Not Lost in Translation: Implementing Side Channel Attacks Through the Translation Lookaside Buffer

Undergraduate Thesis

Noah H [redacted]

May 2023

University of Warwick
Department of Computer Science

Supervised by **Prof. Feng Hao**

Year of Study: Third

Abstract

The increasing complexity of modern micro-architectures exposes an attack surface vulnerable to side-channel attacks. In such cases, attackers can overcome increasingly robust defences by leveraging the side-effects of a system rather than direct flaws in its implementation. Whilst cache side channels are well understood within the field, less explored are side-channel attacks employing the translation lookaside buffer (TLB). This project builds on existing work, further confirming the feasibility of TLB side channels and demonstrating the need for new micro-architectural defences. We implement the existing TLBleed attack on our target system and leak 256-bit EdDSA signature keys with a comparable success rate. In doing so, we use a novel deep learning approach for signal classification and investigate the implications of noise on the side channel. Additionally, we propose new TLB attacks for compromising user privacy by demonstrating a side channel can be used to infer the arrival of pings and identify the execution of common Linux commands. Finally, we show TLB signals can be detected across the user-kernel boundary to reduce the entropy of kernel address space layout randomization (KASLR) in similar example to the TagBleed attack.

Keywords: Side-Channel Attacks, Micro-architecture, Translation Lookaside Buffer, Cache, Deep Learning, Machine Learning, Convolutional Neural Networks.

Acknowledgements

First and foremost I would like to thank Feng Hao for his supervision and guidance throughout the project. Additionally, I would like to thank Ben Gras for his hints and advice on implementing the side channel and Florin Ciucu for his help in finalising this report. Finally, I would like to thank my friends and family who have encouraged and supported me during the year.

Contents

Abbreviations

- **ASLR** : Address Space Layout Randomization
- **CNN** : Convolutional Neural Network
- **DEP** : Data Execution Prevention
- **dTLB** : Level 1 Data Translation Lookaside Buffer
- **iTLB** : Level 1 Instruction Translation Lookaside Buffer
- **KASLR** : Kernel Address Space Layout Randomization
- **L1** : Level 1 (Cache)
- **L2** : Level 2 (Cache)
- **LLC** : Last Level Cache
- **MMU** : Memory Management Unit
- **PCID** : Process Context Identifier
- **PT** : Page Table
- **PTE** : Page Table Entry
- **PTL** : Page Table Level
- **SGX** : Software Guard Extensions
- **sTLB** : Level 2 Shared Translation Lookaside Buffer
- **SVM** : Support Vector Machine
- **TLB** : Translation Lookaside Buffer
- **TSX** : Transactional Synchronization Extensions

1 Introduction

Over recent years, the increasing complexity of systems and difficulty in exploiting traditional vulnerabilities has prompted an interest in less conventional offensive techniques such as side-channel attacks. In such cases, attackers leverage the complexity of system implementations to infer sensitive information through the analysis of its side effects. This contrasts with traditional attacks, such as memory-corruption exploitation, which manipulate the underlying functionality of a system to elicit unintended behaviour. Examples of potential side effects may include execution time, power usage, heat dissipation or electromagnetic radiation. Since all systems will inevitably induce side effects of some sort, successful side-channel attacks can enable simple, clean exploitation of modern systems with increasingly complicated mitigations in place. A simple analogy of a side-channel attack can be demonstrated during the exploitation of a keypad device. Rather than targeting the inner-workings of the device, we could determine the passcode through analysing the wear of each button. Aviv et al. [1] modernised this scenario to determine the pattern or passcode needed to unlock a touch-screen phone based on the smudges left by a user's finger.

In current-day systems many different targets have been used to implement side-channel attacks, but micro-architectural components, such as the cache, are the most popular. These components are highly-specialised, lying at the lowest level of the software stack, meaning they can be difficult to defend against. Countless examples have leveraged cache-based side channels to overcome both traditional software defences, such as (Kernel) Address Space Layout Randomization (ASLR) [9], and leak sensitive information, such as encryption keys [17, 43, 44]. Additionally, applications of such attacks have been extended to less conventional settings such as in the cloud and virtualized environments [37, 31]. In response to this manufactures and researchers have implemented and proposed a number of defences including cache isolation [48, 26] and transactional synchronization extensions (TSX) [13]. However, as we will see these mitigations are mistakenly not extended to the translation lookaside buffer (TLB).

Contributions. This project focuses on demonstrating the feasibility of TLB side channels and highlights the need for new mitigations or existing cache-based defences to be extended to the TLB. In particular, we carry out a number of end-to-end attacks leveraging the TLB, including both existing side channels and new attack scenarios. To do this, we first implement a number of experiments [39] to reverse engineer key properties of our target system necessary for the attack.

Then, to confirm the reverse engineering results and correctness of our implementation primitives, we construct a rudimentary covert channel enabling two processes to communicate through the TLB. With this background, we demonstrate new applications of TLB attacks by showing a Prime+Probe side channel can leak the arrival of pings to a victim process and detect which Linux commands are being executed. We then implement the TLBleed attack [10] enabling a spy process to leak 256-bit EdDSA signature keys. Using a novel machine learning approach for signal classification, we achieve a comparable success rate to the original attack, and additionally investigate the impact of noise on the side channel. Finally, we construct a proof-of-concept demonstrating that TLB signals can be detected across the user-kernel boundary in an Evict+Time attack. This enables us to reduce the entropy of KASLR, similar to the TagBleed attack [22].

Outline. The remainder of this report is structured as follows. In Section 2 we start by covering background information on existing attacks and the necessary technical knowledge to follow the remaining sections. Then, Section 4 introduces our methodology for the reverse engineering process and Section 5 discusses the implementation of our covert channel. In Section 6 we explore the details and results of our TLB side channels before proposing potential mitigations for the attacks in Section 7. Finally, Section 8 talks about the future direction of TLB side channels and the remainder of the report discusses conclusions and reflections of the project.

2 Background

Similar to other areas of security, micro-architectural side channels have evolved as a cat-and-mouse game since their inception. As systems have become more complex with increasing layers of abstraction and hardware components, their attack surface has followed suit. To date, numerous attacks have been demonstrated targeting a number of components, each with different variations and bypasses for their respective countermeasures. In this section we discuss the recent advances in micro-architectural side channels and the necessary technical material to follow the rest of the paper. We start by exploring side channels exploiting speculative execution units before discussing the technical implementation details of cache attacks. Finally, we introduce TLB side channels and how they relate and differ to their cache-based counterparts.

2.1 Speculative Execution & Prefetch Side Channels

Most systems these days employ speculative execution units and prefetching to preemptively compute and load results in case they are later needed. Whilst this has become a key optimization technique for modern devices, as we will see these components are susceptible to side channel leakage.

Spectre & Meltdown. In recent years, arguably the most famous demonstrations of micro-architectural side channels are the Spectre and Meltdown attacks [19, 24] - the original side channels leveraging transient execution. In these attacks, a manipulation of branch prediction and speculative execution can enable a malicious process to leak sensitive information from victim applications, including the kernel. The attacks build on the assumption that the influence of speculatively executed instructions on the cache are persistent regardless of whether the instruction is later squashed. More specifically, in the case of original Spectre variant, the attack first induces the speculative execution of an instruction that would not occur in serial processing due to branch misprediction. Then using a timing attack on the cache the attacker can deduce the sensitive information of the executed instruction. This is summarised in Figure 1. Meltdown builds on a similar principle to exploit a race condition between memory accesses and privilege checks to read kernel memory. Here, the transient instruction accesses a cache line based on the contents of privileged memory. Whilst traditionally this would raise an exception and prevent underlying access to the memory, speculative execution can mean the access is made first. Then as before, the attacker can use a cache side channel to infer the contents of the kernel memory. This is summarised in Figure 2.

```
if (condition) {
      tmp = buffer[*addr];
}
```

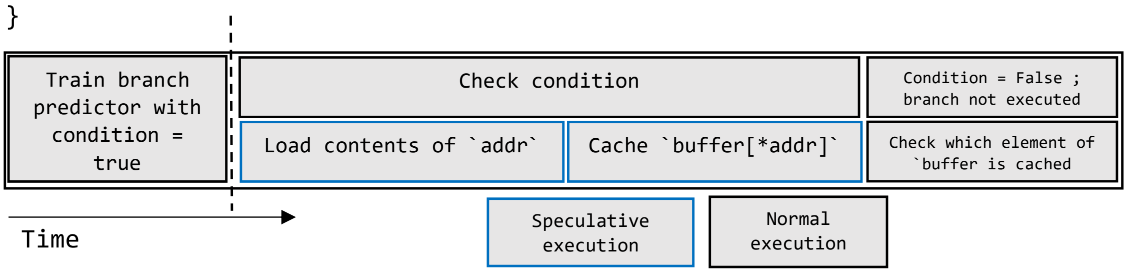

Train branch predictor with condition = true
Check condition
Condition = False ; branch not executed
Load contents of `addr`
Cache `buffer[*addr]`
Check which element of `buffer is cached
Time
Speculative execution
Normal execution

Figure 1: Example Spectre execution timeline.

```
Char tmp = buffer[*kernel_addr];
```

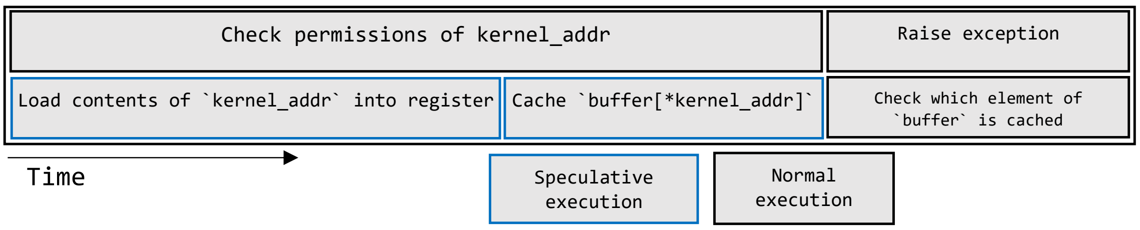

Check permissions of kernel_addr
Raise exception
Load contents of `kernel_addr` into register
Cache `buffer[*kernel_addr]`
Check which element of `buffer` is cached
Time
Speculative execution
Normal execution

Figure 2: Example Meltdown execution timeline.

Foreshadow. Following the original disclosure of Spectre and Meltdown numerous variants of the side channel have been exploited [21, 28]. In one such example [5], the Foreshadow attack compromises Intel's Software Guard Extensions (SGX). SGX aims to provide instruction support for user-level code to define specially-protected portions of memory called enclaves. These are encrypted by the CPU and are decrypted dynamically within the processor only when required, preventing unauthorised access. Similar to previous Meltdown variants the attack leverages transient instruction execution following an exception. However, dereferencing enclave memory does not produce a page fault and instead silently reads dummy data. The attack overcomes this by first clearing the present bit of enclave pages in the page table such that any subsequent access to the memory does induce an exception. The Foreshadow attack then ultimately goes on to leak private keys from within an enclave using a speculative execution side channel as in Meltdown.

PACMAN. The exploitation of traditional memory-corruption vulnerabilities now requires overcoming a number of mitigations such as ASLR, Data Execution Prevention (DEP), and Secure Structured Exception Handling (SafeSEH). However, in a more recent example of a speculative execution side channel, the

PACMAN [36] attack overcomes Pointer Authentication (PA) - a key hurdle for memory-corruption exploitation in ARM. Specifically, PA significantly complicates the modification of protected areas of memory by signing pointers using a cryptographic hash of its value and storing it in unused bits of the pointer. This is known as a PA code (PAC). In the PACMAN attack, they design a gadget leveraging speculative execution which accesses such a signed pointer. Importantly, by using speculative execution, the attackers avoid crashes which normally occur when accessing a signed pointer with an incorrect PAC. Following this, another micro-architectural side channel can be leveraged to determine whether a correct or incorrect PAC was used during the transient access. In the case of the original implementation of the attack the authors leverage a TLB side channel, observing using correct PACs result in more TLB activity than using incorrect PACs. During exploitation of a memory-corruption vulnerability an attacker can repeat this process to identify a correct PAC code for their attack.

Prefetch Side Channel. In a final example, we consider a prefetch attack leveraging a timing and power side channel on AMD architectures [23]. The set of prefetch instructions enable programmers to provide hints to the CPU to preemptively load data. This could include locations that are invalid or inaccessible to the executing process such as privileged and unbacked virtual addresses. In these cases the instruction is simply ignored. Importantly though, in order to prefetch a virtual address, the processor requires the corresponding physical address. This involves accessing the TLB and page tables. As shown by the attack, timing and power variations of prefetch instructions exist depending on the level in the translation hierarchy the instruction aborts at. With this the attack implements two primitives, Prefetch+Time and Prefetch+Power, to deduce the TLB state and page table level of an (inaccessible) address. They obtain precise results by leveraging the fact that the Intel and AMD Zen architectures do not store invalid entries in the TLB enabling them to repeat the primitives over multiple iterations. However, in the case of the Zen 2 architecture, invalid entries are stored in the TLB and so the measured leakage is only representative for the first prefetch. To overcome this they introduce the TLB-Evict+Prefetch primitive. This first evicts the corresponding TLB entry before using Prefetch+Time or Prefetch+Power. They can then use this primitive over a number of iterations to reliably leak the level of the page table used for translating the virtual address of a prefetch instruction on Zen 2. Using these, the authors demonstrate the first micro-architectural break of fine-grained KASLR on AMD by inferring mapped and unmapped kernel pages.

2.2 Cache Side Channels

Almost all modern CPUs implement a number of caches of varying types. In this section we consider the most popular: the data and instruction cache. Such components are located physically close to the CPU and are used to provide fast access to recently used contents from memory. In contrast to speculative execution side channels, cache-based attacks are more explored dating back much further. Over the years many variants of the attack have been developed. We start this section by covering the implementation of caches, useful for our exploration later into TLBs, before discussing the evolution of cache side channels and their variations.

2.2.1 Cache Design

Cache Architecture. Cache systems are implemented as a hierarchy with the smallest, but fastest components at the top and larger, but slower components further down. In most modern systems this hierarchy is three levels high. The first two levels (L1 and L2) are implemented local to each core and the last level cache (LLC) is shared amongst cores. Additionally, in most cases the L1 cache is split into data and instruction components storing memory contents depending on the context of their use. For side channels, it is important to highlight the following difference between the LLC and lower level caches. Since the L1 and L2 cache are local to each core, side channel leakage is only possible between two co-resident processes executing on the same core. This is known as hyper-threading on Intel and simultaneous multithreading on AMD. This differs from the LLC where side-channel attacks can be launched between two processes executing on separate cores. Figure 3 summarises the cache hierarchy.

Cache Implementation. Each cache is defined by two key properties, the number of its sets and ways. Primarily, a cache is partitioned into a number of sets such that each entry for a cache maps to a particular set. This is defined by the mapping (or "hash") function. Within each set, an entry is inserted into any number of the `n` ways in the set. Entries in each set are evicted according to some replacement policy, often implemented as a variation of least recently used (LRU). Designing caches in this way reduces the required complexity of hardware by limiting the entries the replacement policy has to consider. In cases where the cache is not partitioned at all, i.e. it has one set, it is said to be fully-associative.

