Karlsruhe Institute of Technology

High-Performance TLB Covert Channels

Bachelor's Thesis
submitted by

Micha Hanselmann

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Dr.-Ing. Marc Rittinghaus

13. April 2020 – 15. September 2020

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 1, 2020

Deutsche Zusammenfassung

Die zunehmende Bedeutung breit angelegter Virtualisierungslösungen in der Cloud stellt große Herausforderungen an deren Sicherheit. Verschiedene Forschungsarbeiten haben bereits gezeigt, dass geteilte Hardware-Ressourcen (z. B. Caches) das Durchbrechen der Isolation zwischen Prozessen und virtuellen Maschinen ermöglichen. Dabei geriet mit TLBleed [18] der Translation Lookaside Buffer (TLB) in den Fokus, welcher nicht von modernen Sicherheitsmechanismen wie Intel's Cache Allocation Technology (CAT) berücksichtigt wird.

Die grundsätzliche Realisierbarkeit von TLB-basierten Covert Channels veranlasst uns, deren Performanz hinsichtlich Bitrate und Zuverlässigkeit deutlich zu steigern, um auf die Notwendigkeit ganzheitlicher Isolationskonzepte auf Mikroarchitekturebene hinzuweisen. Dazu entwerfen wir ein zweischichtiges Kommunikationsprotokoll zur Lösung des Synchronisationsproblems sowie zur Absicherung gegenüber Störungen aufgrund nebenläufiger Prozesse. Ferner präsentieren wir ein neues Verfahren zur Überwachung von TLB Einträgen mittels Auswertung der Zugriffsbits in Seitentabellen. Wir erreichen fehlerfreie Übertragungen bei einer Bitrate von bis zu 200 kB/s in einer Linux KVM Umgebung mit aktueller Intel Hardware.

Abstract

The ongoing global trend towards large-scale cloud virtualization raises concerns on how secure these systems are. Previous work has shown how shared hardware resources (e.g., caches) can be exploited to break isolation between processes and virtual machines. With TLBleed [18], the Translation Lookaside Buffer (TLB) was identified as a new attack vector which is immune to state-of-the-art security mechanisms such as Intel's Cache Allocation Technology (CAT).

Given the general feasibility of TLB-based covert channels, we aim to considerably increase the performance of TLB-based covert channels in terms of channel bit rate and reliability, thereby demonstrating that holistic techniques for microarchitectural resource partitioning are needed. Therefore, we design a two-layer communication protocol capable of dealing with the issues of synchronization and noise due to concurrently running processes. Furthermore, we present a novel approach to monitor TLB entries by leveraging the accessed bit in page table entries. We are able to achieve error-free transmissions at bit rates of up to 200 kB/s in a Linux KVM environment running on current Intel hardware.

Contents

Chapter 1

Introduction

Most servers nowadays run in cloud data centers. Due to the hosting companies' interest to keep costs at a minimum, multiple virtual machines (VMs) are consolidated per physical machine [6]. Consequently, hardware resources (such as the processor or main memory) are implicitly shared among the VMs. Although it should be impossible to infer sensitive information from these resources, a wide range of side-channel attacks and covert channels proves otherwise [18, 41, 57, 60]. In a side-channel attack, the attacker tries to infer secret data (e.g., keys) from another VM without explicit communication by merely monitoring execution artifacts rather than exploiting software bugs. A covert channel consists of a sending and receiving instance which intentionally exfiltrate information over a channel not meant for communication.

In previous work, especially processor caches have been exploited as communication channels. However, since state-of-the-art cache isolation techniques such as Intel's Cache Allocation Technology (CAT) are able to prevent cache-based attacks [35], security researchers shifted their focus to other implicitly shared resources. Gras et al. [18] were the first to mount a side-channel attack via the Translation Lookaside Buffer (TLB), thus, bypassing state-of-the-art side-channel protections. Beside their main objective of leaking secret keys, Gras et al. [18] also showed that covert channels over the TLB are practical by implementing a proof-of-concept covert channel between processes. In this work, we aim to considerably increase the performance of TLB-based covert channels in terms of channel bit rate and reliability, thereby demonstrating that holistic techniques for microarchitectural resource partitioning are needed.

