table was used by a victim process. While PRIME + PROBE originally targeted the L1 cache [33] to monitor accesses from the same processor core or another hardware thread, the inclusive nature of the LLC in modern Intel processors has led recent work to target the LLC [21, 23, 31], enabling PRIME + PROBE in cross-core and cross-VM setups.

PRIME + ABORT [8] is a variant of PRIME + PROBE that leverages Intel's Transaction Synchronization Extensions (TSX). Intel TSX introduces support for hardware transactions, where the L1 and L3 caches are used as write and read sets, respectively, to keep track of addresses accessed within the transaction. PRIME + ABORT monitors accesses to a single cache set by filling the cache set during a transaction as any additional accesses to same cache set causes the transaction to abort.

FLUSH + RELOAD and FLUSH + FLUSH To reduce the memory footprint, running processes often share identical memory pages. Shared libraries is a prime example of sharing (code) pages. Another example is memory deduplication [32], where an active process searches for pages with identical contents to coalesce them. While there are hardware mechanisms in place to ensure isolation between processes by enforcing read-only or copy-on-write semantics for shared pages, the existence of shared caches results in an exploitable side-channel for such pages. Gullasch et al. [17] use the `CLFLUSH` instruction to evict targets to monitor from the cache. By measuring the time to reload them the attacker determines whether the victim has accessed them—a class of attacks called FLUSH + RELOAD. Further, Yarom and Falkner [42] observe that `CLFLUSH` evicts a memory line from all the cache levels, including the last-level cache (LLC) which is inclusive of the lower cache levels and shared between all processor cores, thus enabling an attacker to monitor a victim from another processor core. In addition, the FLUSH + RELOAD attack

Table 2: Overview of existing cache side-channel defenses.

Name	Same-Core	Cross-Core	Implementation	Strategy
Page Coloring [43]	✓	✓	Software	Sets
CacheBar [44]	✓	✓	Software	Ways
StealthMem [25]	✓	✓	Software	Pinning
Intel CAT [30, 36]	✗	✓	Hardware	Ways
ARM AutoLock [13]	✗	✓	Hardware	Pinning
CATalyst [30]	✗	✓	Hardware	Ways Pinning
Cloak [14]	✓	✓	Hardware	TSX

allows for cross-VM attacks.

A variant of FLUSH + RELOAD, FLUSH + FLUSH [16] builds upon the observation that `CLFLUSH` aborts early in case of a cache miss, leading to a side channel. As the FLUSH + FLUSH attack relies only on the `CLFLUSH` and performs no memory accesses, it is a stealthier alternative to FLUSH + RELOAD.

4 Existing Defenses

As shown in Table 2, the security community developed several defenses both in software and in hardware to mitigate cache side-channel attacks. Given the knowledge of how memory is mapped to the CPU caches, these defenses can freely partition the memory between distrusting processes in a way that partitions the cache, thus preventing the eviction of each other's cache lines. There are three common approaches for achieving this goal: partitioning the cache by sets, partitioning the cache by ways, and locking cache lines such that they cannot be evicted.

4.1 Hardware Defenses

Intel Cache Allocation Technology (CAT) [36] is a hardware mechanism that is available on a select series of Intel Xeon and Atom products. Intel CAT allows the OS or hypervisor to control the allocation of cache ways by assigning a bit mask to a class of service (CLOS). While Intel CAT could be used to assign disjoint bit masks to each security domain, the provided amount of classes of service, and thus security domains, is limited to four or sixteen. Instead, Liu et al. [30] leverage Intel CAT as a defense against LLC side-channel attacks by partitioning the LLC into a secure and a non-secure partition. While applications can freely use the non-secure partition, the secure partition is loaded with cache-pinned secure pages. However, the secure partition is strictly limited in size, limiting the number of secure pages one can support. Similarly, older ARM processors such as the ARM Cortex A9 implement Cache Lockdown [6, 35], which enables software to pin cache lines within the L2 cache by restricting the cache ways that can be allocated.

Another hardware mechanism is ARM AutoLock—originally an inclusion policy designed to reduce power consumption that also happens to prevent cross-core attacks by locking cache lines in the L2 cache when they are present in any of the L1 caches [13, 40]. As a result, to use ARM AutoLock as a defense, sensitive data has to be kept in the L1 caches, which are limited in size.

Intel TSX introduces support for hardware transactions where the L1 and L3 are used as write and read sets, respectively, to keep track of accesses within the transaction. Introduced first on Intel Haswell, Intel initially disabled TSX due to bugs, but it reappeared on Intel Skylake, although in a limited set of products. Cloak [14] leverages Intel TSX to mitigate cache attacks. Intel TSX keeps the working set of a transaction inside the CPU cache sets and aborts if one of the cache sets overflows. Cloak pre-loads sensitive code and data paths into the caches and executes the sensitive code inside a transaction to keep its working set inside the cache sets. If an attacker tries to probe a sensitive cache set, the transaction aborts without leaking whether that cache set was accessed by the protected code. While effective, Cloak requires modification to the application code and is limited to computations whose working set can strictly fit inside CPU caches.

Other than the scalability limitations mentioned above, another concern with hardware-based defenses is their lack of portability. Intel CAT or TSX are only available on a subset of Intel processors and ARM Lockdown only on older ARM processors, hindering their wide-spread deployment.

4.2 Software Defenses

On contemporary processors, the LLC is both set-associative and physically indexed, i.e. part of the physical address determines to which cache set a certain physical memory address maps. While the byte offset within a page determines the least-significant bits of the index, the most-significant bits form the page color. More specifically, a page commonly consists of 64 cache lines that map to 64 consecutive cache sets in the LLC. Thus, pages with a different page color do not map to the same cache sets, a property originally used to improve the overall system performance [3, 24, 43] or the performance of real-time tasks [28] by reducing cache conflicts. Page coloring has been re-purposed to protect against cache side-channel attacks by assigning different colors to different security domains.

StealthMem [25] provides a small amount of colored memory that is guaranteed to not contend in the cache. From this memory, *stealth pages* can be allocated for storing security-sensitive data, such as the S-boxes of AES encryption. To prevent cache side-channel attacks, StealthMem reserves differently colored *stealth pages* for each core and prevents the usage of pages that share the same color or monitors access to such pages by removing access to these pages via page tables. When such accesses are monitored, StealthMem exploits the cache replacement policy to pin *stealth pages* in the LLC.

CacheBar [44] allocates a budget per cache set to each security domain at the granularity of a page size, essentially representing the amount of cache ways that the security domain is allowed to use for each page color. To record the occupancy, CacheBar monitors accesses to cache sets and maintains a queue of pages that are present in the cache set per security domain. To restrict the number of cache ways that are allocated by a security domain, CacheBar actively evicts pages from the cache following an LRU replacement policy.

Note that all these defenses isolate the cache that untrusted, potentially attacker-controlled, code can *directly access*, but do not account for cache partitions the attacker can *indirectly access* by piggybacking on trusted components such as the MMU. As we will show, this provides an attacker with sufficient leeway to mount a successful indirect cache attack.

5 XLATE Attacks

To demonstrate the viability of indirect cache attacks, we focus on an often overlooked trusted hardware component that attacker-controlled code can indirectly control on arbitrary victim platforms: the MMU. As each memory access from the CPU induces a virtual-to-physical address translation for which the MMU has to consult multiple page tables, the MMU tries to keep the results and the intermediate state for recent translations close to itself by interacting with various caches, including the CPU caches. Since the CPU and the MMU share the CPU caches, it is possible to build an eviction set of virtual addresses of which the page table entries map to certain cache sets, allowing one to monitor activities in these cache sets in a similar fashion to PRIME + PROBE.