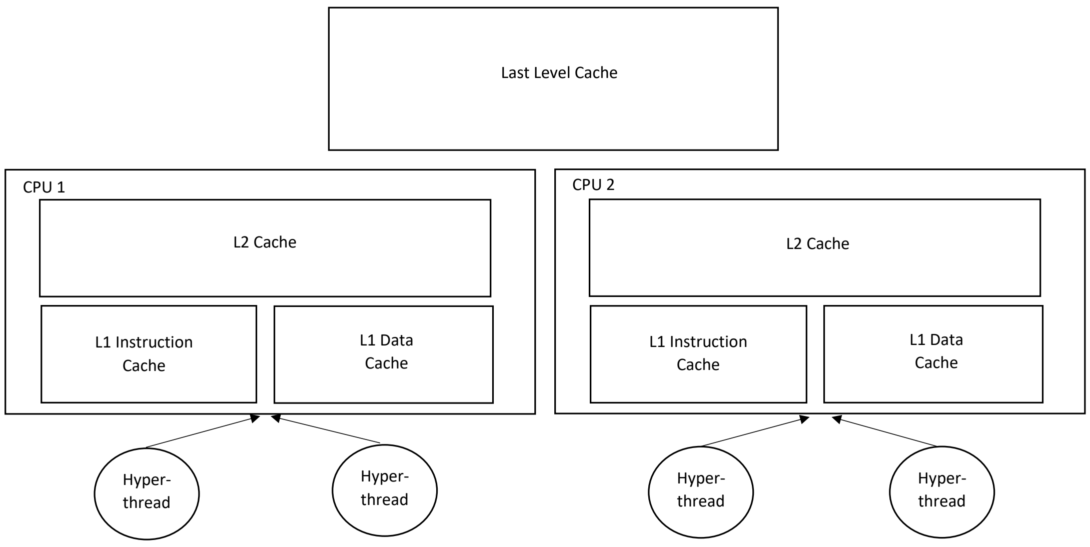


Figure 3: Typical cache hierarchy on a modern system.

Virtual Memory. To implement proper isolation and to optimize system performance it is common place for modern operating systems (OS) to support virtual memory. This abstracts away the underlying implementation of the system's physical memory. Each process has access to its own contiguous virtual address space which is divided up into a number of chunks known as pages. The OS is then responsible for translating virtual addresses to their corresponding physical address as required. This simplifies the implementation of user-space applications for a number of reasons, but primarily by enabling developers to assume their memory is allocated contiguously. In reality though, the OS can distribute processes' memory more efficiently. For example, when two processes load the same contents of memory into their virtual address space, the OS can de-duplicate the underlying memory by mapping them to the same physical addresses. This is significant for the Flush+Reload cache side channel discussed later. Importantly, different levels of the cache hierarchy will implement a mapping function based on either virtual or physical addresses. For the lowest levels of the cache hierarchy, where minimal latency is particularly important, components are often indexed by virtual address. This is to avoid the latency associated with translating a virtual address to its physical address. In contrast, by the time the LLC is accessed it is likely such a translation has completed, and so such components can be indexed by physical address.

2.2.2 Cache Attacks

As discussed, cache components are shared amongst processes. Inevitably, this opens the door for one process to monitor the activity of another through its cache usage. We now consider a number of approaches that have been developed to implement a cache side channel.

Flush+Reload. The Flush+Reload [43] side channel exploits the sharing of physical pages between processes. To do this, the attack targets a physically-indexed LLC meaning the cache entries of the shared physical page can be used by both processes. In the attack, the adversary leverages this to detect a victim thread's accesses to data (or instructions) within the shared page. This works as follows. The attacker first flushes the cache using the `clflush` instruction, removing the entry for the shared data. Then, after waiting for a short period, the attacker times an access to the data. A quick access indicates that the page was cached, and hence the sharing process must have used the page in the time they were waiting. By repeating this during the execution of the victim process, an attacker can reconstruct a partial execution trace. In the case of the original implementation of the attack, the authors trace a victim's function calls during an old implementation of RSA decryption in GnuPG 1.4.13. With these they can reconstruct the victim's private key.

Flush+Flush. The Flush+Flush [12] attack aims to bypass a countermeasure introduced for the Flush+Reload side channel. In particular, assuming all cache side channels induce more hits and misses than benign applications, such attacks can be detected using performance counters measuring this [34, 6]. To overcome this the attack is implemented in a similar process to the Flush+Reload attack, but instead of timing an *access* (or the "Reload") to the shared cache line, the spy can use the `clflush` instruction. The execution time of `clflush` depends on whether its data is cached or not. With this the attackers can implement a similar variation of the Flush+Reload side channel, but without making additional memory accesses.

Evict+Reload. Another proposed defence against the Flush+Reload attack restricts access to the `clflush` instruction. However, by using an Evict+Reload attack [11], the countermeasure can be bypassed. The attack follows the steps outlined in the Flush+Reload attack, except for the initial removal of the shared cache entry. Rather than relying on the availability of `clflush`, an attacker can flush the cache by making a number of accesses such that the entry is evicted "naturally" by the cache's replacement policy.

Prime+Probe. Whilst the previous cache side channels are limited to cases

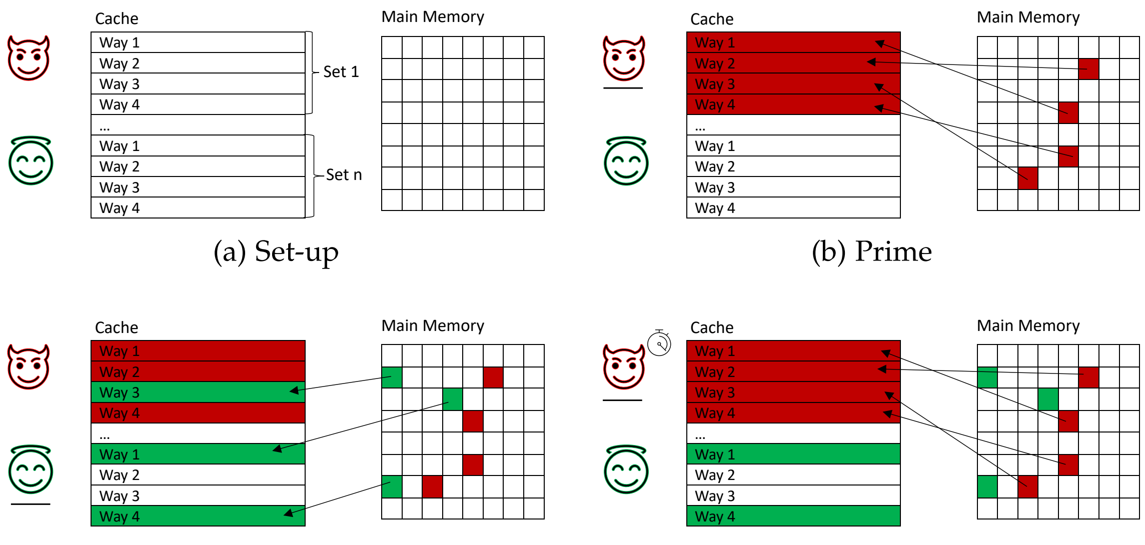


Figure 4: Outline of Prime+Probe cache attack.

where processes share physical pages, a Prime+Probe attack is not [9, 27, 33, 17]. In this implementation, an attacker infers the usage of particular cache sets by a victim. It begins with an attacker accessing an *eviction set*. This is a collection of addresses that fill all ways within a particular cache set. Then, by repeatedly timing accesses to this set the attacker obtains a latency signal. Spikes in the signal, where accesses to the eviction set are slower, indicate competing use for the set by the victim. This attack is summarised in Figure 4. Whilst not immediately obvious how such knowledge can be leveraged for an attack, since there is no reliance on shared pages, it is a popular approach. For example, the AnC attack [9] uses a Prime+Probe side channel to de-randomize ASLR in the browser. More specifically, they use a cache side channel to identify which sets are accessed due to an induced page table walk for a target address. Knowing this, they can identify the offset of the corresponding entry for the target virtual address within each page table level. Since addresses are positioned in the page table according to their virtual address, this enables them to infer the randomized bits of the virtual address.

Evict+Time. One of the earliest implementations of a cache side channel was the Evict+Time attack [32, 9]. In contrast to other cache-based approaches, such as Flush+Reload and Prime+Probe, this attack can only target one cache set at a time. Additionally, to implement the attack, the spy must be able to monitor the execution time of the victim. The attack is implemented in following the three steps. Firstly, the victim program is executed for a number of iterations, establishing a "baseline" execution time. During this, the cache will be populated with entries used by the victim. Subsequently, the attacker evicts a particular cache line (or set) and times a final victim operation. Depending on the usage of the targeted cache line, the execution time will vary. From this, the attacker can infer the cache lines and sets used by variables and functions during the victim execution. This approach has been used to de-randomize ASLR [9, 22].

Prime+Abort. One proposed strategy to defend against cache side channels is Intel's Transactional Synchronization Extensions (TSX) [13]. This implements a number of additional CPU instructions to facilitate transactional support at the hardware level. This enables processes to execute sensitive operations within transactions which fail if their accesses are evicted from the cache. However, the Prime+Abort attack [8] leverages TSX to implement a new variant of a cache side channel. In particular, in a similar fashion to a Prime+Probe approach, the attackers access a set of addresses which map to a particular set within the cache. Importantly though, they do this within a transaction. If the cache set is then subsequently used by another process, their entries are evicted and hence their transactions aborts. This avoids the need for any timing as is the case in all previously mentioned cache side channels, particularly relevant in cases where the resolution or availability of the timestamp counter is limited [29].

2.2.3 Cache Defences

Secret Independence. An obvious approach to mitigating the risk of cache attacks is ensuring any timing information obtainable through a side channel is independent of secret information. For example, in the cryptography space, this means implementing solutions whose execution time and memory access patterns are independent of the key [16, 20]. However, such a solution is often application specific and difficult to verify. Instead an ideal solution resolves the issue at the hardware layer, removing the dependence on the programmer to implement side-channel resistance.

Cache Isolation. Another potential mitigation is to implement better isolation between two processes in the cache. A number of proposed solutions fall into this category. The most obvious approaches implement isolation by limiting the entries in the cache that processes share. This can be done partitioning by ways [26] or by sets [4, 48]. Interestingly, whilst partitioning ways requires hardware support, set isolation can be implemented for physically-indexed caches at the OS level by managing the underlying physical memory of processes. Another isolation-based approach limits cache concurrency [4]. In cases where hyper-threading is disabled, albeit not often these days, a spy can only monitor L1 or

L2 cache activity having preempted a victim process. Therefore, by restricting the number of context switches, a malicious actor has less opportunity to observe victim activity. A similar mitigation implements isolation by flushing the cache on each switch between processes [46]. Unfortunately, despite isolation founding a strong defence, its main drawback is the additional overhead introduced given its restrictive nature.

Timer Restrictions. Environments at higher levels of abstraction, such as in the browser, introduce additional complexities when implementing cache side-channel attacks. Most notably is their lack of precise timing support. In these cases, additional noise is introduced into an attacker's observed cache signal, impairing their ability to leak secret information. This can be extended to hardware timers by limiting their resolution or availability [29]. Unfortunately, despite the effectiveness of such solutions, other timing mechanisms can be implemented by attackers without the need for underlying support. For example, through the use of multithreading, an attacker can initialize a thread which simply increments a counter repeatedly. This can then be read and used as a timestamp. Such an implementation enabled attackers to leverage a cache side channel to break ASLR in the browser [9]. Additionally, as we have seen during the Prime+Abort discussion, not all cache side channels rely on timing.

Transactional Synchronization Extensions. As discussed earlier, TSX enables exclusive cache access to a process during the operation of sensitive operations [13]. This approach is effective at preventing timing-based side channels, but also introduces alternative attacks, namely Prime+Abort [8]. In addition, such a defence requires each application to implement transactional memory where required. This means whilst effective for specialised contexts, such as cryptographic applications, it is less effective as a broader defence.

Randomization. Another proposed defence against cache attacks introduce randomness into its design. Examples such as ScatterCache [41] randomize the mapping of entries in the cache, complicating the construction of eviction sets. Alternatively, approaches such as the Random-Fill Cache [25] implement non-determinism in the entries bought into the cache by additionally caching adjacent memory locations for a particular access. Unsurprisingly though, randomized approaches additionally introduce a (small) performance overhead. Recent work, such has the Prime+Prune+Probe variant, has also demonstrated how probabilistic eviction sets can be used to overcome randomized cache architectures [35, 3].

2.3 TLB Side Channels

In the following section we introduce the Translation Lookaside Buffer (TLB), its relation to cache attacks and previous work in the area.

2.3.1 TLB Design

Virtual Address Translation. As discussed earlier, virtual memory is a core technology supported by modern operating systems. To do so, the OS implements a hierarchical structure to translate between virtual and physical addresses. This is known as the page table (PT) and is implemented by the memory management unit (MMU). The PT is a tree-like structure with leaves storing physical addresses. At each level of the tree a subset of bits in the virtual address indicate the outgoing edge for the next level. On modern x86_64 architectures virtual memory is implemented on a page-by-page basis with a standard page size of 4KB. This means the lowest 12 bits of a virtual address index a particular byte within a page and, since the upper 16 bits of a virtual address are currently unused by modern systems, the remaining 36 bits define the virtual address space. Typically, these 36 bits are divided up into four 9-bit sections defining four page table levels (PTLs). This implies each PT stores 512 page table entries (PTEs). This is summarised in Figure 5.

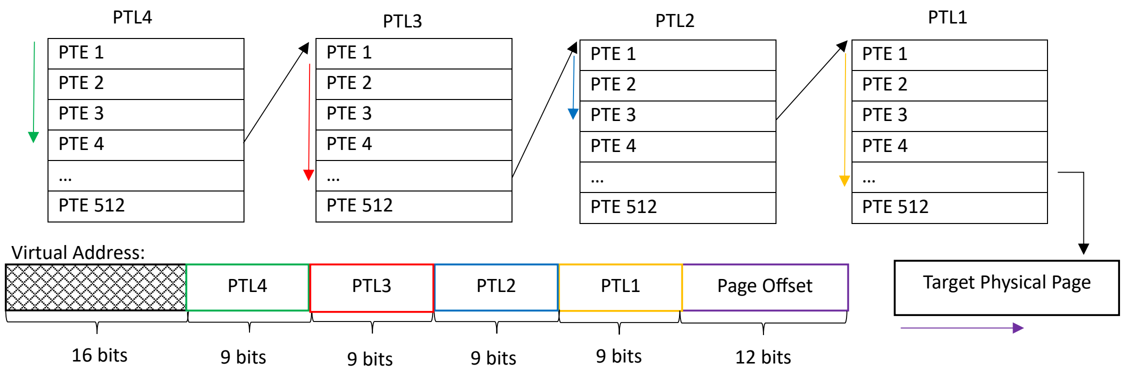


Figure 5: Typical virtual address structure with four page table levels and 4KB page size.