In order to build a reliable covert channel over the TLB, the issues of synchronization and noise due to concurrently running processes have to be handled by an appropriate communication protocol. Similar to Maurice et al. [41], we lend ideas from wireless protocols and construct a multi-layer protocol consisting of a physical and data-link layer. These layers include error detection and

correction mechanisms such as the cyclic redundancy check (CRC) or the Reed-Solomon code. We evaluate the robustness of our covert channel by transmitting real-world data such as documents and multimedia contents across VMs while introducing different levels of interference. Our design is able to achieve bit rates of up to 200 kB/s at an error rate of 0 % on an idle system. In addition, we present a novel technique to accurately monitor TLB entries by leveraging the accessed bit in page table entries.

In Chapter 2, we provide background information required to understand TLB-based covert channels and discuss previously published side-channel attacks and covert channels. Chapter 3 introduces the attack scenario we assume throughout our work and presents techniques to monitor TLB entries. Furthermore, we analyze the challenges we face when building TLB-based covert channels. The proposed design of our channel is depicted in Chapter 4, followed by a detailed evaluation in Chapter 5. The thesis concludes with Chapter 6 where we summarize the most important points and motivate future work.

Chapter 2

Background

In this chapter, we provide background information required to understand TLB-based covert channels. We recall the concept of virtual memory and look at its modern implementation with focus on the TLB. Next, we motivate side-channel attacks and covert channels, along with an examination of real-world examples. Eventually, we give an overview over error detection and correction mechanisms which we will use in this thesis.

2.1 Virtual Memory

With the rise of multiprogramming, virtual memory has become a core feature of every modern operating system [56, p. 181ff]. Each process is assigned its own address space, so that it cannot access data from another process or crash the machine by performing invalid memory accesses. To enforce address space isolation, processes can only operate on memory via virtual addresses. On access, the memory management unit (MMU) in the CPU transparently translates the virtual address into a physical address. An invalid access raises a page fault which has to be handled by the operating system. To implement address spaces, operating systems use techniques such as base and limit registers, segmentation, or paging.

The general flow of an address translation on a modern page-based architecture (e.g., x86) is depicted in Figure 2.1. The address space is divided into equally-sized pages, typically of 4 KiB granularity. All pages of a process are organized in a page table which is managed by the operating system. Within the page table, each virtual memory page is represented by a page table entry (PTE). Typical information in a PTE include [11]:

- *Physical Frame Number*: Physical base address of the page.
- *Present Bit*: Indicates whether the page is currently mapped in physical memory.
- *Protection Bits*: Specify access permissions such as read, write, and execute as well as whether the page is accessible from kernel-mode only and others.
- *Accessed and Dirty Bits*: Whenever the MMU uses a PTE as part of an address translation, it sets the accessed bit in that entry (if it is not already set). Whenever there is a write to an address, the MMU sets the dirty bit (if it is not already set) in the PTE belonging to the address.[1]

Since the address space is usually sparsely populated, allocating all PTEs required to cover the whole address space results in a waste of memory. Therefore, many architectures employ a multi-level page table, where PTEs of each level