As the activity of the MMU is trusted, existing software-based defenses do not attempt to isolate page table pages. This makes it possible to abuse the MMU as a confused deputy and mount indirect cache attacks that bypass these defenses. More specifically, the MMU can be used to build eviction sets that map to cache sets outside the current security domain. We refer to this new class of attacks as XLATE attacks and discuss how they leverage the MMU for mounting cache attacks (Section 5.1). We then show how XLATE attacks can be used to bypass the different defense strategies that we discussed earlier (Section 5.2). Implementing XLATE attacks involves addressing a number of challenges (Section 5.3) which we overcome in our concrete implementation of XLATE attacks described in Section 6.

5.1 Leveraging the MMU

Analogous to the EVICT + TIME, PRIME + PROBE and PRIME + ABORT, we now introduce XLATE + TIME, XLATE + PROBE and XLATE + ABORT. There is no generally-applicable counterpart to FLUSH + RELOAD in the XLATE family of attacks. Although prior work has proposed page table deduplication to share identical page tables between processes [9] (enabling MMU-based FLUSH + RELOAD), this feature is not readily accessible on commodity platforms.

All of the XLATE attacks rely on the same building block, namely finding an eviction set of virtual addresses of which the page table entries map to the same cache set. In PRIME + PROBE, we find eviction sets for a target address by allocating a large pool of pages and adding each of the pages to an eviction set until accessing the entire eviction set slows down accessing the target. For XLATE attacks, eviction sets can be found using a similar approach, but by using page tables instead of pages.

In XLATE + TIME, we fill a specific cache set with the page table entries from the eviction set and then measure the victim's execution time to determine if the victim is accessing the same cache set. To avoid having to measure the execution time of the victim, we can mount a XLATE + PROBE attack where the attacker repeatedly measures the time it takes to refill the cache set, using the page table entries of the eviction set, as a memory access to the same cache set causes one of the page table entries to be evicted (resulting in a slowdown). Finally, XLATE + ABORT leverages Intel TSX by filling the cache set with the page table entries of the eviction set within a hardware transaction. After filling the cache set, the attacker waits for a short period of time for the victim to execute. If the victim has not accessed a memory address that maps to the same cache set, the transaction is likely to commit, otherwise it is likely to abort.

5.2 Bypassing Software-based Defenses

As discussed in Section 4, existing software-based cache defenses partition the LLC either by cache ways or sets [43, 44], or by pinning specific cache lines to the LLC [25]. As mentioned, all these defenses focus on isolating untrusted components such as code running in a virtual machine, but allow unrestricted access to the cache to trusted operations—such as the page table walk

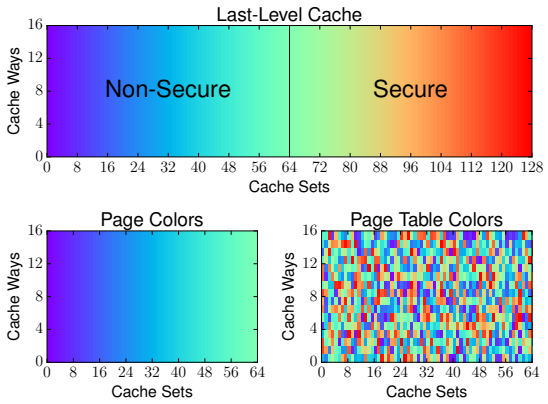

Figure 1: The top shows the LLC being divided into 128 unique page colors, the bottom left shows how the LLC can be partitioned such that programs can only access a subset of these page colors, the bottom right shows the situation for their respective page tables.

performed by the MMU. The implications can be seen in Figure 1, which shows an example of page coloring to partition the LLC. Even though the cache lines of the pages themselves are limited to a specific subset of page colors, and thus a specific subset of cache sets, their respective page tables are able to access all page colors.

Similarly, software implementations that restrict the amount of ways that can be occupied by untrusted applications for each cache set, such as CacheBar [44], typically use the page fault handler for this purpose. However, as the page fault handler is only able to monitor accesses to pages from the CPU, accesses to page tables by the MMU go unnoticed. Therefore, the MMU is not restricted by this limitation and is free to allocate all the ways available in each cache set. To implement cache pinning, STEALTHMEM also uses the page fault handler for the specific cache sets that may be used to host sensitive data in order to reload those cache lines upon every access. As the page table accesses by the MMU are not monitored by the page fault handler, accesses to page tables that map to the same cache set as the sensitive data, do not reload those cache lines.

5.3 Summary of Challenges

There are three main challenges that we must overcome for implementing successful XLATE attacks:

1. Understanding which caches the MMU uses, how it uses them, and how to program the MMU to load page table entries in the LLC.

2. Finding an eviction set of pages of which their page tables map to the same cache set as our target. These eviction sets should target page colors outside the security domain enforced by existing defenses.

3. Similar to existing cache attacks, XLATE attacks are subject to noise. Worse, due to their indirect nature, addressing the sources of noise is more challenging. We need to overcome this noise for an effective implementation of XLATE.

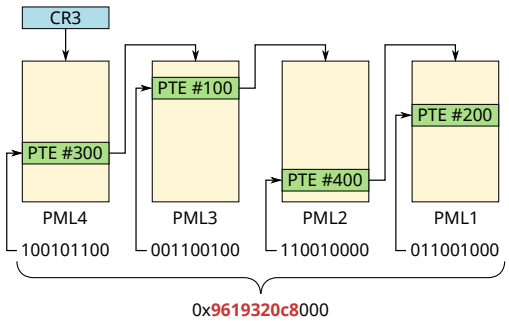

Figure 2: MMU's page table walk to translate `0x9619320c8000` to its corresponding memory page on the `x86_64` architecture.

Next we discuss how we overcome these challenges in our implementation of XLATE attacks.

6 Implementing XLATE Attacks

Before we can use the MMU to mount XLATE attacks, we need to fully understand how the MMU performs a page table walk when translating virtual addresses into their physical counterparts. Even though it is already known that the MMU uses the TLB and the CPU caches as part of its translation process [12], there are also other caches (e.g., translation caches [1]) with mostly an unknown architecture. We need to reverse engineer their architecture before we can ensure that our virtual address translations end up using the CPU caches where our victim data is stored. We reverse engineer these properties in Section 6.1. In Section 6.2, we show how we retrofit an existing algorithm for building PRIME + PROBE eviction sets to instead build suitable eviction sets for XLATE attacks. We further show how XLATE can blindly build eviction sets for security domains to which it does not have access. Finally, in Section 6.3, we identify different sources of noise and explain how to mount a noise-free XLATE attack.