TLB Hierarchy. As we have seen for each virtual memory reference, we can require up to four preliminary memory accesses to walk the PT and obtain the corresponding physical address. Whilst the MMU does support caching mechanisms for page tables, the TLB is another component used to reduce latency in address translation. In particular, the TLB is a cache which stores recently computed virtual to physical address translations. Similar to traditional caches, the TLB is a hierarchical structure often implemented with two levels, where the first is split between data and instruction translations (the dTLB and iTLB respectively). The second level on the other hand stores translations for both data and instruction accesses (the L2 sTLB). Importantly, current systems do not implement a TLB component shared between cores as is the case with the LLC. Mirroring their cache counterparts, TLB components also implement partitioning of entries into sets and ways.

Virtual Address Space Identifiers. Since a virtual address space is unique to each process, the TLB entries of one process should not be used by another. To enforce this, a blunt approach flushes the TLB on each context switch. However, clearly this solution has performance implications and does not consider concurrent TLB accesses between hyper-threads. To overcome this, modern systems "tag" TLB entries with an address space identifier, referred to as a Process Context Identifier (PCID) on Intel. Then, for a translation with a matching virtual address, the TLB additionally compares the hyper-thread's PCID with the tag.

2.3.2 TLB Manipulation

TLB Attacks. To the best of our knowledge only two existing attacks implement a micro-architectural side channel with exclusively the TLB. One attack, TLBleed, extends the Prime+Probe cache side channel to the TLB. In particular, they use a TLB side channel to leak the signature key used by a victim during the execution of an implementation of EdDSA by libgcrypt [10]. We discuss this in more depth in Section 6 when we explore our implementation of the attack on our target system. In an earlier example, the Double Page Fault attack [15] leverages the caching of translations in the TLB even when they raise an exception. Using this they implement a side channel to de-randomize KASLR. More specifically, they measure two executions of the kernel page fault handler for a particular address. A quicker second execution indicates an entry was made in the TLB and so the corresponding page was mapped by the kernel.

Broader TLB Exploitation. A couple of examples exploit a TLB side channel in combination with another micro-architectural attack. In one such case, as discussed earlier, the PACMAN attack [36] leverages a TLB side channel in conjunction with speculative execution to break pointer authentication on ARM. In the other case the TagBleed side channel [22] uses a combination of the AnC cache attack [9] and an Evict+Time TLB side channel to de-randomize KASLR. More specifically, since the kernel is mapped with 2MB pages, knowing which sTLB set a virtual address maps to is sufficient to de-randomize the lowest $log_2(s)$ bits (where s is the total number of sets in the sTLB). Using the AnC attack they can de-randomize the remaining bits. We discuss this is greater depth is Section 6. Other broader examples of TLB manipulation include during the exploitation of a Rowhammer vulnerability [18] in the PTHammer attack [47]. The Rowhammer vulnerability is a hardware weakness in DRAM which enables an attacker to cause a leak of electrical charge by repeatedly accessing a row in DRAM. This results in bit flips in neighbouring rows. In the PTHammer attack they leverage repeated PT walks to hammer DRAM, but in order to do so require flushing the TLB and cache of the corresponding PTEs. In a similar vein, the Leaky Cauldron [40] attack and the previously discussed power and timing prefetch side channels [23] rely on flushing the TLB to implement their exploit.

TLB Reverse Engineering. Implementing micro-architectural side channels attacks require a detailed understanding of the underlying hardware of the target system. In particular, for cache and TLB side channels knowledge of the sets, ways and hash function is generally a minimal prerequisite. In the case of the TLBleed attack this was determined using Linux Performance Counters, but more recent work introduces a new *desynchronized* approach for reverse engineering the TLB [39]. We discuss this in-depth in Section 4. Importantly though, their work has uncovered the replacement policies implemented within the TLB, enabling the efficient construction of eviction sets. For example, a naive approach flushes a particular entry within the TLB by accessing a total of `w` virtual addresses which map to the same set (where each set has `w` ways). However, an optimal approach can leverage an understanding of the underlying replacement policy to evict entries in fewer than `w` accesses. With this they can improve the performance of side channels relying on TLB manipulation and implement a robust covert channel.

TLB vs Cache Attacks. Despite the similarities between traditional caches and TLB components, there are a number of reasons that differentiate TLB side channel implementations. Most obviously, unlike LLC attacks which do not require core co-residency between the attacker and victim, TLB components are local to each core. This means for side channels relying on concurrent access between the victim and spy, as is the case in most Prime+Probe based approaches, hyperthreading must be enabled. Additionally, since TLB entries are unique to each process, TLB side channels cannot implement traditional cache attacks relying on shared physical pages such as Flush+Reload, Flush+Flush and Evict+Reload. Another complicating factor is the limited granularity of a TLB side channel compared to a cache side channel. As discussed, entries in the TLB correspond to virtual page translations, which is implemented at the granularity of the system page size. Generally this is 4KB. In contrast, common cache lines are multiple orders of magnitude smaller, having a size of at most 128 bytes. This means, whilst a cache side channel can differentiate between victim memory accesses that are at least 128 bytes apart, a TLB attack can only do so when the accesses are made to distinct pages. Because of this, TLBleed relies on the temporal analysis of its side channel which differs to the most common approaches used in cache attacks where spatial analysis is sufficient.

TLB Defences. Mitigations for TLB side channels are discussed in depth in Section 7.

3 Threat Model

The remainder of the report outlines our process of implementing various end-to-end TLB side-channel attacks on a target device. (Relevant parts of its specification can be found in Appendix A). For real-world applications of our work we rely on a number of key assumptions. Most importantly, we assume an attacker has local, unprivileged access to the target system. Our attack relies on the ability to execute a process concurrently with a victim on the same core. We also assume the adversary has the means to obtain the required prerequisite knowledge to stage an attack. This includes an understanding of the underlying hardware of the target system. Realistic circumstances under which the above satisfy could include local privilege escalation. Given the complexity of the attack, we believe TLB side channels are most relevant for advanced persistent threats (APTs) such as nation-state actors.

4 Reverse Engineering

This project focuses on implementing TLB side channels based on the Prime+Probe and Evict+Time cache-based techniques. For both approaches, it is important to understand how eviction sets can be crafted for the particular TLB hierarchy on the target system. Most importantly, this requires identifying how virtual addresses are assigned entries in the TLB, but additionally involves determining its number of sets and ways. With this, we can design a malicious process capable of observing TLB activity. In this section we outline our approach to identifying these key TLB properties.

4.1 Desynchronization

To reverse engineer the TLB we design and implement a number of experiments whose results enable us to infer key properties and implementation decisions of the component. During these tests it will be necessary to confidently differentiate between hits and misses in the TLB. In the TLBleed and TagBleed attacks [10, 39] this was done through the use of Linux Performance Counters. However, such an approach has limited granularity and is only able to count the total number of misses over a section of code. This places restrictions on the tests that can be designed and introduces additional noise into the results. Instead, a more modern approach [39] uses *desynchronization* to be able to provide a finer-grained tool for differentiating whether a single memory access hits or misses in the TLB hierarchy. This is based on the observation, that unlike in the cache, the translation hierarchy does not implement proper coherency protocols. As a result we can invalidate entries further down the hierarchy (in page tables) to distinguish hits and misses further up (in TLB components).

More specifically, using desynchronization we can confidently deduce the eviction of a particular TLB entry. We start by probing a chosen virtual memory address resulting in an entry in the TLB being populated. We then alter the corresponding page table entry in memory in some way such that it becomes *desynchronized* with the TLB entry. On a subsequent access to our virtual address we can distinguish between a hit and a miss depending on which translation is used, either the original TLB entry, or the *desynchronized* page table entry. There are a couple of options for the desynchronization process, one of which invalidates the page table entry resulting in a subsequent miss causing a page fault. Whilst easy to measure this approach will result in poor performance and generate noise as the OS attempts to handle the page fault. A better approach simply alters the corresponding physical address of the translation in the page table entry such that it points to a different frame holding different data. That way a miss is distinguished from a hit depending on the data that is read (or executed) on accessing a page. This approach provides a more robust method and is much less prone to error and noise than relying on timing or performance counters. This process is summarised in Figure 6.

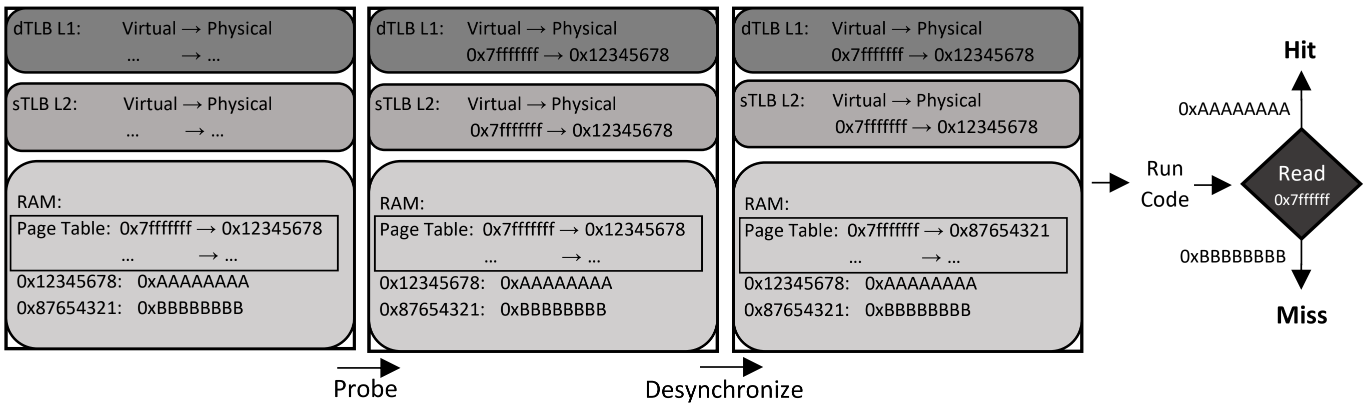


Figure 6: Desynchronized approach to determine whether a section of code evicts a particular TLB entry.

4.2 Inclusivity & Exclusivity

Exclusivity. Whilst inclusivity and exclusivity are not particularly relevant for TLB side channels, they are a good starting point for exploring how desynchronization can be utilised to reverse engineer the TLB. In this context, a TLB being exclusive defines that no entry in one particular level can be present in another. We test for this by probing some virtual page, filling an entry in first level of the TLB. We then desynchronize the TLB from the page table and issue a subsequent probe of the opposite type (instruction or data). A hit indicates the original access must have filled an entry in both the first and second levels of the TLB and so is non-exclusive. On our target system we repeat this for a large number of iterations and observe less than 1% of subsequent accesses miss, implying the TLB is non-exclusive.

Inclusivity. In contrast, inclusivity defines that an entry in one level of the TLB must be present in levels further up. To test for inclusivity we probe some address with either an instruction or data fetch resulting in entries in both L1 and L2 being filled. We then flush all components of the TLB storing translations of the opposite type by accessing a large number of pages exclusively using the other type. Finally, we determine if a subsequent access of the original type to the original page results in a hit or a miss. A hit would indicate that the first level of the TLB is separated for data and instruction translations and the entry must have been stored in the first level of the TLB despite having its entry flushed out of the second level (and so in non-inclusive). On our target system we find the TLB hierarchy is non-inclusive.

4.3 Set Mapping

L1 TLB. To be able to construct eviction sets for a side channel, knowledge of the number of sets and ways in the TLB is necessary. This also includes determining the mapping / hash function of the TLB. As shown in the TLBleed attack [10], modern systems implement a linear hash function for the L1 TLB components. Building on their work we reverse engineer these properties on our target system. We test this under the assumption that for an L1 component with s^* sets (and w^* ways), accessing an address in virtual page VA will fill an entry in some set $s' = VA \ mod \ s^*$. Given this, we design an experiment that enumerates all reasonable combinations of total sets s and ways w. For the correct combination of s and w we observe that probing $w^* + 1$ virtual pages at a stride of $s^* \cdot PAGE\_SIZE$ (so that they map to the same set) will result in exactly one eviction. We also note that probing $w > w^*$ pages that map to the same set will result in additional evictions. This is also the case when testing s is a multiple (or factor) of s^*. Given this, to test some combination of s and w we do the following. We first probe, via data or instruction loads, $w + 1$ pages mapping to the same set. We then flush the second level TLB by issuing a large number of probes with the opposite access type. Finally we re-access the original $w + 1$ pages counting any misses. We note the smallest combination of w and s inducing any misses must correspond to the true values s^* and w^*.

L2 sTLB. Whilst the hash function used by the dTLB and iTLB components are linear, this is not always the case for the L2 sTLB. Instead, similar to LLC components [45], previous work [10, 39] has demonstrated many modern Intel systems employ a hash function which XORs a subset of the lowest bits of the virtual address. More specifically, for a given 4KB virtual page VA, the XOR_N function computes its corresponding sTLB set s by taking the lowest $2N$ bits, after the page offset, and XORs both halves together. More formally this could be denoted by $s = VA[N + 13 : 13] \oplus VA[2N + 14 : N + 14]$. This gives a total number of 2^N sTLB sets. This is visualised in Figure 7. To identify which value of N our system implements we perform a similar experiment to that used for the iTLB and dTLB.

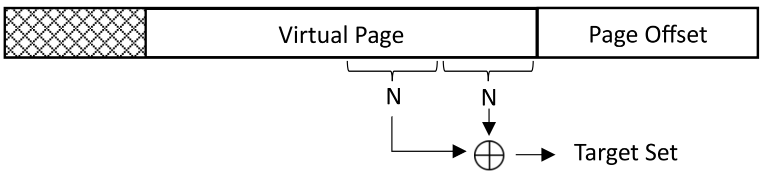
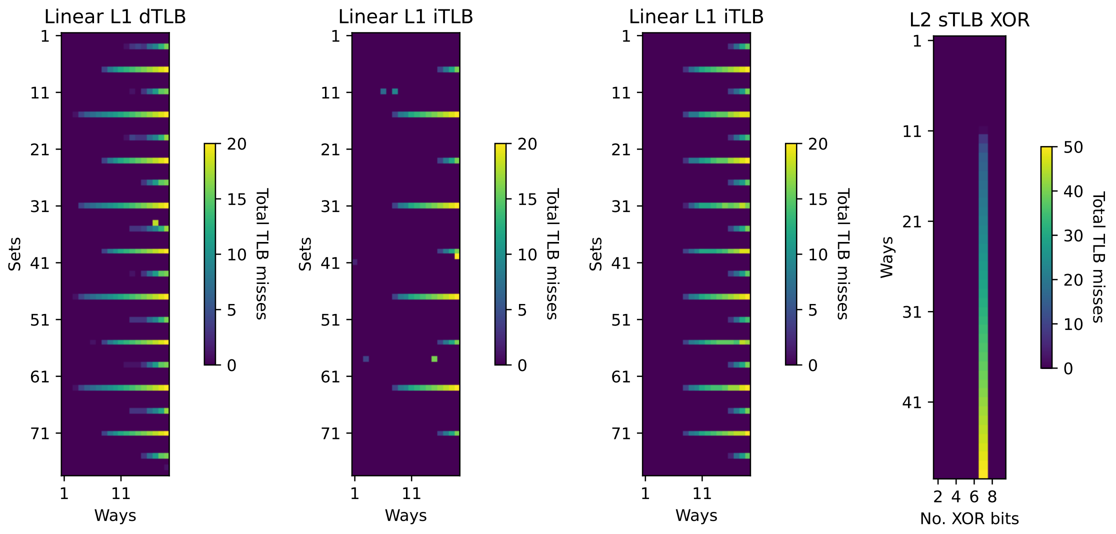


Figure 7: XOR-N Mapping for sTLB.