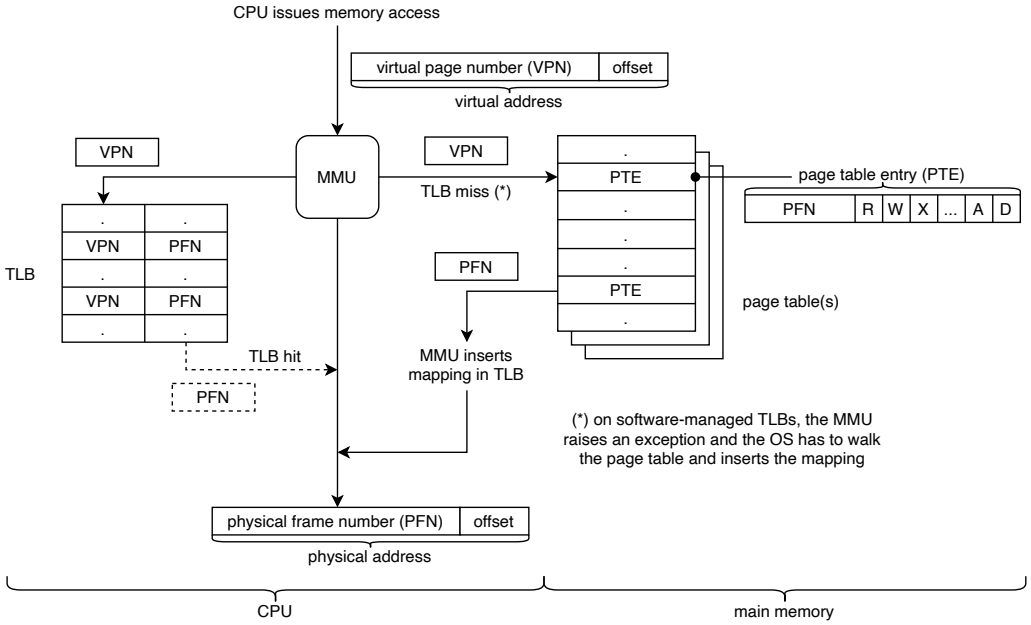


Figure 2.1: Memory address translation on a page-based architecture. The Translation Lookaside Buffer (TLB) plays a fundamental role in speeding up the process by caching recent address translations, thereby eliminating page walks.

[1]Note that the processor may cache these bits internally. This implies that, if software resets the accessed or dirty bit ($1 \rightarrow 0$), the corresponding bit might not be set again on a subsequent access to the same address unless the cache has been written back [11].

point to the physical page of a nested-level table. PTEs of the last level point to the actual physical page. If a specific portion of the memory is not used, the PTE of a higher level can be invalidated and lower-level tables for that entry do not need to be allocated. x86 specifies a two-level hierarchy for 32-bit and four or five levels for 64-bit architectures (regarding 4 KiB pages). While a multi-level page table efficiently diminishes memory consumption, it comes at the cost of an additional memory access per level during the virtual address resolution. This brings up the idea to cache translations from virtual to physical memory in order to skip the page table walk and its associated memory accesses. This cache is referred to as Translation Lookaside Buffer (TLB).

2.1.1 Translation Lookaside Buffer (TLB)

Caches in general evolved from the tradeoff that low-latency storage is technically challenging, constraining its capacity due to cost and physical space, whereas less expensive, larger storage does not meet the bandwidth needed for efficient processing [55, p. 120ff]. A cache can be thought of as a key-value store with a fixed number of entries, where the key or tag identifies a cache line containing the associated data. We speak of a *cache hit* in the case that the requested data is found in the cache memory, or of a *cache miss*, if the data is not cached and has to be fetched from the backing storage. There are three different methods to map tags to cache lines [55, p. 133ff]:

- *fully associative*: A tag can map to any cache line. This makes the cache most flexible, however, since all entries have to be searched for the tag on a data request, the hardware implementation is quite expensive.

- *direct mapped*: A tag can map to one cache line only. Therefore, just one entry has to be searched for the tag on a data request. The problem is that an unfortunately chosen set of tags could map to the same cache line, always evicting each other while the remaining cache lines stay unused.

- n*-way set associative*: The cache is organized in sets, each spanning n cache lines (i.e., n ways). A tag can map to any cache line within a specific set. On a data request, only the entries within this set have to be searched. The number of sets is chosen to balance hit rate and implementation cost.

Once all cache lines are filled with data, a replacement policy is applied to determine which entry will be evicted on further insertions. Examples for replacement policies encompass random eviction, first-in-first-out, or least-recently-used replacement. Which cache organization and replacement policy is used for a cache has to be considered on a case-by-case basis.

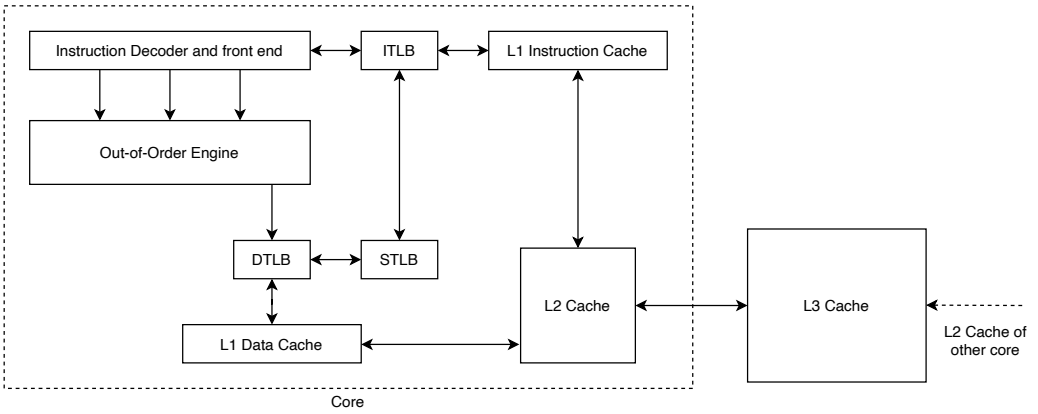


Figure 2.2: Cache hierarchy of modern Intel CPUs [11]. The L1 caches are split for instructions and data. The L3 cache is shared among all cores. In hyperthreaded designs, the caches and TLBs are either competitively shared or statically partitioned.