6.1 Reverse Engineering the MMU

The MMU is a hardware component available in many modern processor architectures, that is responsible for

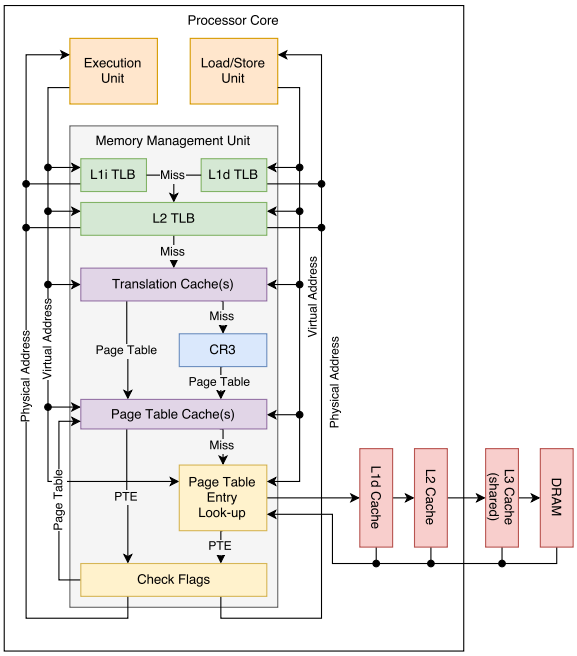

Figure 3: A generic implementation of an MMU and all the components involved to translate a virtual address into a physical address.

the translation of virtual addresses to their corresponding physical address. These translations are stored in page tables–a directed tree of multiple levels, each of which is indexed by part of the virtual address to select the next level page tables, or at the leaves, the physical page. Hence, every virtual address uniquely selects a path from the root of this tree to the leaf to find the corresponding physical address. Figure 2 shows a more concrete example of how the MMU performs virtual address translation on `x86_64`. First, the MMU reads the `CR3` register to find the physical address of the top-level page table. Then, the top nine bits of the virtual address index into this page table to select the page table entry (PTE). This PTE contains a reference to the next-level page table, which the next nine bits of the virtual address index to select the PTE. By repeating this operation, the MMU eventually finds the corresponding physical page for `0x644b321f4000` at the lowest-level page table.

The performance of memory accesses improves greatly if the MMU can avoid having to resolve a virtual address that it already resolved recently. Hence, the MMU stores resolved address mappings in a fast Translation Lookaside Buffer (TLB). To further improve the performance of a TLB miss, the PTEs for the different page table levels are not only stored in the CPU caches, but modern processors also store these in *page table caches* or *translation caches* [1]. While *page table caches* simply store PTEs together with their corresponding physical address and offset, *translation caches* store partially resolved virtual addresses instead. With *translation caches*, the MMU can look up the virtual address and select the entry with the longest matching prefix to skip the upper levels of the page table hierarchy. Figure 3 visualizes how different caches interact when the MMU translates a virtual address.

We rely on the fact that the MMU's page table walk ends up in the target processor's data caches to learn about translation caches. More specifically, the TLB can only host a limited number of virtual address translations. Therefore, if we access at least that many pages, we can evict the TLB, and consequently enforce the MMU to perform a page table walk. We now fix our target address in such a way that we know the cache sets that host the PTEs for that virtual address. We then mount an EVICT + TIME attack for each of the page table levels, where we evict the TLB and the cache set that we expect to host the PTE for that level. Then we measure the time it takes for the MMU to resolve the address to determine if the page table walk loads the PTE in the expected cache set. If the translation caches are not flushed, then the page table walk skips part of the page table hierarchy and simply starts from a lower level page table. As a result the page table walk does not load the PTEs for the higher level page tables to their respective cache sets. Therefore, we now have a basic mechanism to detect whether we properly flushed the translations caches. While the sizes of the TLB and the CPU caches are already known, the sizes of the translation caches are not.

We can use the aforementioned mechanism to reverse engineer the size of translation caches. For instance, a second-level page table maps 2 MiB worth of virtual memory. Thus, if we access any page within that 2 MiB region, the page table walk loads the corresponding PTE pointing to the second-level page table to the translation cache. Similar to TLBs, the number of entries in such a translation cache is limited. Therefore, if we access at least that many 2 MiB regions, we can flush the corresponding translation cache. We use the aforementioned algorithm to tell us whether we the amount of 2 MiB regions is sufficient to flush the translation cache, and thus we know the size of the corresponding translation cache. Finally, we proceed using this algorithm to find the sizes of the translation caches for all the page table levels.

6.2 Building Eviction Sets with the MMU

To build eviction sets for XLATE attacks, we draw from traditional eviction set building algorithms described in the literature for PRIME + PROBE (and derivatives) as shown in Algorithm 1. We first identify the page colors available to our security domain by building eviction sets using PRIME + PROBE. More specifically, we first find eviction sets for the available subset of page colors:

Algorithm 1: Algorithm to build eviction sets dynamically for either a given or a randomly chosen target.

```
Input: a set of potentially conflicting cache lines pool, all
       set-aligned, and an optional target to find an
       eviction set for.
Output: the target and the eviction set for that target
working set ← {};
if target is not set then
    target ← choose(pool);
    remove(pool, target);
end
while pool is not empty do
    repeat
        member ← choose(pool);
        remove(pool, member);
        append(working set, member);
    until evicts(working set, target);
    foreach member in working set do
        remove(working set, member);
        if evicts(working set, target) then
            append(pool, member);
        else
            append(working set, member);
        end
    end
    foreach member in pool do
        if evicts(working set, member) then
            remove(pool, member);
        end
    end
end
return target, working set
```

① We allocate a sufficiently large pool of pages to build these eviction sets. ② We pick random pages from this pool of pages and add them to the eviction set until it is able to evict one of the remaining pages in the pool, the target of our eviction set. ③ We optimize the eviction set by removing pages that do not speed up the access to the target after accessing the eviction set. Upon finding the eviction set, the other pages in the pool are colored using this eviction set and we repeat the process until all the pages have been colored, yielding eviction sets for all the available colors in our security domain. If the amount of page colors is restricted, this results in fewer eviction sets, whereas if the amount of cache ways is restricted, these eviction sets consist of fewer entries.

Using page tables Now we retrofit this algorithm to use the MMU to evict a given page, the target of our choice. More specifically, we build eviction sets of page tables that evict the target page. Instead of allocating pages, we will map the same shared page to multiple locations to allocate unique page tables. Then we apply the same algorithm as before: ① We allocate a sufficiently large pool of page tables to build these eviction sets. ② We pick random page tables (by selecting their corresponding virtual addresses) from this pool of page tables and add them to the eviction set until it is able to evict the target page. ③ We optimize the eviction set by removing page tables that do not speed up the access to the target after accessing the eviction set. Upon finding the eviction set, the other page tables in the pool are colored using this eviction set. We can then repeat this for other pages until all the page tables have been colored, yielding eviction sets for all the available colors in our security domain.

Defeating way partitioning To defeat software-based cache defenses using way partitioning, we now try to find eviction sets that cover the whole cache set. First, we build eviction set of normal pages to find all the available page colors. Then for each of the eviction sets, we build an eviction set of page tables that evicts any page in the eviction set. Since these eviction sets of page tables map to the full cache sets, they bypass way partitioning.

Defeating set partitioning In case of StealthMem and cache defenses using set partitioning, or more specifically, page coloring, we end up with a pool of the remaining page tables that could not be colored. To find the remaining eviction sets, we apply the same algorithm as before to the remaining page tables. This time, however, we choose a random page table from the pool of page tables to use as the target for our algorithm. Ultimately, we end up with the eviction sets for all the remaining page colors. Therefore we are able to bypass cache defenses that use page coloring.

6.3 Minimizing Noise in XLATE Attacks

To mount XLATE attacks, we are interested in finding an eviction set for our target, of which the PTEs for each of the pages in the eviction set map to the same cache set as our target. However, as we are trying to perform an indirect cache attack from the MMU, there are various source of noise that potentially influence our attack. To minimize the noise for XLATE attacks, we rely on the following: (1) translation caches, (2) pointer chasing, (3) re-using physical pages, (4) and transactions.

Translation caches Now that we have reverse engineered the properties of the MMU, we can control which PTEs hit the LLC when performing a page table walk. To improve the performance and to reduce the amount of noise, we are only interested in loading the page tables closer to the leaves into the LLC. Thus, we want to only flush the TLB, while we preserve the translation caches. Algorithm 2 extends PRIME + PROBE to flush

Algorithm 2: XLATE + PROBE method for determining whether an eviction sets evicts a given cache line.