Results. Our results for the set mapping tests are show in Figure 8 and summarised in Table 1. For the most part they are consistent with the system's specification (found in Appendix A). However, interestingly the iTLB appears to implement dynamic partitioning, halving the number of available sets depending on the activity of the co-resident hyper-thread. This is consistent with existing work [39, 10]. An important implication of dynamic partitioning suggests the iTLB cannot be used to stage a side channel and instead the L1 dTLB (and L2 sTLB) are the only TLB components shared between co-resident hyperthreads. We also observe that the number of ways identified in the sTLB actually appears to be double that advertised by the specification. This discrepancy is also consistent with previous work though.



(a) dTLB (b) iTLB, With Hyper-threading. (c) iTLB, W/o Hyper-threading. (d) sTLB.

Figure 8: Identifying set-mapping function for L1 & L2 TLB Components.

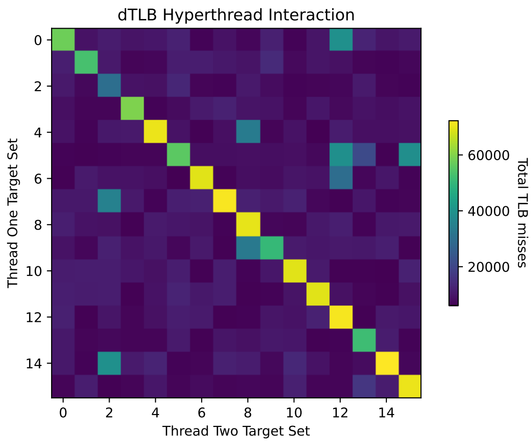
TLB Component	Sets	Ways
dTLB	16	4
iTLB (with hyperthreading)	8	8
iTLB (w/o hyperthreading)	16	8
sTLB	128	12

Table 1: Reverse engineered sets and ways for each TLB component.

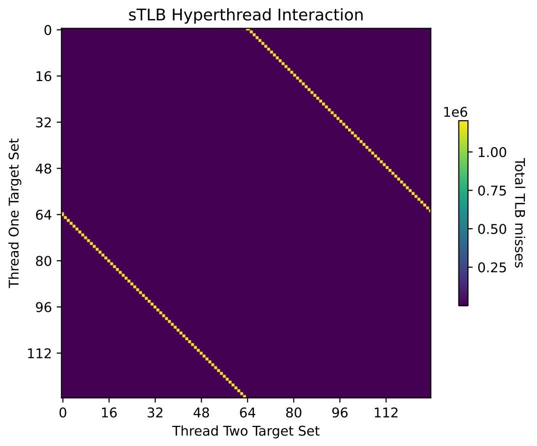
4.4 Hyper-thread Interaction

With an understanding of the sets, ways and mapping function of the TLB we are in a position to craft eviction sets and investigate how two hyper-threads interact in a the TLB. To do this, we initialize two hyper-threads which each allocate a number of virtual pages that fill a given TLB set. The hyper-threads then concurrently access their eviction sets and record any misses. In cases where the two hyper-threads target the same TLB set we expect to observe more misses as they compete for entries within the same set. We repeat this for all possible combinations of sets within the dTLB and sTLB (we have already shown the iTLB implements partitioning). Our results are show in Figure 9. As expected, in the dTLB we observe more misses when the two hyper-threads access a collection of pages mapping to the same set. Interestingly though, we only observe competition for the sTLB when the two targeted sets are offset by 64. This suggests the hash function used by the sTLB also incorporates the ID of a hyper-thread. We conclude the highest bit of the sTLB mapping function is XOR-ed with the hyper-thread ID. This is consistent with previous work [39, 10]. If one thread has $ID = 1$ and the other has $ID = 0$ we can re-define the XOR_N mapping function as follows:

$$XOR_N(VA) = VA[N+13:13] \oplus VA[2N+14:N+14] \oplus ID \cdot 2^N$$



(a) dTLB



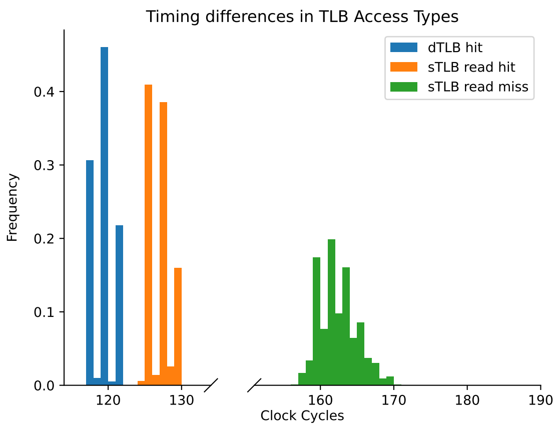
(b) sTLB

Figure 9: Investigating Hyper-Thread Interaction in the TLB.

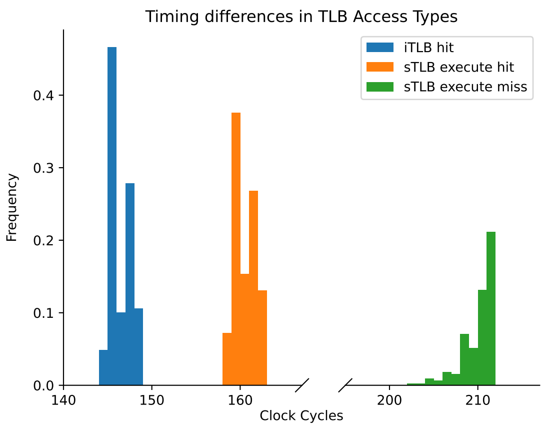
4.5 Timing

Whilst desynchronization is a useful primitive for reverse engineering the underlying hardware, it requires accessing kernel memory such as page tables. This is inconsistent with our threat model which builds on the assumption an attacker implements a TLB side channel using unprivileged access. Instead, to implement the side channel, we require another primitive capable of distinguishing between TLB misses without escalated privileges. In rare cases this could include performance counters, however their availability is target specific. Instead we relying on timing.

To investigate this, we measure the time taken to access virtual addresses with translations at specific levels of the hierarchy. For the L1 TLBs we simply make two accesses of the same type (data or instruction) to a virtual address and time the second. To measure access times of the L2 sTLB we make an access of one type, filling the corresponding L1 TLB and L2 sTLB, and then time a subsequent access of the opposite type. And finally, to measure TLB misses, we first flush the TLB by issuing a large number of distinct accesses, and then time a subsequent access to virtual page not previous used. We use the `rdtsc` and `rdtscp` instructions to take our measurements and serialize accesses using the `lfence` instruction. Our results are shown in Figure 10. We can conclude timing is sufficient to monitor the TLB. Interestingly though, we attribute the higher execution times associated with instruction accesses to the nature of our test code. Whilst data accesses are a simple `MOV` instruction, instruction accesses `CALL` the target virtual address which introduces additional overhead such as pushing the return address onto the stack.



(a) Data Accesses



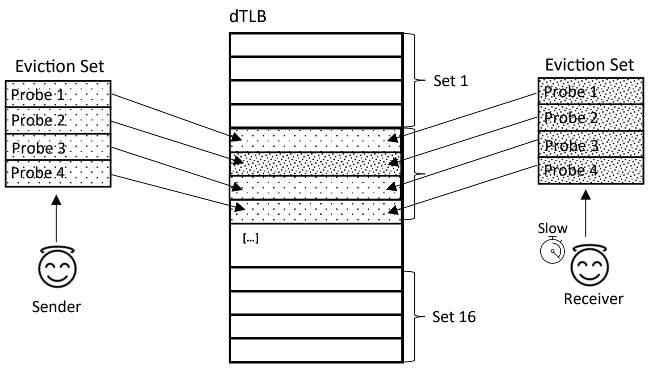
(b) Instruction Accesses

Figure 10: Using Time to Distinguish Between TLB Misses & Hits.

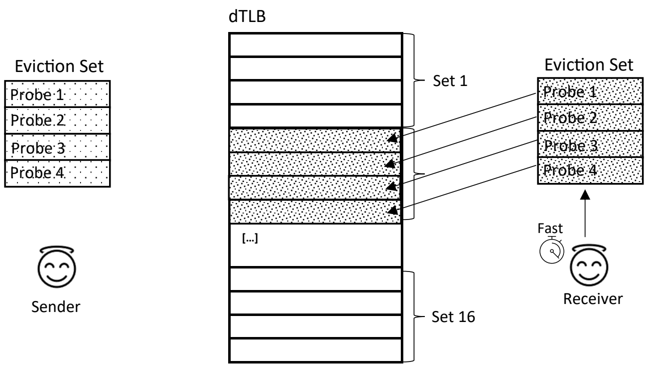
5 Covert Channel

Having reverse engineered the underlying structure of the hardware on our target system we are in a position to leverage the TLB for adversarial objectives. In this section we outline our approach to implementing a covert channel through the TLB. This verifies our reverse engineering efforts and core TLB-related functions which will ultimately be used to stage a side-channel attack in Section 6. Covert channels play an important role in circumstances where parties wish to obfuscate their conversation or are unauthorized to communicate. For example, a common application includes data exfiltration. We implement a covert channel enabling uni-directional communication between two processes using the access latency of virtual pages mapping to a pre-established dTLB set.

Implementation. Inspired by the Prime+Probe cache side channel, the sending and receiving hyper-threads begin by allocating an eviction set mapping to some dTLB set. During communication, the receiver repeatedly records the access time to its entire eviction set. The sender sends a "1" bit by accessing its own eviction set, causing the eviction of the receiver's entries in the TLB and hence a slower access time. The sender sends a "0" bit by simply not touching its eviction set resulting in a faster access time for the receiver. This is visualised in Figure 11. The two hyper-threads synchronize using shared memory, but the protocol could be extended to support synchronization and error detection for a truly isolated covert channel.



(a) Sending a 1 Bit.



(b) Sending a 0 Bit.

Figure 11: Covert Channel Visualisation

As previously outlined, to mitigate the effect of micro-architectural optimizations we serialize all accesses using the `lfence` instruction. Additionally, we implement strict pointer chasing such that each virtual page in an eviction set is required to be processed before the next can be fetched. Finally, to mitigate the impact of the cache, we back all virtual pages in our eviction set with the same underlying physical page. This is summarised in Figure 12 and Listing 1.

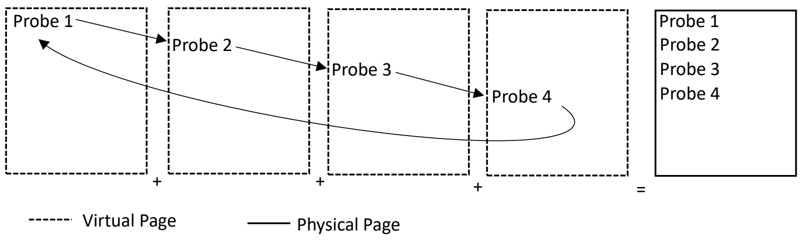
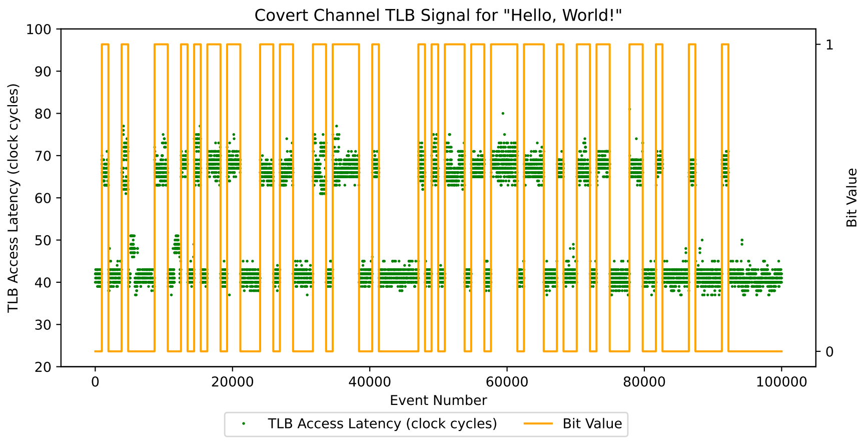


Figure 12: Example dTLB eviction set with 4 entries, using pointer chasing & the same underlying physical page.

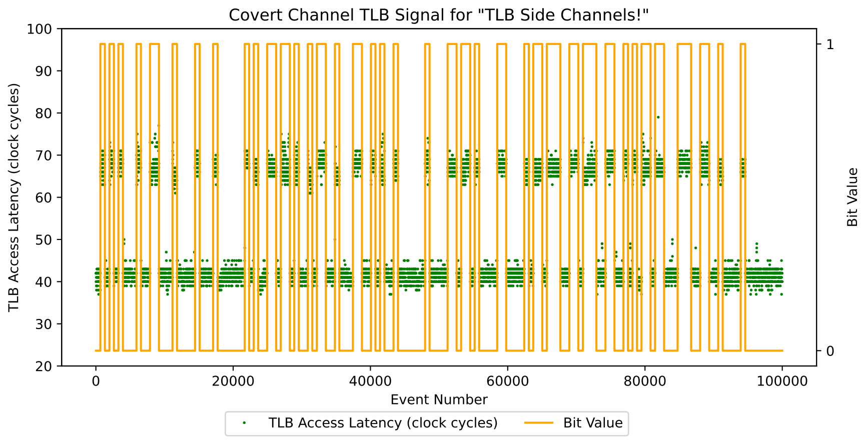
Listing 1 Source to time access to eviction set using pointer chasing.

```
inline __attribute__((always_inline)) unsigned int time_access(unsigned long addr) {
    unsigned int time1, time2;

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

    return time2 - time1;
}
```

Results. Our covert channel is capable of facilitating uni-directional communication even in the presence of heavy noise. Figure 13 depicts example TLB latency signals observed by the receiver during our covert channel. We test our covert channel during the execution of `stress -m 2`, preoccupying two of the four cores on our system. This generates a high degree of activity throughout the machine. Even in this case, our sender successfully transmits 100 bits in 92% of total communications. Interestingly, in the remaining cases our covert channel fails due to synchronization issues between the sender and receiver rather than observing bit flips. This is unsurprising after observing that the introduction of noise only increases the frequency our processes are interrupted and not the TLB access latency. Unlike existing work our covert channel relies on shared memory to synchronize the processes making results incomparable. Nonetheless this work provides a good foundation to understand how to monitor TLB activity which will be necessary for our exploration into side-channel attacks.



(a) Sender transmitting "Hello World!"



(b) Sender transmitting "TLB Side Channels!"

Figure 13: Example TLB Signals observed by the receiver during the covert channel. The "Bit Value" indicator depicts the ground truth of the bit being sent.

6 TLB Side Channels

Having identified the underlying hardware of the target and designed primitives to monitor the TLB we are in a position to investigate in-depth TLB side channel attacks. We start by building on the design of our covert channel to implement a Prime+Probe based attack before exploring an Evict+Time variation.

6.1 Ping Detection

As a starting point, we show that using a TLB side channel a spy can infer the arrival of pings to a co-resident victim process running the `ping` UNIX command. During the execution of `ping` a process is relatively passive outside of receiving a packet. At this point though, the packet is handed from the kernel to the user process, parsed to determine key information and the process outputs the relevant data. Unsurprisingly, we show this spike in activity is reflected in the TLB and can be detected using a side channel.

Implementation. As in the covert channel, the spy process monitors a particular dTLB set using a Prime+Probe based approach. We find that given the volume of activity on receiving a ping any set is sufficient. Concurrently with that, the victim process executes the `ping` command on the co-resident core. Then using a simple moving average over the observed TLB latencies the spy can detect competition for the dTLB set and infer the arrival of a ping. We find this approach results in no false positives or negatives and is capable of pinpointing the arrival of a ping to within the duration the victim is processing the packet. Figure 14 displays the observed TLB signal of the spy with the pink background showing the period the victim is processing a ping packet.

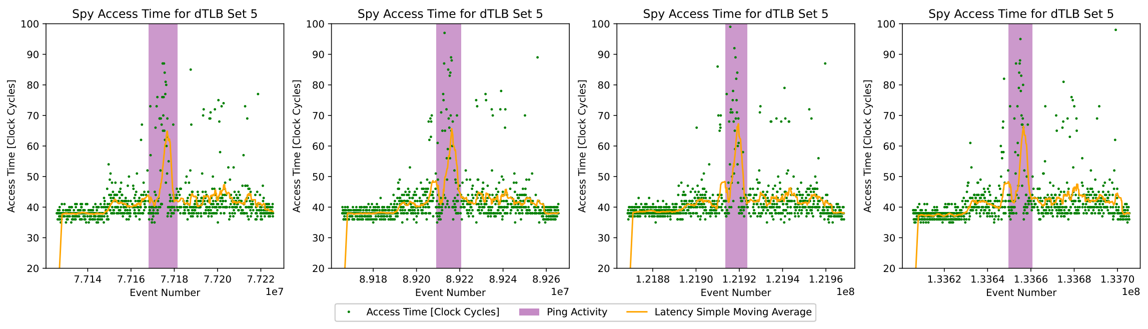


Figure 14: Using a Prime+Probe TLB side channel to infer the arrival of pings to a co-resident victim.

6.2 Command Identification

We next demonstrate the attack surface exposed by the TLB is much broader than perhaps initially expected. In particular, we attempt to generalise the process of using a TLB side channel to discern between a co-resident hyper-thread's execution of different applications and execution paths. We show that the TLB signals leaked by a spy during the victim's execution of a number of UNIX commands can be differentiated.

Implementation. To do this, we design a simple `victim-harness` application capable of initially synchronizing with the spy before launching a given command. We collect the dTLB signals observed by the spy targeting a particular set for a number of commands and plot the results in Figure 15. Observing the signals we see that to the human eye the signals cannot be differentiated. However, we next show that using machine learning approaches we can leak the co-resident application being executed during the observed signal.

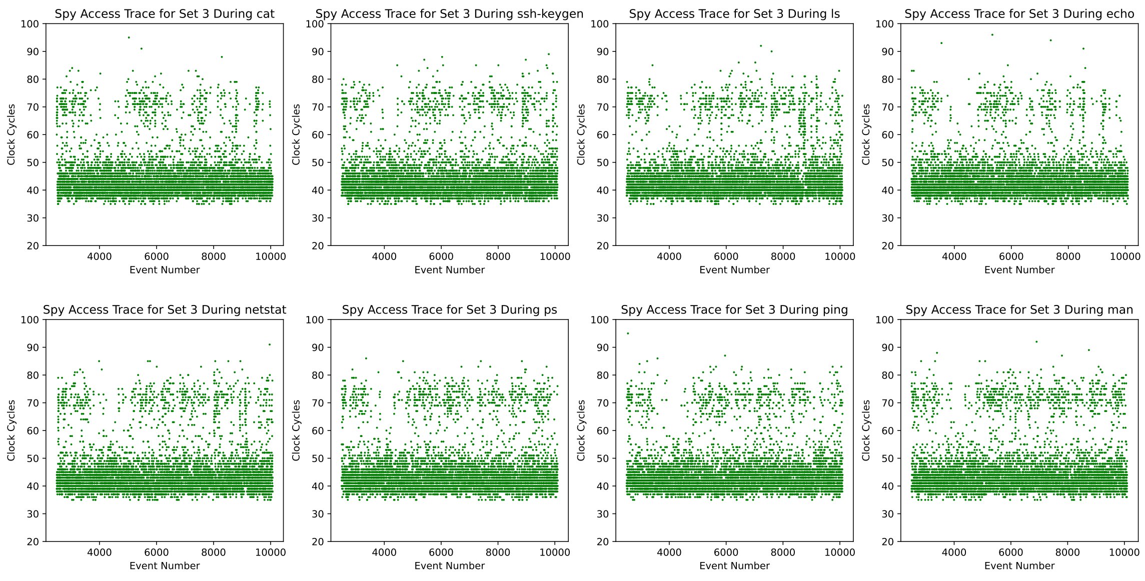
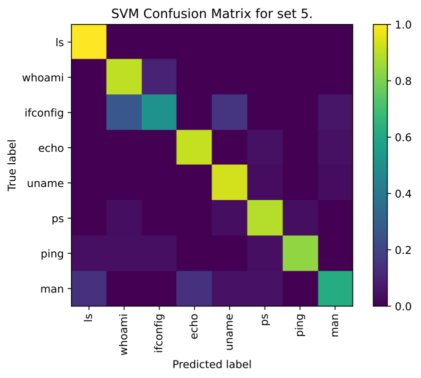


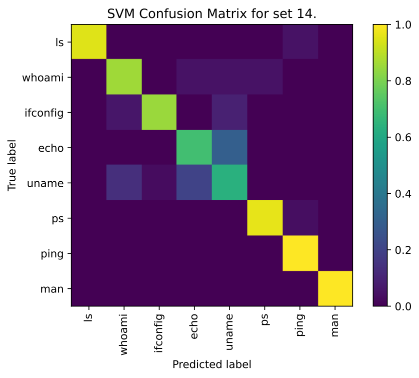
Figure 15: Spy TLB signals observed during co-resident execution of common UNIX commands.

Signal Classification. For the attack, the spy has the option to target their Prime+Probe side channel in any of the dTLB sets. Whilst it might seem obvious that different applications induce a different signal in the TLB, it is important to highlight the *same* application will also induce different signals in different dTLB sets. For now we assume ASLR is disabled. This implies when the spy observes the same set over multiple executions of the same command, the latency signal should be, theoretically, the same. Given this, the attacker aims to identify the source of a signal observed in a particular dTLB set using a pre-trained classifier. To collect sufficient data for a classifier, we record 250 dTLB signals obtained during the execution of each command. We then extract 7500 access latency measurements from each signal to form a sample. To mitigate the impact of noise from initializing execution, we take these samples from a fixed offset after the start of the observed signal. The samples are then used to train a Support Vector Machine (SVM). We repeat this to train a classifier to classify signals obtained from each dTLB set.

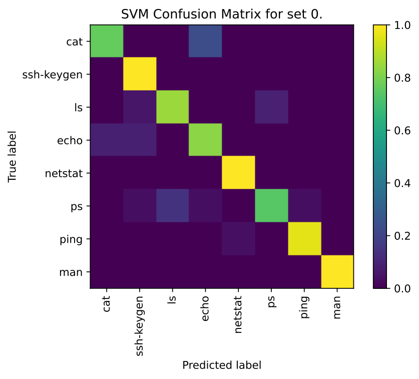
We conclude that with an accuracy of upwards of 75% an attacker is able to differentiate the TLB signals observed during the execution of 8 common UNIX applications. Results of the trained classifiers for particular dTLB sets are shown in Figure 16. We find that using no particular dTLB set for the side channel enables better results than the rest.



(a) dTLB Set 5



(b) dTLB Set 14



(c) dTLB Set 0

Figure 16: Identifying the execution of common UNIX commands using a TLB side channel and SVM classifier.

6.3 Key Extraction

We now demonstrate our implementation of the TLBleed [10] attack on the target system. This involves using a TLB side channel to leak the signature key of a co-resident victim during the execution of Libgcrypt's EdDSA implementation from version 1.6.3. For this we target the execution of the code in Listing 2. This depicts the elliptic curve cryptography (ECC) multiplication function used during a digital signature. We can see that on line 13 the algorithm iterates over the signature key and for each bit executes the `_gcry_mpi_ec_dup_point` function. Additionally though, if the current bit is set, the code calls the `_gcry_mpi_ec_add_points` function. In this attack, we show that the difference in execution between set and unset bits in the key leaves a trace in the TLB that can be leaked using a side channel.

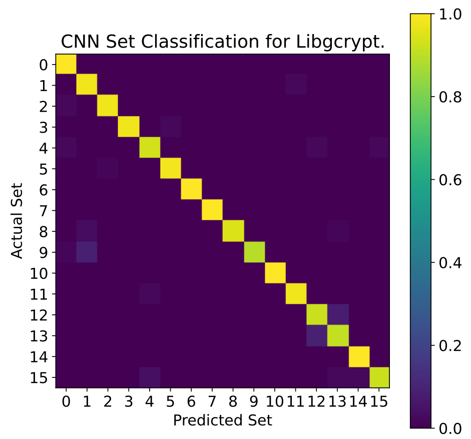
Listing 2 Libgcrypt ECC Multiplication Source.

```c
/* Scalar point multiplication - the main function for ECC.  If takes
   an integer SCALAR and a POINT as well as the usual context CTX.
   RESULT will be set to the resulting point. */
void _gcry_mpi_ec_mul_point (mpi_point_t result, gcry_mpi_t scalar,
                             mpi_point_t point, mpi_ec_t ctx) {
    [...]
    if (mpi_is_secure (scalar)) {
        /* If SCALAR is in secure memory we assume that it is the
          secret key we use constant time operation.  */
        [...]
    } else {
        /* Iterate over bits. */
        for (j = nbits - 1; j >= 0; j--) {
          _gcry_mpi_ec_dup_point (result, result, ctx);

          /* If current bit is set. */
          if (mpi_test_bit (scalar, j))
              _gcry_mpi_ec_add_points (result, result, point, ctx);
        }
    }
}
```

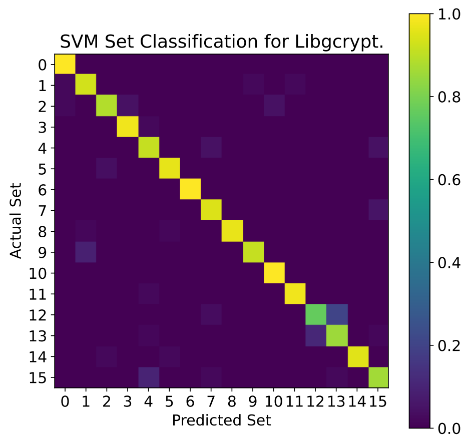
Overcoming Address Space Layout Randomization. ASLR has important implications on the design of our TLB attack for real-world, end-to-end applications. ASLR randomly initializes the base virtual address of applications on each run. This means the access pattern of applications in the TLB is not the same from execution to execution. However, given the linear mapping function implemented by the dTLB, the observed signals for each dTLB set are simply reshuffled on each execution. For example, on one execution of an application the spy may observe a dTLB signal in the second set. However, on the next execution of the same application the same signal may be observed in the fifth dTLB set instead.

To overcome this we implement a two stage process to leaking secret information using a TLB side channel. As a preliminary step we use a pre-trained classifier to determine which signal the spy has observed. In our case the dTLB has 16 sets resulting in 16 possible signals for a single application. In cases where we additionally wish to identify the observed application we require more classes. For our implementation of TLBleed we collect data and train this classifier in a similar approach to that used for victim command identification. In particular, we use samples of 2500 TLB latency measurements taken near the start of a signature trace. We train both an SVM classifier and convolutional neural network

(CNN) and observe similar results. We discuss the architecture of the network later. Importantly, as highlighted in the TLBleed attack, we find normalizing samples is an important step in enabling accurate results. For each sample, we subtract the mean from each latency measurement and divide by the standard deviation. Figure 17 outlines our results. We conclude that with an accuracy of greater than 95% we can differentiate between the signals of each dTLB set, enabling us to overcome ASLR for the exploitation of TLB side channels.



(a) CNN



(b) SVM

Figure 17: Overcoming ASLR for TLB side channel attacks.

Having identified the set of the observed dTLB signal, the attacker can then use another classifier trained for the particular set to leak the desired secret information. In our case, with 16 dTLB sets, we train 16 classifiers. Figure 18 outlines the threat-model and stages of a Prime+Probe TLB side channel, such as TLBleed.

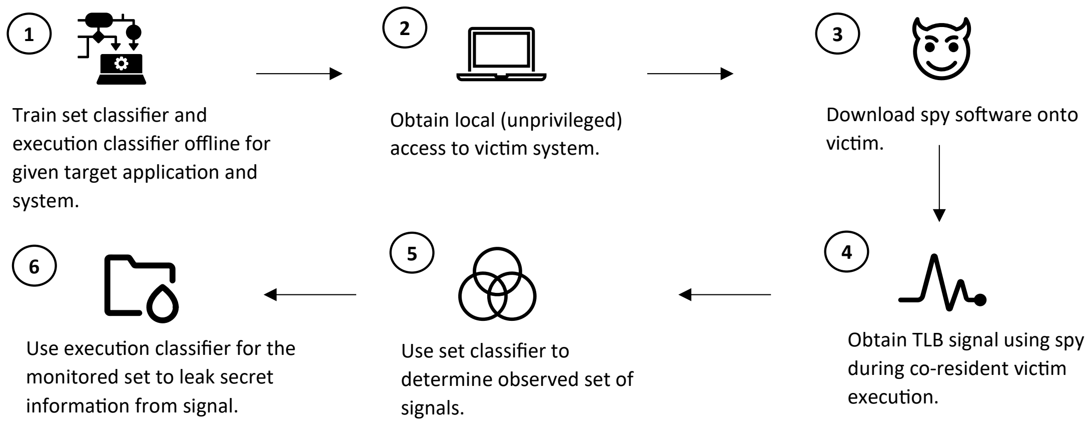

1
Train set classifier and execution classifier offline for given target application and system.
2
Obtain local (unprivileged) access to victim system.
3
Download spy software onto victim.
4
Obtain TLB signal using spy during co-resident victim execution.
5
Use set classifier to determine observed set of signals.
6
Use execution classifier for the monitored set to leak secret information from signal.

Figure 18: Overview of Prime+Probe TLB side channel exploitation / threat model.