Input: the eviction set `eviction set` and the target `target`.
Output: true if the eviction set evicted the target, false otherwise.

```
timings ← {};
repeat
    access(target);
    map(access, TLB set);
    map(access, eviction set);
    map(access, reverse(eviction set));
    map(access, eviction set);
    map(access, reverse(eviction set));
    append(timings, time(access(target)));
until length(timings) = 16;
return true if median(timings) ≥ threshold else false
```

the TLB using the technique described in Section 6.1. To preserve the translation caches, we reduce the number of 2 MiB region accesses by keeping the pages in the TLB eviction set (i.e., `TLBSet`) sequential. This guarantees that an eviction set of PTEs can evict the target from the LLC.

Pointer chasing Hardware prefetchers in modern processors often try to predict the access pattern of programs to preload data into the cache ahead of time. To prevent prefetching from introducing noise, the eviction set is either shuffled before each call to XLATE + PROBE or a technique called *pointer chasing* is used to traverse the eviction set, where we build an intrusive linked list within the cache line of each page. Because the prefetcher repeatedly mispredicts the next cache line to load, it is disabled completely not to hamper the performance. To defeat adaptive cache replacement policies that learn from cache line re-use, we access the eviction set back and forth twice as shown in Algorithm 2.

Re-using physical pages To perform a page table walk, we have to perform a memory access. Unfortunately, the *page* and its corresponding *page table pages* could have different colors. Therefore, we want to craft our XLATE attack in a way that only page table can evict the target page. For this reason we propose three different techniques to make sure that *only* the cache lines storing the PTEs are able to evict our target's cache line. First, we can exploit page coloring to ensure that the pages pointed to by page tables in the eviction set do not share the same page color as the target page. This way, only the page table pages can evict the target page. Second, by carefully selecting the virtual addresses of the pages in our eviction set, we can ensure that the cache lines of these pages do not align with the cache line of the target page. Therefore, by only aligning the cache line of the corresponding page tables, we can ensure that only the page tables can influence the target page. Third, we allocate a single page of shared memory and map it to different locations in order to allocate many different page tables that point to the exact same physical page. Since we only have one physical page mapped to multiple locations, only the page tables are able to evict the cache line of the target page. In our implementation, we use the third technique, as it shows the best results.

Transactions In XLATE + ABORT, we leverage Intel TSX in a similar fashion to PRIME + ABORT. We observe that page table walks performed by the MMU during a hardware transaction lead to an increase in conflict events when the victim is also using the same cache set. Therefore, we can simply measure the amount of conflict events and check whether this exceeds a certain threshold.

7 Evaluation

We evaluate XLATE on a workstation featuring an Intel Core i7-6700K @ 4.00GHz (Skylake) and 16 GB of RAM. We also consider other evaluation platforms for reverse engineering purposes. To compare our XLATE attack variants against all the state-of-the-art cache attacks, we also implemented FLUSH + RELOAD, FLUSH + FLUSH, EVICT + TIME, PRIME + PROBE, and PRIME + ABORT and evaluated them on the same evaluation platform. We provide representative results from these attacks in this section and refer the interested reader to more extended results in Appendix A.

Our evaluation answers four key questions: (i) *Reverse engineering*: Can we effectively reverse engineer translation caches on commodity microarchitectures to mount practical XLATE attacks? (ii) *Reliability*: How reliable are XLATE channels compared to state-of-the-art cache attacks? (iii) *Effectiveness*: How effective are XLATE attacks in leaking secrets, cryptographic keys in particular, in real-world application scenarios? (iv) *Cache defenses*: Can XLATE attacks successfully bypass state-of-the-art software-based cache defenses?

7.1 Reverse Engineering

Table 3 presents our reverse engineering results for the translation caches of 26 different contemporary microarchitectures. Our analysis in this section extends the results we presented in a short paper at a recent workshop [39]. On Intel, we found that Intel's Page-Structure Caches or split translation caches are implemented by Intel Core and Xeon processors since at least the Nehalem microarchitecture. On Intel Core and Xeon pro-

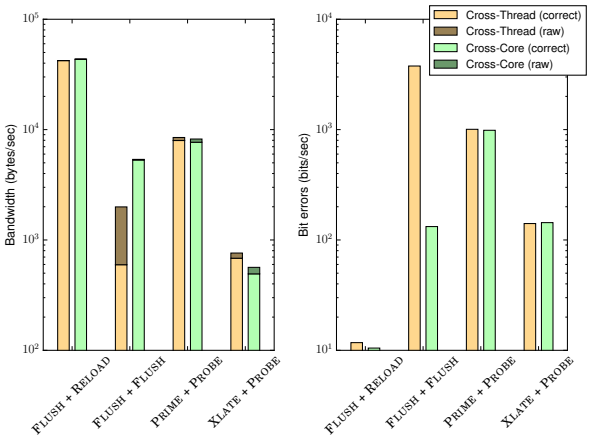

Figure 4: Reliability comparison of different cache side-channel attacks using a reference covert channel implementation on both cross-thread and cross-core setups.

cessors, we also found translation caches available for 32 Page Directory Entries (PDEs) and 4 Page Directory Pointer Table Entries (PDPTEs). In contrast, Intel Silvermont has only a single translation cache for 16 PDEs. On AMD, we found that AMD K10 employs a 24-entry dedicated and unified page table cache and AMD Bobcat employs an 8 to 12 entries variant, respectively. Since AMD Bulldozer, the L2 TLB has been re-purposed to also host page table entries, allowing it to store up to 1024 PDEs on AMD Bulldozer and Piledriver and up to 1536 PDEs on AMD Zen. We also found that AMD Zen introduces another L2 TLB with 64 entries dedicated to 1G pages, allowing it to store up to 64 PDPTEs. On ARM, we found that the low-power variants implement unified page table caches with 64 entries. In contrast, we found that performance-oriented variants implement a translation cache with 16 PDEs on `ARMv7-A` and one with 6 PDPTEs on `ARMv8-A`. Overall, our results show that *translation caches take very different and complex forms across contemporary microarchitectures. As such, our reverse engineering efforts are both crucial and effective for devising practical MMU-based attacks and defenses.*

7.2 Reliability

To evaluate the reliability of XLATE and compare against that of state-of-the-art cache attacks, we implemented an LLC-based covert channel framework, where the sender and the receiver assume the roles of the victim and the attacker respectively. The receiver mounts one of the cache attacks to monitor specific cache lines, while the sender accesses the cache line to transmit a one and does nothing to send a zero otherwise. In order to receive acknowledgements for each word sent, the sender monitors a different set of cache lines. For our implementation, we built a bidirectional channel that is able to transfer 19-bit words at a time. To synchronize both the sender and the receiver, we dedicated 6 bits of the 19-bit word to sequence numbers. Furthermore, we use 4-bit Berger codes to detect simple errors and to prevent zero from being a legal value in our protocol, as it could be introduced by tasks being interrupted by the scheduler. We used our framework to compare the raw bandwidth, the (correct) bandwidth, and the bit error rate between hardware threads on the same CPU core and between different CPU cores. Figure 4 presents our results.

Our results show that FLUSH + RELOAD was able to achieve a bandwidth of around 40 KiB/s with the least noise. PRIME + PROBE performs slightly worse, with a bandwidth of about 8 KiB/s. While FLUSH + FLUSH performs quite well on the cross-core setup with a bandwidth of about 4 KiB/s, it performs much worse on the cross-thread setup with a bandwidth of a mere 500 bytes/s. This is due to the timing difference of flushing a cache line depending on the cache slice hosting it. Compared to the other covert channels, XLATE + PROBE only reaches a bandwidth of 900 bytes/s. While this is slower than other covert channels, the low error rate indicates this is only due to the higher latency of indirect MMU-mediated memory accesses, rather than noisier conditions. This experiment demonstrates XLATE provides a reliable channel and can hence be used to mount side-channel attacks in practical settings as we show next.