Key Extraction Overview. We now demonstrate how an attacker can leak the signature key from a co-resident victim executing the code from Listing 2. To do this the attacker needs to detect the execution of the `_gcry_mpi_ec_dup_point` and `_gcry_mpi_ec_add_points` functions. Given these the attacker can derive the key. As in previous cases, we train a classifier to detect the execution of these functions given the extracted TLB signal. Figure 19 displays an example signal observed by the spy during the execution of the signature algorithm. Given the TLB latencies, the attacker attempts to reconstruct the background.

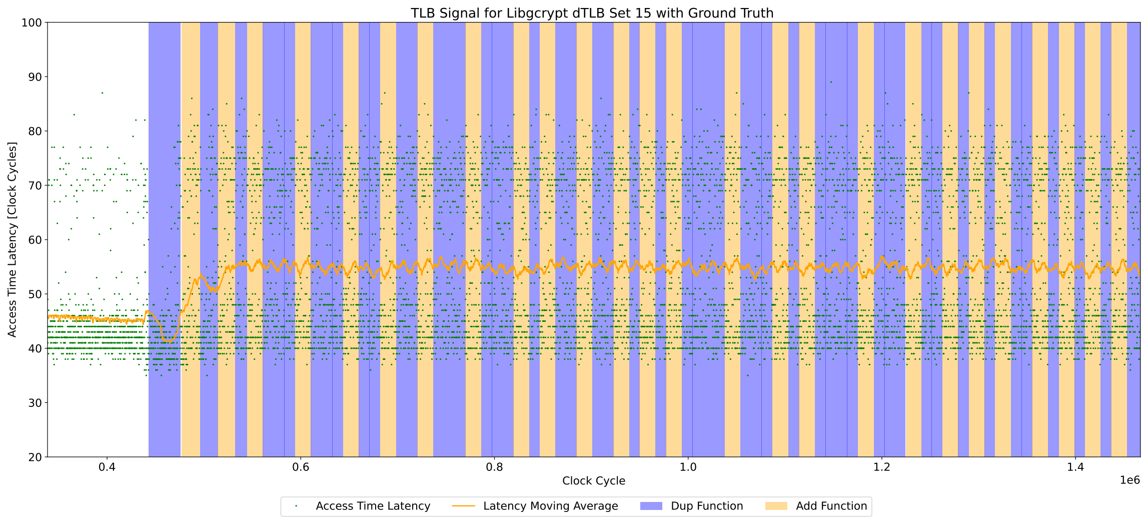


Figure 19: Example dTLB signal observed by the spy during a co-resident victim signature. The background depicts function being executed by the victim. The moving average is included to improve visual inspection of the latency, but is not used by the classification model.

Data Collection. We train the classifier using normalized samples of 150 consecutive TLB latency measurements extracted from the signal. To collect the ground-truth of the samples we augment the victim to record the clock cycle of each `dup` and `add` call which we can then correlate with the clock cycle of each access latency of the spy. Importantly, the code introduced to record the ground truth uses virtual addresses mapping to sets other than the set being monitored by the spy. This prevents additional noise from being introduced into the signal that will not be present in the actual attack.

Each sample of 150 consecutive TLB measurements is classified into one of six classes. Two classes are introduced for samples beginning at the call of either `dup` or `add`. Additionally, given the startup noise of the first two function calls, we also include two additional classes for the first calls in an execution to `dup` and `add`. We include a "neither" / "non-boundary" class for samples taken during the execution of either of the functions and before the iteration over the signing key. And finally, the sixth class is for samples taken at the end of the iteration over the signing key.

We train the classifier over a number of executions. A single signature trace contains 256 `dup` calls, since keys are 256-bits, and on average 128 `add` calls. Additionally, each execution contains only single sample for end of the key iteration and two samples for the first two calls to the relevant functions. On the other hand, the number of "non-boundary" samples we can collect per execution is determined by the frequency of the spy's measurements. In general, the spy records thousands of latencies per signature meaning the proportion of these "non-boundary" samples are much larger than the samples of other classes. To overcome this we include a number of signature traces for training where we do not use samples taken from "non-boundaries".

Network Architecture. We use a CNN (or ConvNet) to classify the signature samples (and for overcoming ASLR as previously discussed). This is a standard machine learning model used in deep learning, particularly suited for time series data. In contrast to traditional deep neural networks, CNNs use convolution operations to extract high-level features. Such networks are capable of identifying unique patterns within inputs even when not occurring at the same location or are varied in structure. This is particularly suited to this application where the signals within the same class are likely to be shifted variations with fluctuating degrees of noise. Other models that could be suited for this work include recurrent neural networks (RNNs) and long-short-term-memory (LSTM) networks. However, our CNN is capable of obtaining a near perfect accuracy and so we do not investigate alternative implementations. Figure 20 visualises the architecture of our CNN.

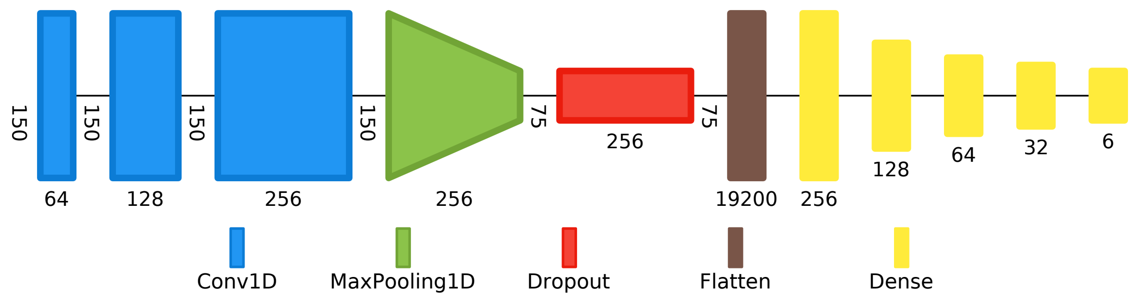


Figure 20: CNN Architecture Visualisation for Libgcrypt Signal Classification Using Net2Vis [2].

Classification Results. We find that our CNN is capable of obtaining near perfect accuracy after being trained on roughly 10 signature executions. The classification results are shown in Figure 21 and an example of a perfectly classified signal is displayed in Figure 22. Unfortunately, even with a near perfect classifier a single signature trace requires classifying thousands of samples depending on the frequency of the spy's measurements. Given this, inevitably a number misclassifications will be made. Similar to the original implementation of TLBleed, we design a simple algorithm for detecting and correcting these mistakes. The most common example are cases where the classifier fails to detect a call to either `dup` or `add`. We detect these instances by measuring the duration of the classified functions and inserting guesses at points where the duration is longer than expected. This approach can also be used to detect cases where the classifier pre-emptively identifies the start of the iteration over the signing key. Additionally, despite introducing separate classes for the first two function calls, the classifier occasionally misses them and so the algorithm inserts guesses at the start of the signal too. Finally, we observe a handful of cases where the spy is interrupted during monitoring the dTLB. This introduces noise into the observed signal and throws off the classifier. These cases can be detected by measuring the duration between each latency recorded by the spy and corrected by inserting guessed function calls. Example visuals of misclassified signals can be found in Figure 26 in Appendix B.

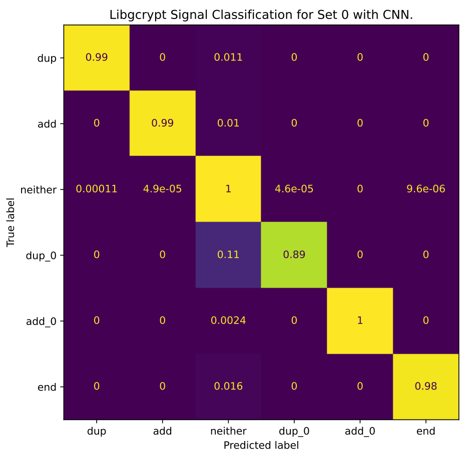


Figure 21: Confusion Matrix for Libgcrypt TLB Signal Classification using a CNN.

Evaluation. To evaluate the performance our the end-to-end side channel we attempt to reconstruct the signature key from the signals of 1000 independent executions. On a quiescent system we successfully extract the key in 95.4% of cases requiring an average of 0.882 classification corrections. In the unsuccessful extractions the reconstructed key and actual key had a bit-wise accuracy of 75%. However, in most of these cases the extracted key was a shifted / offset variation of the actual key derived from an uncorrected misclassification - the position of which could be identified using visual inspection of the classified signal.

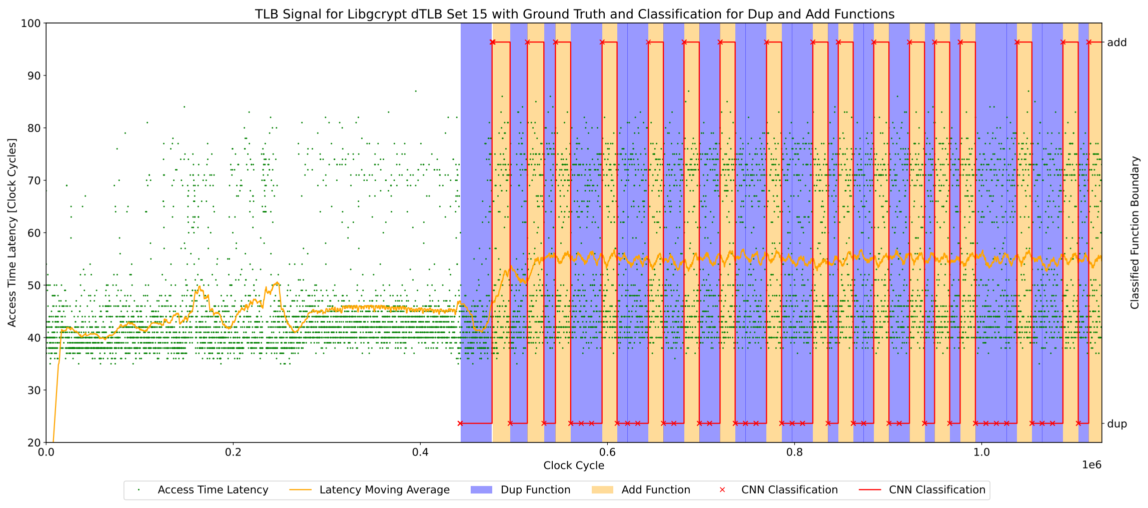


Figure 22: Example of Perfectly Classified dTLB Libgcrypt Signal.

Our results are comparable with the performance of the original implementation of the attack which achieved a success rate 99.3%. We attribute the slight under-performance of our side channel to the implementation of the correction algorithm. Whilst the original work does not discuss the accuracy of their classifier, we believe improving the accuracy of our CNN would not enable better results given the number of samples that are required to be classified per signature. Instead a more robust correction algorithm is required in order to fix the remaining mistakes of our classifier that we could identify through visual inspection in the unsuccessful extractions.

To further test the feasibility of TLB side channels we implement the attack in two additional environments: one in which 25% of the target system's cores are active and another in which half are active. To do this we use the `stress` command with the `-m` parameter. As expected the success rate of the the side channel decreases. In particular, in the case of the lighter load, the proportion of successfully extracted keys drops to 92.3% with an average of 1.425 classification corrections. In the noisiest environment this further drops to 62.4% with an average of 2.304 classification corrections. Interestingly, we observe the largest contributor to unsuccessful extractions are cases where the spy's signal is interrupted. These cases are the hardest for our correction algorithm to recover from given the uncertainty when inserting guess function calls. We hypothesise that the side channel is more resistant to noise on systems with a greater number of cores given the decreased probability of the spy being interrupted. However, this is not something we have been able to test given the lack of availability of hardware. The original TLBleed work does not discuss results of the side channel in noisy environments to compare to.

One limitation of our side channel when compared to the original attack is the overall exploitation time. The use of a CNN in our implementation introduces an additional overhead to the classification time when compared to using a simpler machine learning model. This increases our end-to-end attack to in the range of 1-2 minutes whereas the original work uses an SVM to classify signals within 17 seconds. However, given our threat model and the complexity of the attack, this additional execution time is not particularly relevant since any adversary exploiting a TLB side channel is unlikely to operating within a time-sensitive operation.

A final point to highlight is that both the results listed here and in the original work are obtained using the signal observed during a single signature trace. However, as already outlined, a TLB side channel requires local access to the target system. In these cases it is likely an attacker would be able to obtain signals from multiple executions using the same key. This exponentially improves the success rate of the side channel in real-world applications.

6.4 KASLR De-randomization

Similar to ASLR, KASLR randomizes the location of the kernel image on each initialization. Identifying this position is a key pre-requisite for memory-corruption exploitation within the kernel. In a final example, we demonstrate that TLB signals can be observed across the user-kernel boundary to overcome KASLR. In particular, we show using a TLB side channel it is possible to de-randomize the lowest 7-bits of the kernel image and reduce the entropy of KASLR within kernel modules. This is similar to the TagBleed attack [22] which combines a cache and TLB side channel for a full KASLR break.

Overview. As discussed in Section 2 modern TLBs employ tagging to avoid flushing the TLB on every context switch. However, this introduces side channel leakage. Using an Evict+Time side channel a user process can infer kernel usage of particular sTLB sets. By correlating this activity with kernel accesses we can identify the sTLB set of particular virtual addresses within the kernel. We next explain how this is useful for reducing the entropy of KASLR.

Most modern systems back the kernel using 2MB huge pages and randomize the lowest 9 bits after the 21-bit page offset. As reverse engineered for the TagBleed attack, an sTLB using an XOR_N hash function for 4KB pages determines the set for 2MB huge pages using a linear function of the lowest N bits in the virtual page number. This implies identifying the mapped sTLB set of a huge page is sufficient to infer the lowest N bits of the virtual address. In contrast, kernel modules are backed using 4KB pages and so identifying the mapped sTLB set is not sufficient to deduce the values of any bits. However, doing so still places constraints on the randomized bits. This is summarised in Figure 23.

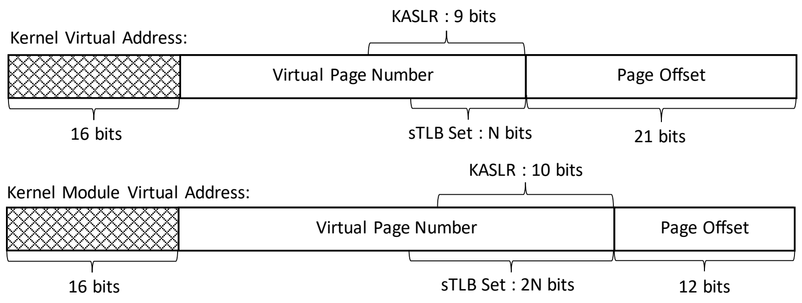


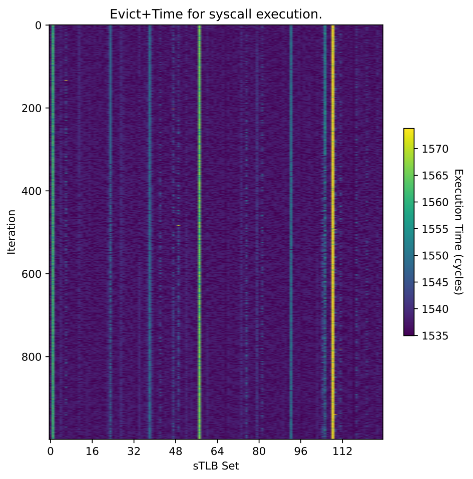
Figure 23: Relation between randomized bits of KASLR and mapped sTLB set using XOR_N hash function. Typically $N \in \{7,8\}$.