7.3 Effectiveness

To evaluate the effectiveness of XLATE, we mounted a side-channel attack against a real-world security-sensitive application. To compare our results against state-of-the-art cache attacks, we focus our attack on the OpenSSL's T-table implementation of AES, using OpenSSL 1.0.1e as a reference. This attack scenario has been extensively used to compare the performance of cache side-channel attacks in prior work (e.g., recently in [8]).

The implementation of AES in our version of OpenSSL uses T-tables to compute the cipher text based on the secret key k and plain text p. During the first round of the algorithm, table accesses are made to entries $T_j[p_i \oplus k_i]$ with $i \equiv j \bmod 4$ and $0 \leq i < 16$. As these T-tables typically map to 16 different cache lines, we can use a cache attack to determine which cache line has been accessed during this round. Note that in case p_i is known, this information allows an attacker to derive $p_i \oplus k_i$, and thus, possible key-byte values for k_i.

More specifically, by choosing p_i and using new random plain text bytes for p_j, where $i \neq j$, while triggering

Table 3: Our reverse engineering results for the translation caches of 26 different microarchitectures.

CPU	Year	Caches			TLBs			Translation Caches			Time
		L1d	L2	L3	4K pages	2M pages	1G pages	PML2E	PML3E	PML4E	
Intel Core i7-7500U (Kaby Lake) @ 2.70GHz	2016	32K	256K	4M	1600	32	20	24-32	3-4	0	5m49s
Intel Core m3-6Y30 (Skylake) @ 0.90GHz	2015	32K	256K	4M	1600	32	20	24	3-4	0	6m01s
Intel Xeon E3-1240 v5 (Skylake) @ 3.50GHz	2015	32K	256K	8M	1600	32	20	24	3-4	0	3m08s
Intel Core i7-6700K (Skylake) @ 4.00GHz	2015	32K	256K	8M	1600	32	20	24	3-4	0	3m41s
Intel Celeron N2840 (Silvermont) @ 2.16GHz	2014	24K	1M	N/A	128	16	N/A	12-16	0	0	52s
Intel Core i7-4500U (Haswell) @ 1.80GHz	2013	32K	256K	4M	1088	32	4	24	3-4	0	2m53
Intel Core i7-3632QM (Ivy Bridge) @ 2.20GHz	2012	32K	256K	6M	576	32	4	24-32	3	0	3m05s
Intel Core i7-2620QM (Sandy Bridge) @ 2.00GHz	2011	32K	256K	6M	576	32	4	24	2-4	0	3m11s
Intel Core i5 M480 (Westmere) @ 2.67GHz	2010	32K	256K	3M	576	32	N/A	24-32	2-6	0	2m44s
Intel Core i7 920 (Nehalem) @ 2.67GHz	2008	32K	256K	8M	576	32	N/A	24-32	3	0	4m26s
AMD Ryzen 7 1700 8-Core (Zen) @ 3.3GHz	2017	32K	512K	16M	1600	1600 [1]	64	0	64	0	13m16s
AMD Ryzen 5 1600X 6-Core (Zen) @ 3.6GHz	2017	32K	512K	16M	1600	1600 [1]	64	0	64	16	30m50s
AMD FX-8350 8-Core (Piledriver) @ 4.0GHz	2012	64K	2M	8M	1088	1088 [2]	1088 [2]	0	0	0	2m50s
AMD FX-8320 8-Core (Piledriver) @ 3.5GHz	2012	64K	2M	8M	1088	1088 [2]	1088 [2]	0	0	0	2m47s
AMD FX-8120 8-Core (Bulldozer) @ 3.4GHz	2011	16K	2M	8M	1056	1056 [2]	1056 [2]	0	0	0	2m33s
AMD Athlon II 640 X4 (K10) @ 3.0GHz	2010	64K	512K	N/A	560	176	N/A	24	0	0	7m50s
AMD E-350 (Bobcat) @ 1.6GHz	2010	32K	512K	N/A	552	8-12	N/A	8-12	0	0	5m38s
AMD Phenom 9550 4-Core (K10) @ 2.2GHz	2008	64K	512K	2M	560	176	48	24	0	0	6m52s
Rockchip RK3399 (ARM Cortex A72) @ 2.0GHz	2017	32K	1M	N/A	544	512 [1]	N/A	16	6	N/A	17m49s
Rockchip RK3399 (ARM Cortex A53) @ 1.4GHz	2017	32K	512K	N/A	522	512 [1]	N/A	64	0	N/A	7m06s
Allwinner A64 (ARM Cortex A53) @ 1.2GHz	2016	32K	512K	N/A	522	512 [1]	N/A	64	0	N/A	52m26s
Samsung Exynos 5800 (ARM Cortex A15) @ 2.1GHz	2014	32K	2M	N/A	544	512 [1,3]	N/A	16	0	N/A	13m28s
Nvidia Tegra K1 CD580M-A1 (ARM Cortex A15) @ 2.3GHz	2014	32K	2M	N/A	544	512 [1,3]	N/A	16	0	N/A	24m19s
Nvidia Tegra K1 CD570M-A1 (ARM Cortex A15; LPAE) @ 2.1GHz	2014	32K	2M	N/A	544	512 [1,3]	N/A	16	0	N/A	6m35s
Samsung Exynos 5800 (ARM Cortex A7) @ 1.3GHz	2014	32K	512K	N/A	266	256 [1,3]	N/A	64	0	N/A	17m42s
Samsung Exynos 5250 (ARM Cortex A15) @ 1.7GHz	2012	32K	1M	N/A	544	512 [1,3]	N/A	16	0	N/A	6m46s

[1] 4K and 2M pages are shared by the L2 TLB. [2] 4K, 2M and 1G pages are shared by the L2 TLB. [3] The TLB is used to store 1M pages on ARMv7-A.

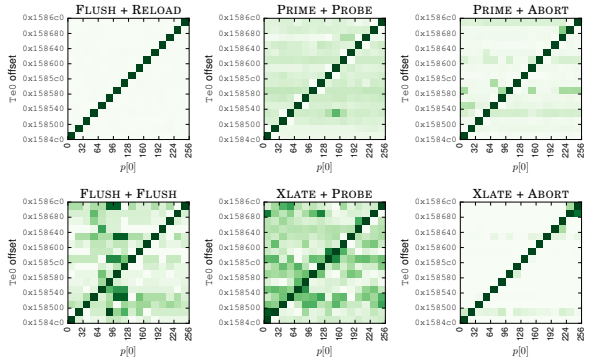

Figure 5: Effectiveness comparison of different cache side-channel attacks using the OpenSSL's T-table implementation of AES (16,000,000 encryption rounds per cache line in `Te0`).

encryptions, an attacker can find which p_i remains to always cause a cache hit for the first cache line in a T-table. By extending this attack to cover all 16 cache lines of the T-table, an attacker can derive the four upper bits for each byte in secret k, thus revealing 64 bits of the secret key k. This is sufficient to compare XLATE against state-of-the-art cache attacks.

For this purpose, we ran a total of 16,000,000 encryptions for each of the cache lines of `Te0` and captured the signal for each cache attack variant. Figure 5 shows that all the cache attacks we considered, including XLATE + PROBE and XLATE + ABORT, are able to effectively retrieve the signal. Moreover, Table 4 shows the end-to-end attack execution times, which strongly correlate with the bandwidth of our covert channels. This experiment shows that XLATE attacks can effectively complete in just seconds, confirming they are a realistic threat against

Table 4: Execution time for various cache side-channel attacks when performing 16,000,000 encryption rounds in OpenSSL.