Implementation. With an understanding of how we reduce the entropy of KASLR by knowing the sTLB of a kernel virtual address, we next show how this is possible. Listing 3 outlines the implementation. We first start by filling the TLB with entries used in the kernel by accessing the kernel a number of times. We then remove all entries from the L1 TLBs and the entries within a particular sTLB set. Finally we time the execution time of a subsequent access to the kernel. We repeat this for a number of iterations and for all sTLB sets. Depending on the evicted sTLB set, the execution time will vary. For example, Figure 24a displays our results when triggering kernel access using an arbitrary syscall. From the results we can infer that the syscall uses a number of pages mapping to sTLB sets 110 and 53. By comparing the results of carefully selected kernel accesses we can identify the mapped sTLB set of a particular kernel symbol.

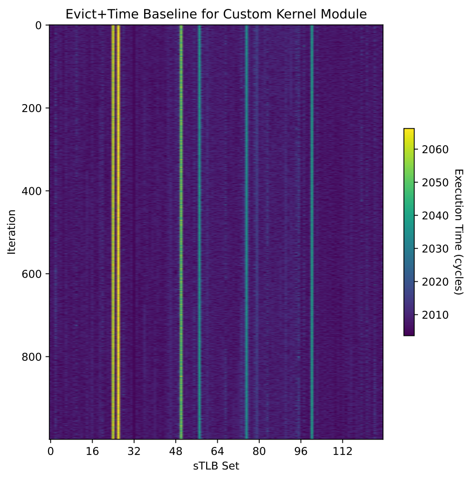
Listing 3 Pseudocode for Evict+Time TLB Side Channel Algorithm.

```
for each sTLB_set:
    for each warm_up:
        kernel_access()

    evict_l1_tlb()
    evict_stlb_set(sTLB_set)

    start = time()
    kernel_access()
    end = time()

    results[sTLB_set] = end - start
```

Results. To demonstrate the feasibility of such an attack we implement a kernel module which on each load initializes a target variable at a random address. Using the Evict+Time side channel we identify the sTLB set which the variable



(a) Syscall



(b) (Custom) Kernel Module

Figure 24: Evict+Time results for syscall and (custom) kernel module access.

maps to. We use the `ioctl` system call to communicate with the module. To be able to distinguish between noise associated with normal access to the module and not the target we first collect a baseline set of results. To do this, we use the Evict+Time side channel and communicate with the module without accessing the target. Figure 24b shows the results. Next we collect a set of results for accesses to the module which then writes to the target variable. By subtracting the two sets of results we can isolate the additional noise introduced into the second signal as a result of accessing the target variable. We repeat this for a number of different initializations of the module. Figure 25 shows the results for three initializations. We find that in all cases we are able to identify the sTLB set of the target variable.

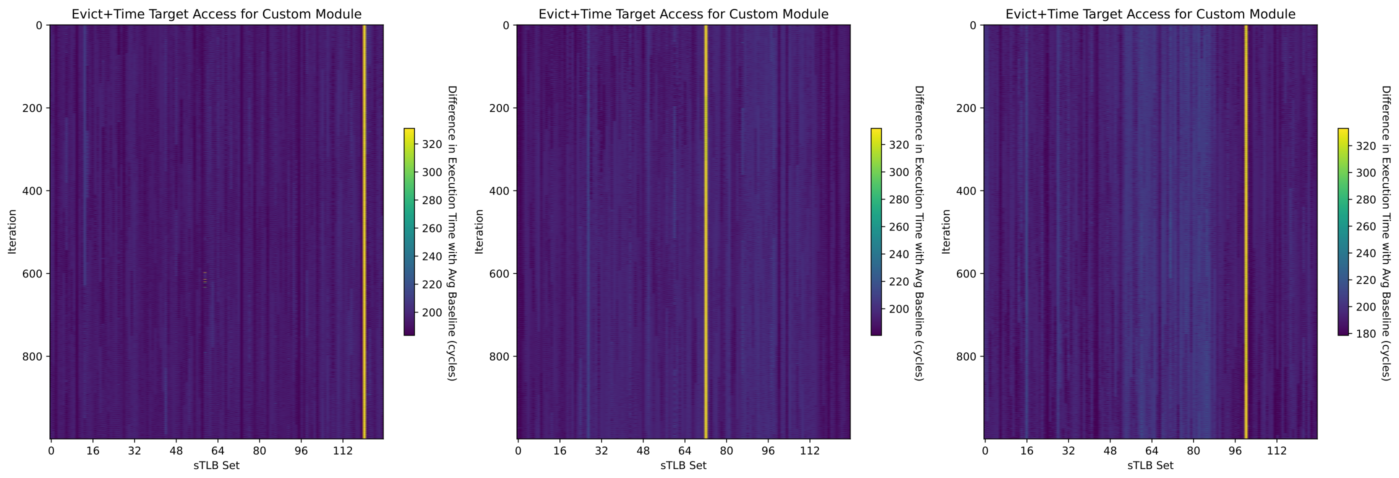


Figure 25: Evict+Time results for three initializations of custom kernel module. In each case we can identify the sTLB set of the target variable.

7 Mitigations

Having demonstrated the feasibility of TLB side channels, it is important to consider potential defences to mitigate their impact. In this section we explore a number of proposed approaches to prevent TLB leakage.

Extending Existing Cache Defences. Perhaps the most obvious mitigations for TLB side-channel attacks extend existing cache-based defences to the TLB. This includes implementing better isolation in the TLB to prevent side channels relying on concurrent access between the attacker and victim. As with the cache, this could be implemented either by partitioning the TLB by sets or ways. To isolate by ways this could be done at the hardware level by re-implementing Intel's Cache Allocation Technology (CAT) [26] in the TLB. In contrast, similar to the cache, it could be possible to partition the TLB by sets at the OS level by managing the virtual address space of hyper-threads [10]. However, this is far from an ideal solution since processes rely on contiguous virtual memory as implemented currently. Another sub-optimal solution isolates the TLB simply by disabling hyper-threading. Whilst this prevents Prime+Probe based side channels, such as TLBleed [10], this wastes resources and again only addresses attacks relying on concurrent TLB access. In a similar vein, simply flushing the TLB on each context switch prevents a subset of side channels, such as Evict+Time based approaches like TagBleed [22], at the cost of performance. Other examples of cache based defences that could be extended to the TLB include timer restrictions and TSX as discussed in Section 2.

Secure TLB Design. In contrast to the previously mentioned approaches, a number of proposed secure TLB designs attempt to address the problem at the lowest level [7]. Inspired the iTLB's dynamic partitioning and Intel's CAT, the Statically-Partitioned TLB (SP-TLB) aims to isolate processes in the TLB by preventing two hyper-threads from using the same ways within a set. However, as previously highlighted, isolation fails to prevent all classes of TLB side channels. Instead, similar to the Random-Fill Cache (RF-Cache) [25], the RF-TLB introduces non-determinism into the replacement policies implemented by the TLB. This de-correlates memory accesses patterns from the entries filled in the TLB. More specifically, on particular TLB misses the replacement policy passes the corresponding physical address directly to the CPU and fills an entry in the TLB with a random translation. This approach introduces randomness into any timing-based TLB signal, complicating secret extraction dramatically. Unsurprisingly though, the authors demonstrate both designs additionally introduce a performance overhead. In the case of the RF-TLB, this is a 9% penalty when compared to traditional set-associate TLBs. Given recent success in overcoming randomized cache architectures [35, 3], the improved TLBCoat implementation [38] proposes a more robust randomized design than the RF-TLB. Other randomization techniques such as shuffler [42] periodically randomizes the locations of symbols within the virtual address space. This can help prevent against longer-running TLB side channels.

8 Future Work

This project has demonstrated a number of concrete TLB side channels exploiting specific applications. However, the attacks described are potentially much further reaching. Compared to cache side channels, the novelty of TLB attacks means there exist many open questions and opportunities to build on existing work.

Browser De-randomization. One avenue we began to purse was implementing a TLB side channel for reducing the entropy of ASLR in the browser. In demonstrating a TLB side channel could be exploited remotely, such a feat would have implications on the understood attack surface exposed by the TLB. However, this was not something we were able to achieve given the time constraints of the project and additional complexities in the attack. Working at a much higher level of abstraction introduces a number of unique complications not yet explored in local TLB attacks. Most obviously this includes the greater degree of noise present in a signal obtained using a side channel implemented in JavaScript compared to C. The lack of native support for precise timers additionally introduces difficulties, although previous has work has overcome this for cache side channels in the browser [9]. And finally, it is unclear how an implementation in JavaScript could de-randomize ASLR when it lacks support for pointers and explicit memory allocation as facilitated by C. Whilst there could exist novel JavaScript functionality to implement similar behaviour, we briefly discuss another direction to explore which we suspect could also overcome this complication.

Without the ability to allocate pages at specific virtual addresses it becomes difficult to construct eviction sets for the TLB and manipulate the TLB state. However, we speculate it could be possible to identify the address of an attacker-allocated page by leveraging the non-linearity of the hash function implemented in the L2 sTLB. With this, it would be possible to construct eviction sets and implement an attack, similar to that described in Section 6.4, to identify the virtual address of a particular symbol within the browser. We assume the attacker has the ability to allocate a large buffer of virtually-contiguous pages. Then, given the non-linearity of the XOR_N function and depending on the base address of the buffer, there are a limited number of collections of offsets from the base that could form an sTLB eviction set. The attacker can test the likelihood of each collection forming an eviction set with the following experiment. First, they access one offset in the collection of pages that potentially map to the same sTLB set. They then subsequently access a total of w other offsets in same collection (where w is the number of sTLB ways). Finally, they then time a subsequent access to the original offset. We expect the longest access times to occur in cases where all offsets map to the same sTLB set. Knowing which offsets correspond to sTLB eviction sets places constrains on the virtual address of the buffer.

Further Applications. The evolution of traditional cache attacks saw such side channels being used to exploit less conventional environments [31, 37]. We expect TLB side channels to equally follow-suit and could be used in, for example, virtualized settings. Other target applications, aside from encryption protocols, browsers and the kernel, could include attacks compromising user privacy. For example, building-off our ping detection and command identification work, we expect TLB side channels could used be to leak other behaviours of co-resident hyper-threads. This could include extending cache-based input identification attacks to the TLB [14].

Alternate TLB Architectures. As highlighted, the underlying design of the TLB on the target system is particularly important in implementing a side channel. This suggests different architectures will involve different complications during a TLB attack. For example, AMD architectures implement fully associative L1 TLBs with separated data and instruction L2 TLBs [39]. This introduces difficulties in manipulating TLB state and has so far prevented TLB side channels, such as TLBleed, being implemented on AMD. In another example, the TLB on the M1 architecture implements two independent L1 iTLB components, one for user space and one for kernel space [36]. Additionally, the TLB is implemented as a three level hierarchy with the iTLBs at the lowest level, the dTLB at the next level and an sTLB at the highest level.

Further Attack Implementations. Previous work has attempted to model all possible timing-based TLB attack approaches [7]. However, the two concrete TLB attacks that had been demonstrated prior to the authors' publication [10, 15] covered only eight of the 24 possibilities. Since then, the TagBleed attack

[22] demonstrated the feasibility of an additional two. This leaves a remaining 14 approaches yet to be implemented concretely and includes extending known cache-based side channels, such as Evict+Probe and Prime+Time, to the TLB. In addition to the unexplored approaches, variants of existing implementations are possible. For example, recent work investigated the impact of using a pair of dTLB and sTLB sets simultaneously for a Prime+Probe TLB side channel [39]. These variations are not included in the originally theorised 24 TLB attacks, opening the door to many other attacks.

9 Conclusion

This project has demonstrated the feasibility of TLB side-channel attacks and consequently the need to implement new defences protecting the TLB. We have achieved comparable results to the TLBleed and TagBleed attacks to provide concrete examples against encryption algorithms and the kernel. Additionally, we have proposed new applications of TLB attacks, including for breaking user privacy, by implementing side channels for ping detection and command identification. We have outlined a number of approaches for mitigating the impact of these attacks, highlighting that no solution is without its downsides. However, our work emphasises that, despite the performance overhead incurred by potential mitigations, new micro-architectural defences are necessary. We concluded by exploring the future direction of TLB side channels, proposing further work to be investigated.

10 Project Management

Retrospect & Challenges. In completing the project, the most notable takeaway has been deeper appreciation for the many levels of abstraction that exist in modern systems. Working at the lowest levels of the software stack has meant on many occasions our code did not produce the expected results. In these cases, even reviewing the compiled assembly was often not sufficient to uncover an explanation for the observed discrepancies. Instead, the project has demanded an ability to think critically about the underlying workings of high-level software to construct hypotheses and design experiments to validate them. This has highlighted the importance of having a holistic understanding of complex systems.

Timeline. Given the nature of research projects, devising a thorough timetable proved unproductive. Instead, we identified a more general timetable was appropriate. For this we set rough objectives to achieve by particular points in the project. This included the completion of reverse engineering, implementation of the covert channel, extraction of signals and obtaining end-to-end attack results. The final timetable of the project is depicted in Appendix C.

Ethics. The project raises no ethical, social or professional issues. Additionally, since any software and tools used during the project were open-source and the security vulnerabilities explored are previously known, there are no legal considerations. Potential mitigations for the side channels have been discussed and whilst new attack scenarios have been demonstrated, the vulnerabilities enabling our work have been previously disclosed to engineers of the relevant software and hardware.

Author's Assessment of the Project. We consider the project to be a success; all objectives of the project have been achieved. In our research, we have come to understand the cutting-edge in micro-architectural side channels. Additionally, in implementing concrete side channel examples, including proposing new attack scenarios, we have drawn on a number of specialised areas within computer science such as hardware security, computer architecture and deep learning. We hope our work demonstrates that the attack surface exposed by the increasing complexity of modern micro-architectures should not be underestimated.

References

[1] Adam J Aviv et al. "Smudge attacks on smartphone touch screens". In: *4th USENIX Workshop on Offensive Technologies (WOOT 10)*. 2010.

[2] Alex Bäuerle, Christian van Onzenoodt, and Timo Ropinski. "Net2Vis – A Visual Grammar for Automatically Generating Publication-Tailored CNN Architecture Visualizations". In: *IEEE Transactions on Visualization and Computer Graphics* 27.6 (2021), pp. 2980–2991. DOI: 10.1109/TVCG.2021.3057483.

[3] Thomas Bourgeat et al. "CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches". In: Oct. 2020, pp. 1110–1123. DOI: 10.1109/MICRO50266.2020.00092.

[4] Benjamin Braun, Suman Sekhar Jana, and Dan Boneh. "Robust and Efficient Elimination of Cache and Timing Side Channels". In: *ArXiv* abs/1506.00189 (2015).

[5] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, 991–1008. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[6] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters". In: *Applied Soft Computing* 49 (2016), pp. 1162–1174. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2016.09.014. URL: https://www.sciencedirect.com/science/article/pii/S1568494616304732.

[7] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. "Secure TLBs". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 346–359. ISBN: 9781450366694. DOI: 10.1145/3307650.3322238. URL: https://doi.org/10.1145/3307650.3322238.

[8] Craig Disselkoen et al. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 51–67. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen.

[9] Ben Gras et al. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *Network and Distributed System Security Symposium*. 2017.

[10] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 955–972. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, 2015, pp. 897–912. ISBN: 9781931971232.

[12] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. San Sebastián, Spain: Springer-Verlag, 2016, pp. 279–299. ISBN: 9783319406664. DOI: 10.1007/978-3-319-40667-1_14. URL: https://doi.org/10.1007/978-3-319-40667-1_14.

[13] Daniel Gruss et al. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 217–233. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss.

[14] T. George Hornby. "Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD". In: 2016.

[15] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 191–205. DOI: `10.1109/SP.2013.23`.

[16] Emilia Kasper and Peter Schwabe. *Faster and Timing-Attack Resistant AES-GCM*. Cryptology ePrint Archive, Paper 2009/129. `https://eprint.iacr.org/2009/129`. 2009. URL: `https://eprint.iacr.org/2009/129`.

[17] Mehmet Kayaalp et al. "A High-Resolution Side-Channel Attack on Last-Level Cache". In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450342360. DOI: `10.1145/2897937.2897962`. URL: `https://doi.org/10.1145/2897937.2897962`.

[18] Yoongu Kim et al. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 9781479943944.

[19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002`.

[20] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES". In: *Topics in Cryptology – CT-RSA 2008*. Ed. by Tal Malkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 187–202. ISBN: 978-3-540-79263-5.

[21] Esmaeil Mohammadian Koruyeh et al. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. URL: `https://www.usenix.org/conference/woot18/presentation/koruyeh`.

[22] Jakob Koschel et al. "TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2020, pp. 309–321. DOI: `10.1109/EuroSP48549.2020.00027`.

[23] Moritz Lipp, Daniel Gruss, and Michael Schwarz. "AMD Prefetch Attacks through Power and Time". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 643–660. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/lipp`.

[24] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[25] Fangfei Liu and Ruby B. Lee. "Random Fill Cache Architecture". In: *Princeton Architecture Laboratory for Multimedia and Security (PALMS)* (). URL: `http://palms.princeton.edu/system/files/%5C%25232_liu.pdf`.

[26] Fangfei Liu et al. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 406–418. DOI: `10.1109/HPCA.2016.7446082`.

[27] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy* (2015), pp. 605–622.

[28] Giorgi Maisuradze and Christian Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: Oct. 2018, pp. 2109–2122. DOI: `10.1145/3243734.3243761`.

[29] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 118–129. ISBN: 9781450316422.

[30] Robert Martin, John Demme, and Simha Sethumadhavan. "TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks". In: *SIGARCH Comput. Archit. News* 40.3 (June 2012), pp. 118–129. ISSN: 0163-5964. DOI: `10.1145/2366231.2337173`. URL: `https://doi.org/10.1145/2366231.2337173`.

[31] Clémentine Maurice et al. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. Vol. 17. 2017, pp. 8–11.

[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-32648-9.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. *Cache attacks and Countermeasures: the Case of AES*. Cryptology ePrint Archive, Paper 2005/271. `https://eprint.iacr.org/2005/271`. 2005. URL: `https://eprint.iacr.org/2005/271`.

[34] Mathias Payer. "HexPADS: A Platform to Detect "Stealth" Attacks". In: *Engineering Secure Software and Systems*. 2016.

[35] Antoon Purnal et al. "Systematic Analysis of Randomization-based Protected Cache Architectures". English. In: *42th IEEE Symposium on Security and Privacy*. 42th IEEE Symposium on Security and Privacy, IEEE SP 2021 ; Conference date: 20-05-2021 Through 21-05-2021. May 2021.

[36] Joseph Ravichandran et al. "PACMAN: Attacking ARM Pointer Authentication with Speculative Execution". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022. ISBN: 9781450386104. DOI: `10.1145/3470496.3527429`. URL: `https://doi.org/10.1145/3470496.3527429`.

[37] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. "Inferring Firewall Rules by Cache Side-channel Analysis in Network Function Virtualization". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 1798–1807. DOI: `10.1109/INFOCOM41043.2020.9155449`.

[38] Florian Stolz et al. *Risky Translations: Securing TLBs against Timing Side Channels*. Cryptology ePrint Archive, Paper 2022/1394. `https://eprint.iacr.org/2022/1394`. 2022. URL: `https://eprint.iacr.org/2022/1394`.

[39] Andrei Tatar et al. "TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 989–1007. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/tatar`.

[40] Wenhao Wang et al. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2421–2434.

[41] Mario Werner et al. "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692. ISBN: 978-1-939133-06-9. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/werner`.

[42] David Williams-King et al. "Shuffler: Fast and Deployable Continuous Code Re-Randomization". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 367–382. ISBN: 978-1-931971-33-1. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king`.

[43] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 9781931971157.

[44] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". In: *Journal of Cryptographic Engineering* 7.2 (June 2017), pp. 99–112. ISSN: 2190-8516. DOI: 10.1007/s13389-017-0152-y. URL: https://doi.org/10.1007/s13389-017-0152-y.

[45] Yuval Yarom et al. *Mapping the Intel Last-Level Cache*. Cryptology ePrint Archive, Paper 2015/905. https://eprint.iacr.org/2015/905. 2015. URL: https://eprint.iacr.org/2015/905.

[46] Yinqian Zhang and Michael K. Reiter. "DüPpel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 827–838. ISBN: 9781450324779. DOI: 10.1145/2508859.2516741. URL: https://doi.org/10.1145/2508859.2516741.

[47] Zhi Zhang et al. "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 28–41. DOI: 10.1109/MICRO50266.2020.00016.

[48] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 871–882. ISBN: 9781450341394. DOI: 10.1145/2976749.2978324. URL: https://doi.org/10.1145/2976749.2978324.

Appendices

A Target System Specification

CPU Model: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

CPU(s): 4

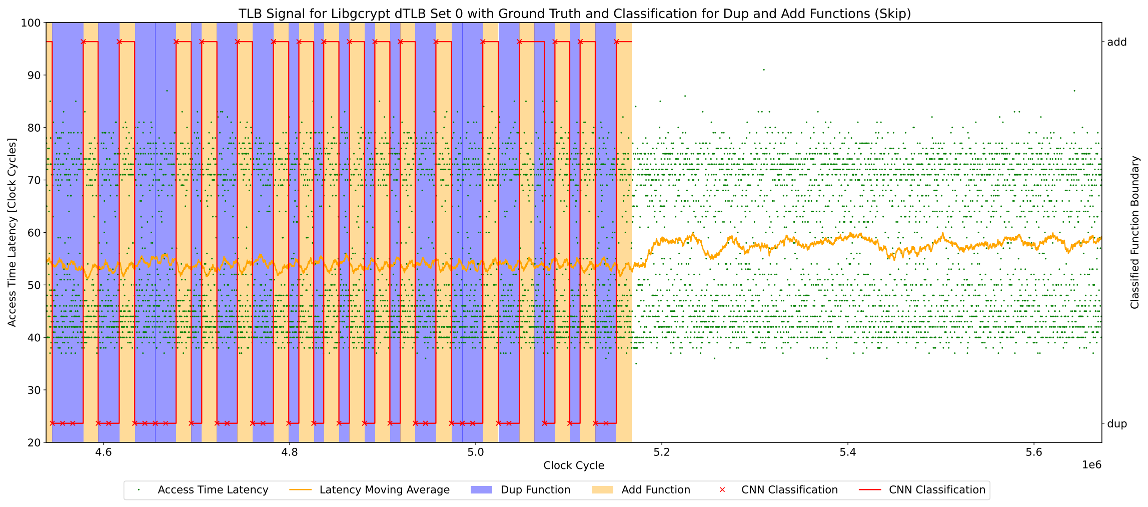
Thread(s) per core: 2

TLB:

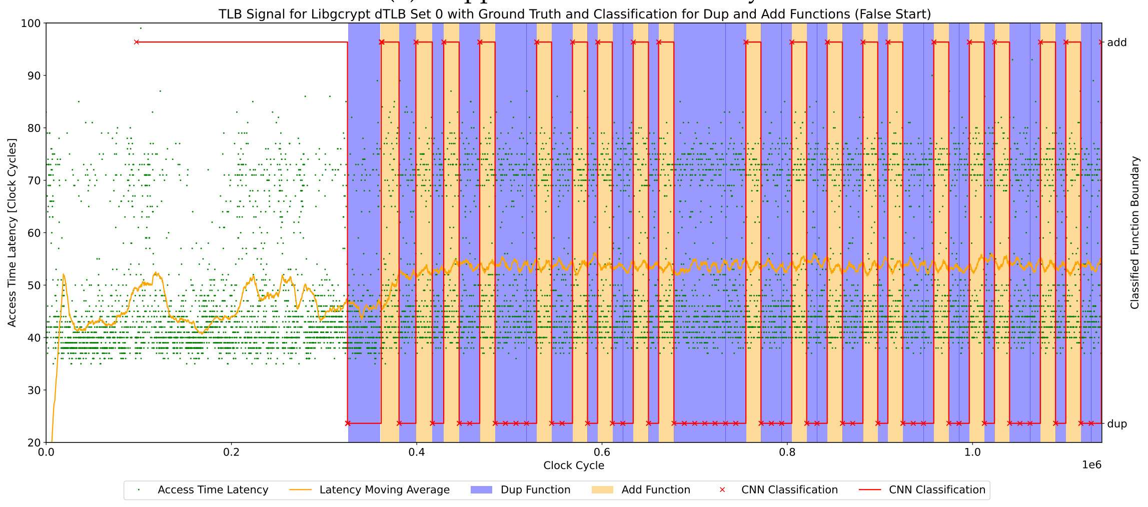
Instruction TLB: 4K, 8-way, 64 entries[1]
Instruction TLB: 2M/4M pages, fully, 8 entries
Data TLB: 4K pages, 4-way, 64 entries
Data TLB: 2M/4M pages, 4-way, 32 entries
Data TLB: 1G pages, 4-way, 4 entries
L2 TLB: 4K/2M pages, 6-way[2], 1536 entries

[1] Depending on co-resident hyper-thread activity.
[2] Reverse Engineering efforts demonstrated 12-way sets.

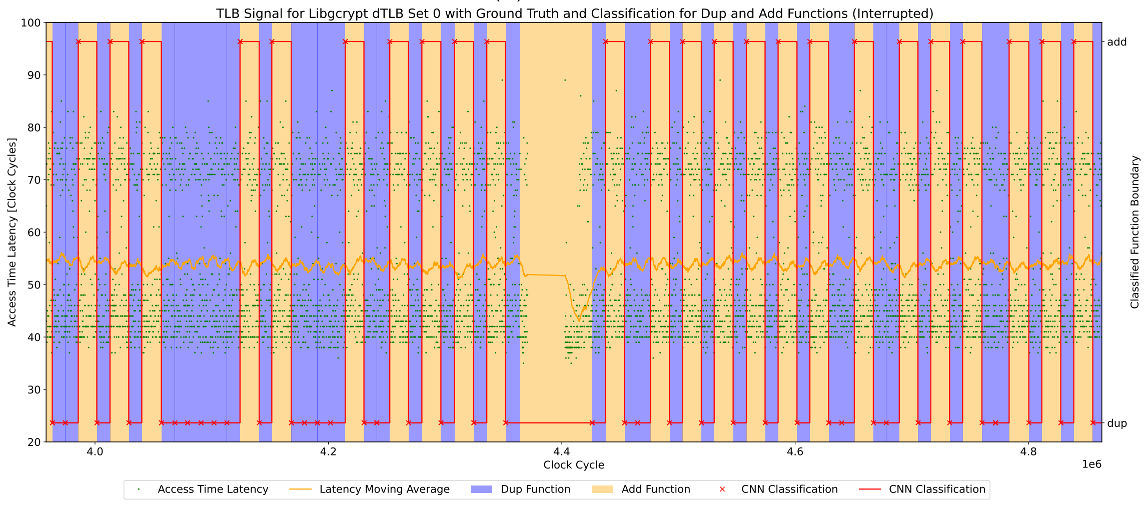
B Misclassification Examples



(a) Skipped Function Boundary



(b) False Start



(c) Interrupted Signal

Figure 26: Example misclassifications to correct for key extraction.

C Timetable

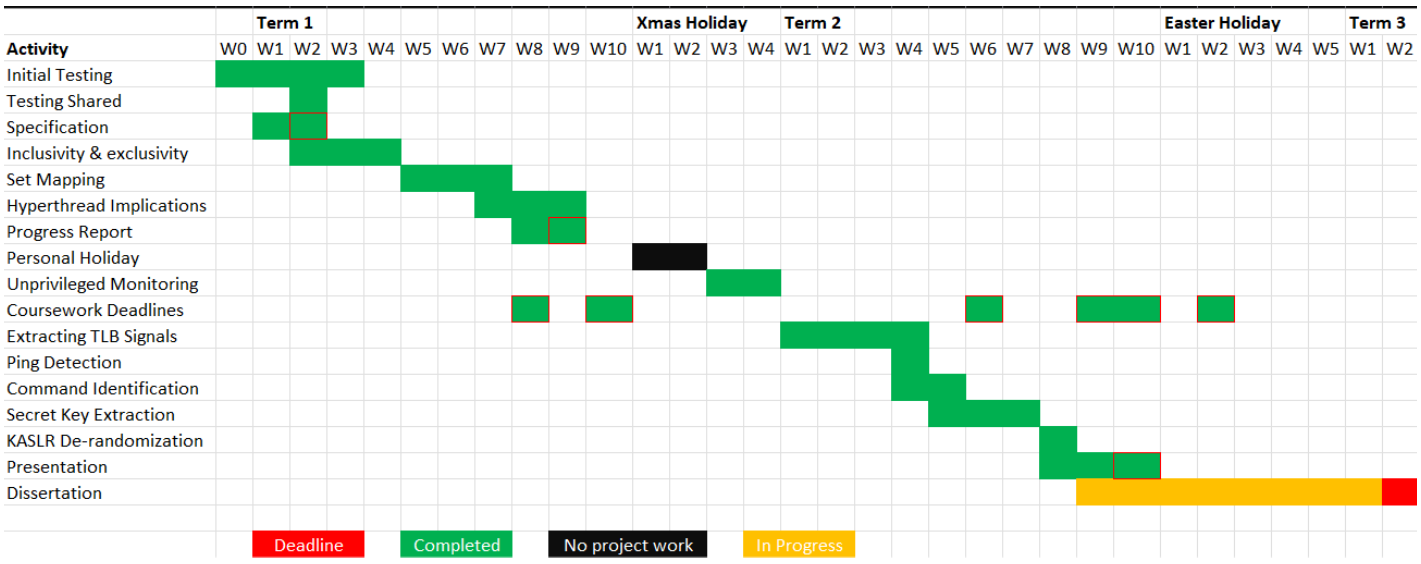


Figure 27: Final timetable for the project represented as a Gnatt chart.