Name	Time	Success Rate
FLUSH + RELOAD	6.5s	100.0%
FLUSH + FLUSH	10.0s	78.8%
PRIME + PROBE	11.9s	91.7%
PRIME + ABORT	11.3s	100.0%
XLATE + PROBE	66.6s	80.0%
XLATE + ABORT	60.0s	90.2%

real-world production applications.

7.4 Cache Defenses

To evaluate the ability of XLATE attacks to bypass state-of-the-art software-based defenses, we perform the same experiment as in Section 7.3 but now in presence of state-of-the-art software-based cache defenses. For this purpose, we consider the different cache defense strategies discussed in Section 4 and evaluate how PRIME + PROBE and XLATE + PROBE fare against them.

For this experiment, we simulate a scenario where the attacker and the victim run in their own isolated security domains using page coloring and way partitioning. The attacker has access to only 8 ways of each cache set. Since StealthMem uses dedicated cache sets to pin cache lines, this defense is already subsumed by page coloring.

Without additional assumptions, PRIME + PROBE would trivially fail in this scenario, since the preliminary eviction set building step would never complete due to the cache set and ways restrictions. For a more interesting comparison, we instead assume a much stronger attacker with an oracle to build arbitrary eviction sets. To simulate such a scenario, we first allow the attacker

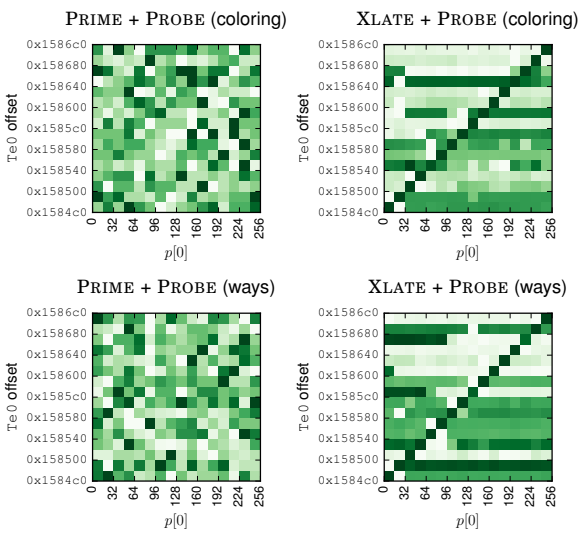

Figure 6: PRIME + PROBE and XLATE + PROBE against the OpenSSL's T-table implementation of AES in presence of state-of-the-art software-based cache defenses.

to dynamically build the eviction set for the target in the victim and then we restrict the eviction set to meet the constraints of the cache defenses considered. Figure 6 presents our results. As shown in the figure, both page coloring and way partitioning disrupt any signal to mount (even oracle-based) PRIME + PROBE attacks, given that the eviction set is prevented from sharing cache sets or ways (respectively) with the victim. In contrast, XLATE + PROBE's MMU traffic is not subject to any of these restrictions and the clear signal in Figure 6 confirms XLATE attacks can be used to bypass state-of-the-art software-based defenses in real-world settings.

8 Mitigations

Even though existing software-based cache defenses are effective against existing side-channel attacks such as PRIME + PROBE and PRIME + ABORT, they are not effective against the XLATE family of attacks. We now investigate how to generalize existing software-based defenses to mitigate XLATE attacks and indirect cache attacks in general. Our analysis shows that, while some software-based defenses can be generalized to mitigate XLATE attacks, most defenses are fundamentally limited against this threat. In addition, countering future, arbitrary indirect cache attacks remains an open challenge for all existing defenses.

8.1 Mitigating XLATE Attacks

As discussed in Section 4, there are three different strategies to mitigate cache attacks, each with their own

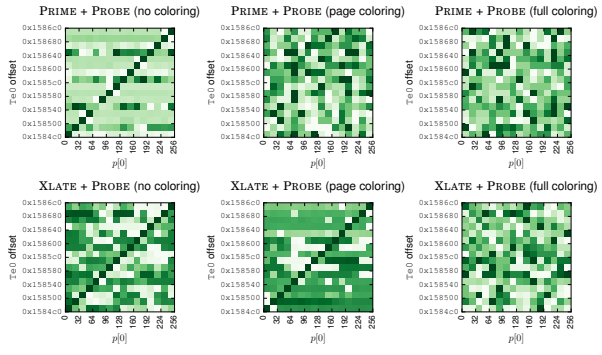

Figure 7: PRIME + PROBE and XLATE + PROBE against OpenSSL's AES T-table implementation on our evaluation platform before and after the mitigation of coloring page tables.

software-based implementation. We now reconsider each software-based defense and discuss possible mitigations against XLATE attacks.

We first reconsider page coloring [43], a software-based defense that relies on the mapping of memory pages to different cache sets to restrict the amount of page colors available to a security domain. In order to harden page coloring against the XLATE family of attacks, its design has to be extended to also color the page tables. By applying the same subset of page colors to both pages and page table pages on a per-domain basis, it is impossible for an attacker to control page table pages outside the assigned security domain.

We show that extending page coloring to also color the page tables is effective by extending the experiment presented in Section 7.4. For each attack on OpenSSL, we compared the PRIME + PROBE and XLATE + PROBE signals for the baseline, after applying traditional page coloring, and after applying both page and page table coloring (*full coloring*). Figure 7 presents our results, showing that, unlike traditional page coloring, full coloring is effective in mitigating XLATE.

The second defense strategy we consider is the cache way partitioning scheme implemented by CacheBar [44]. By monitoring page faults, CacheBar tracks the occupancy of each cache set and, once an application is about to exceed the provided budget, it evicts the least-recently used page and re-enables page fault-based monitoring. This strategy imposes a hard limit to the number of entries used for each cache set. In order to harden this scheme against the XLATE family of attacks, its design needs to be extended to monitor MMU-operated page table accesses. Unfortunately, monitoring such events is impractical as it cannot be done via page faults or other efficient software-based mechanisms, thus leaving this scheme vulnerable to our attacks.

The third and final defense strategy we consider is the cache pinning scheme implemented by StealthMem [25].

StealthMem dedicates specific cache sets to host secret memory pages that should be protected from cache attacks. More specifically, StealthMem pins these memory pages to their respective cache sets by monitoring page faults for pages that map to the same cache set. When a page fault occurs, StealthMem simply reload the corresponding secure pages to preserve cache pinning. In order to harden this scheme against the XLATE family of attacks, we again need to monitor MMU accesses to the page table pages. As mentioned earlier, this is impractical, leaving this scheme vulnerable to our attacks.

Alternatively, the XLATE family of attacks can be stopped in hardware by not sharing the data caches between the CPU and the MMU. While this strategy is effective, it also negates the advantages of software-based defenses, essentially implementing strong isolation provided by hardware-based cache defenses.

8.2 Mitigating Indirect Cache Attacks

While it is possible to mitigate some of the software-based cache defenses against the XLATE family of attacks, the MMU is hardly the only component that can be used as a confused deputy in indirect cache attacks. In fact, there are numerous components both in software and hardware, such as the kernel and integrated GPUs [10] to give a few examples, that could be leveraged for indirect cache attacks as well. More specifically, any component that interacts with the CPU caches and that an attacker can get control over could be leveraged to perform indirect cache attacks. Against such attacks, existing defenses that assume only CPU-based cache accesses (which can be intercepted via page faults), such as CacheBar and StealthMem, are structurally powerless in the general case. Page coloring is more promising, but the challenge is coloring all the possible pages that can be *indirectly* used by a given security domain with the corresponding color. Given the increasing number of software and integrated hardware components on commodity platforms, it is hard to pinpoint the full set of candidates and their interactions. At first glance, bypassing this challenge and coloring all the "special pages" such as page table pages with a reserved "special color" may seem plausible, but the issue is that the attacker can then mount indirect cache attacks against the special pages of the victim (e.g., MMU-to-MMU attacks) to leak information. Even more troublesome is the scenario of trusted components managing explicitly (e.g., kernel buffers) or implicitly (e.g., deduplicated page tables [9]) shared pages across security domains, whose access can be indirectly controlled by an attacker. Coloring alone cannot help here and, even assuming one can pinpoint all such scenarios, supporting a zero-sharing solution amenable to coloring may have deep implications for systems design and raise new performance-security challenges and trade-offs. In short, there is no simple software fix and this is an open challenge for future research.

We conclude by noting that addressing this challenge is non-trivial for hardware-based solutions as well. For instance, the published implementation of CATalyst [30] explicitly moves page table pages mapping secure pages out of the secure domain, which, can, for instance, open the door to MMU-to-MMU attacks. A quick fix is to keep secure page table pages in the secure domain, but this would further reduce CATalyst's number of supported secure pages (and hence scalability) by a worst-case factor of 5 on a 4-level page table architecture.

9 Related Work

We have already covered literature on cache attacks and defenses in Sections 3 and 4. Here we instead focus on related work that use side-channel attacks in the context of Intel SGX or ASLR.

9.1 Intel SGX

Intel Security Guard eXtensions (SGX) is a feature available on recent Intel processors since Skylake, which offers protected enclaves isolated from the remainder of the system. The latter includes the privileged OS and the hypervisor, allowing for the execution of security-sensitive application logic on top of an untrusted run-time software environment. However, when a page fault occurs during enclave execution, the control is handed over to the untrusted OS, revealing the base address of the faulting page. This property can be exploited in a controlled-channel (page fault) attack, whereby a malicious OS can clear the present bit in the Page Table Entries (PTEs) of a victim enclave, obtain a page-level execution trace of the victim, and leak information [41].

Many defenses have been proposed to counter controlled-channel attacks. Shih et al. [37] observe that code running in a transaction using Intel TSX immediately returns to a user-level abort handler whenever a page fault occurs instead of notifying a (potentially malicious) OS. With their T-SGX compiler, each basic block is wrapped in a transaction guaranteed to trap to a carefully designed springboard page at each attack attempt. Chen et al. [5] extend such design not to only hide page faults, but to also monitor suspicious interrupt rates. Constan et al. [7] present Sanctum, a hardware-software co-design that prevents controlled-channel attacks by dispatching page faults directly to enclaves and by allowing enclaves to maintain their own virtual-to-physical mappings in a separate page table hierarchy in enclave-private memory. To bypass these defenses, Van

Bulck et al. [38] observe that malicious operating systems can monitor memory accesses from enclaves without resorting to page faults, by exploiting other side-effects from the address translation process.

9.2 ASLR

Address Space Layout Randomization (ASLR) is used to mitigate memory corruption attacks by making addresses unpredictable to an attacker. ASLR is commonly applied to user-space applications (e.g., web browsers) and OS kernels (i.e., KASLR) due to its effectiveness and low overhead. Unfortunately ASLR suffers from various side-channel attacks which we discuss here.

Memory deduplication is a mechanism for reducing the footprint of applications and virtual machines in the cloud by merging memory pages with the same contents. While memory deduplication is effective in improving memory utilization, it can be abused to break ASLR and leak other sensitive information [2, 4]. Oliverio et al. [32] show that by only merging idle pages it is possible to mitigate security issues with memory deduplication. The AnC attack [12] shows an EVICT + TIME attack on the MMU that leak pointers in JavaScript, breaking ASLR.

Hund et al. [20] demonstrate three different timing side-channel attacks to bypass KASLR. The first attack is a variant of PRIME + PROBE that searches for cache collisions with the kernel address. The second and third attacks exploit virtual address translation side channels that measurably affect user-level page fault latencies. In response to these attacks, modern operating systems mitigate access to physical addresses, while it is possible to mitigate the other page fault attacks by preventing excessive use of user-level page faults leading to segmentation faults [20]. To bypass such mitigations, Gruss et al. [15] observe that the `prefetch` instruction leaks timing information on address translation and can be used to prefetch privileged memory without triggering page faults. Similarly, Jang et al. [22] propose using Intel TSX to suppress page faults and bypass KASLR.

10 Conclusion

In recent years, cache side-channel attacks have established themselves as a serious threat. The research community has scrambled to devise powerful defenses to stop them by partitioning shared CPU caches into different security domains. Due to their scalability, flexibility, and portability, software-based defenses are commonly seen as particularly attractive. Unfortunately, as we have shown, they are also inherently weak. The problem is that state-of-the-art defenses only partition the cache based on direct memory accesses to the cache by untrusted code. In this paper, we have shown that *indirect* cache attacks, whereby another trusted component such as the MMU accesses the cache on the attackers' behalf, are just as dangerous. The trusted component acts as a confused deputy so that the attackers, without ever violating the cache partitioning mechanisms themselves, can still mount cache attacks that bypass all existing software-based defenses. We have exemplified this new class of attacks with MMU-based indirect cache attacks and demonstrated their effectiveness against existing defenses in practical settings. We have also discussed mitigations and shown that devising general-purpose software-based defenses that stop arbitrary direct and indirect cache attacks remains an open challenge for future research.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and was supported by the MALPAY project and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). ISCA '10.

[2] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT '15.

[3] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *ACM SIGPLAN Notices*, volume 29, pages 158–170. ACM, 1994.

[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. S&P '16.

[5] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting Privileged Side-channel Attacks in Shielded Execution with Déjá Vu. ASIA CCS '17.

[6] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. *ACM SIGPLAN Notices*, 50(4):177–189, 2015.

[7] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. USENIX Security '16.

[8] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. USENIX Security '17.

[9] Xiaowan Dong, Sandhya Dwarkadas, and Alan L Cox. Shared Address Translation Revisited. EuroSys '16.

[10] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. S&P '18.

[11] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. USENIX Security '18.

[12] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU.

[13] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. USENIX Security '17.

[14] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. USENIX Security '17.

[15] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. CCS '16.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush + Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *S&P '11*.

[18] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family. HPCA '16.

[19] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. S&P '13.

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. S&P '13.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES. S&P '15.

[22] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. CCS '16.

[23] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. DAC '16.

[24] Richard E Kessler and Mark D Hill. Page Placement Algorithms for Large Real-Indexed Caches.

[25] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. USENIX Security '12.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution.

[27] Nate Lawson. Side-Channel Attacks on Cryptographic Software.

[28] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled Cache Predictability for Real-time Systems. RTAS '17.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space.

[30] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. HPCA '16.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. S&P '15.

[32] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure Page Fusion with VUsion. SOSP '17.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[34] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. USENIX Security '15.

[35] ARM Limited. PL310 Cache Controller Technical Reference Manual.

[36] CAT Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[37] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks against Enclave Programs. NDSS '17.

[38] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. USENIX Security '17.

[39] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. EuroSec '17.

[40] Barry Duane Williamson. Line Allocation in Multi-Level Hierarchical Data Stores, September 18 2012. US Patent 8,271,733.

[41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. S&P '15.

[42] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. USENIX Security'14.

[43] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. PACT '14.

[44] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. CCS '16.

Appendix A Extended Results

Figure 8 shows a comparison of PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT while applying page coloring or way partitioning with 4, 8 and 12 ways available to the attacker. Figure 9 shows that we can fully mitigate the XLATE family of attacks by extending page coloring to page tables.

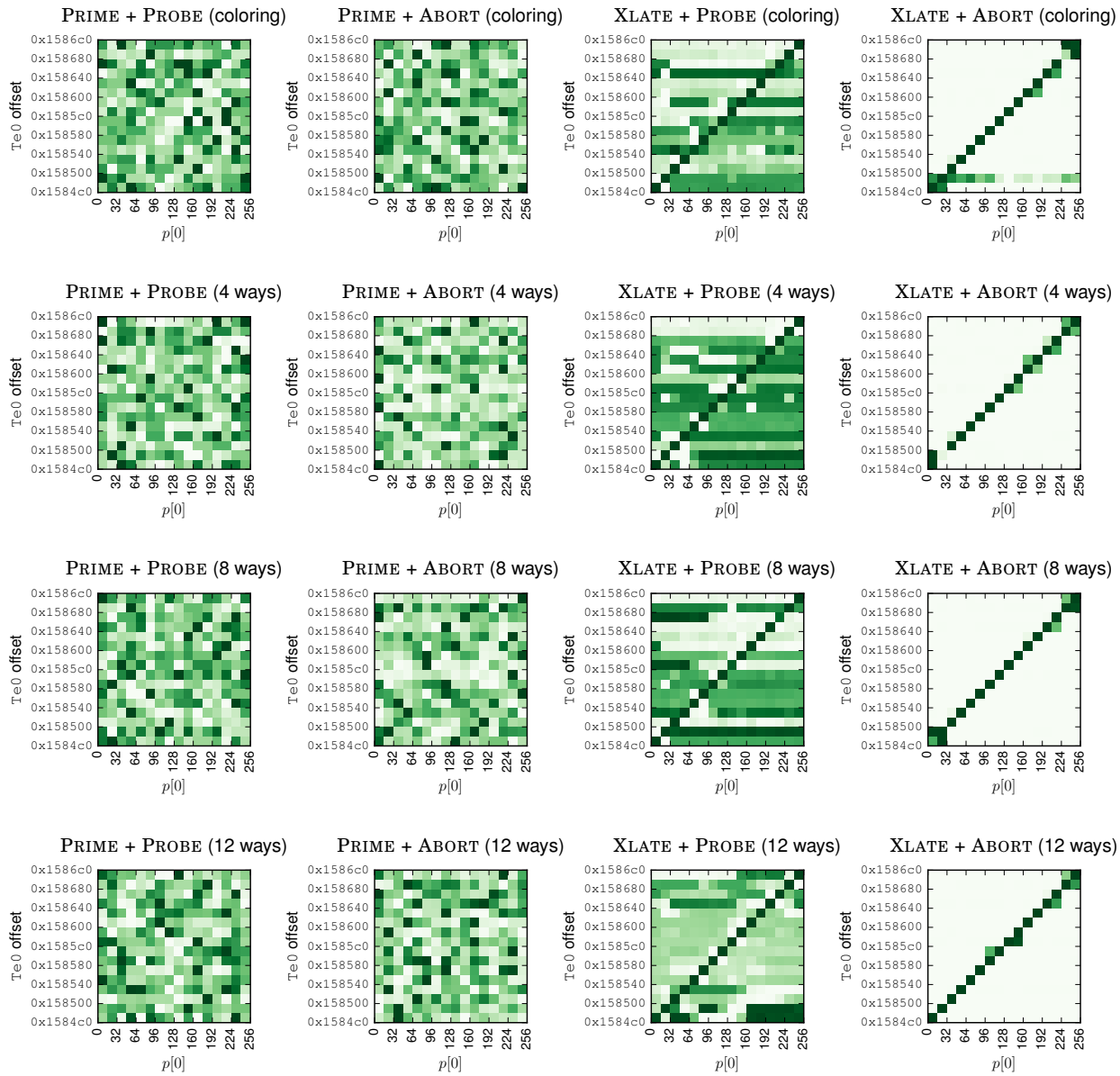

Figure 8: PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT against the AES implementation using T-tables in OpenSSL on an Intel Core i7-6700K @ 4.00GHz (Skylake) while various software-based cache defenses are active.

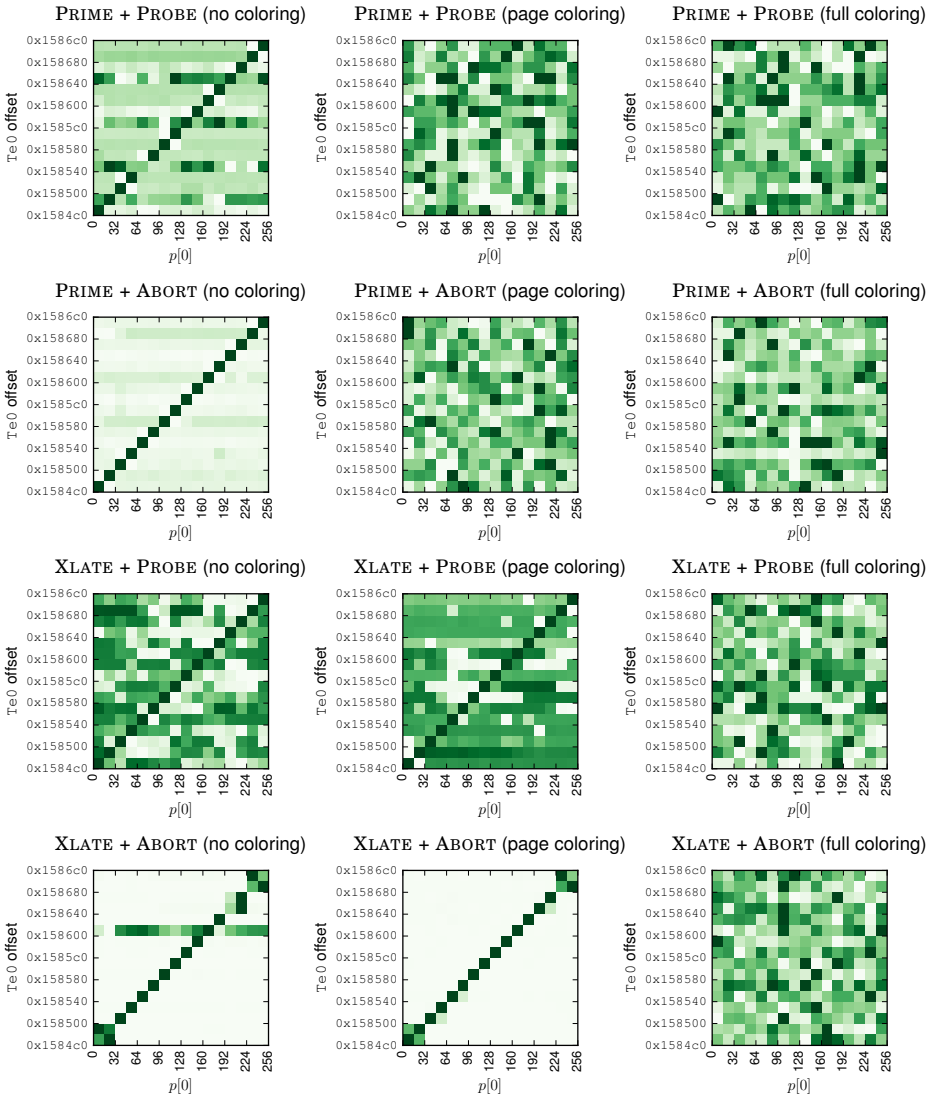

Figure 9: PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT against the AES implementation using T-tables in OpenSSL on an Intel Core i7-6700K @ 4.00GHz (Skylake) before and after the mitigation of coloring page tables.