Fast caches are expensive to implement in hardware and are therefore inherently small. Larger caches have better hit rates at the cost of higher latency. That is why chip designers often choose to embed a multi-level cache hierarchy. In Intel's current designs, three cache levels (L1-L3) are embedded in the CPU to cache data or instructions from the main memory [11]. Figure 2.2 summarizes the cache hierarchy of modern Intel CPUs. The first level (L1) closest to the execution unit is of limited size (few kilobytes) and split into two dedicated caches for data and instructions. The second level (L2) holds both data and instructions, but is still exclusive per core. The third level (L3) is the largest cache level (multiple megabytes) and usually shared between all cores.

The virtual address resolution as described in Section 2.1 issues multiple memory accesses. Although cache hits in the L1-L3 caches reduce the latency of a single access, the accumulated latency of multiple accesses still negatively impacts performance. For this reason, the MMU caches recent address translations from virtual to physical memory in the TLB to skip the page table walk and its associated memory accesses altogether[2]. On each memory access, the MMU performs a lookup in the TLB.

Depending on the architecture, the TLB is managed either in software (e.g., MIPS) or hardware (e.g., x86). When implemented in software, TLB misses raise an exception that has to be handled by the operating system. The interrupt handler walks the page table, returns the translated physical address and inserts it as a new TLB entry. On hardware-managed TLBs, as it is the case on x86,

[2]In fact, Intel provides additional caches for paging structures to speed up the page walk [7].

		ITLB			DTLB			STLB		
Architecture	Year	s	w	m	s	w	m	s	w	m
Haswell	2013	32	4	lin	16	4	lin	128	8	lin
Broadwell	2014	32	4	lin	16	4	lin	256	6	XOR-8
Skylake	2015	16	8	lin	16	4	lin	128	12	XOR-7
Coffee Lake	2017	16	8	lin	16	4	lin	128	12	XOR-7
Ice Lake	2019	16	8	-	16	4	-	128	16	-

Table 2.1: TLB organizations per microarchitecture [10]. All levels use set associative mapping. s = number of sets, w = number of ways, m = mapping according to [18]. The total number of entries equals *s* times *w*.

the MMU automatically walks the page table on a TLB miss and inserts the TLB entry. On a TLB hit (i.e., the address translation already exists as a TLB entry), the MMU directly returns the physical address and does not access the page table.

Following the concept of multi-level memory caches (L1-L3), the TLB can be structured in multiple levels as well. For instance, recent Intel microarchitectures employ a two-level TLB hierarchy [10]. The first-level TLB (sometimes referred to as L1 TLB, not to be confused with the L1 cache) is split into a dedicated TLB for address translations belonging to instructions (ITLB) and data address translations (DTLB). The second-level TLB is shared for both address types (STLB, sometimes L2 TLB). In Table 2.1, we list actual TLB organizations for the currently used architectures concerning 4 KiB page translations[3].

The actual mapping scheme, that is, how virtual addresses are mapped to sets of the TLB, is implementation-specific and has not been made publicly available by Intel. Nonetheless, Gras et al. [18] reverse-engineered some TLB properties on recent Intel microarchitectures. While producing various address access patterns, they gather fine-grained information about TLB misses provided by the processor's performance counters (PMCs). The authors identify two mapping schemes:

- *linearly-mapped*: The target set is calculated as *page* modulo *s*, with *page* being the virtual address without the page offset (the 12 least significant bits for 4 KiB pages) and *s* being the number of sets.
- *complex-mapped*: The TLB set number can be expressed as an XOR of a subset of bits of the virtual address. For instance, an *XOR-8* mapping

[3] TLBs may contain separate regions for different page sizes, each with their own cache organization [10]. Though we focus on 4 KiB pages in this thesis, TLB-based covert channels are possible with other page sizes in the same way.

XORs 8 consecutive virtual address bits. Assuming 4 KiB pages (12 bit page offset), bits 12 to 19 are XORed with bits 20 to 27, with 0 being the least significant bit of the virtual address.

Table 2.1 includes the mapping schemes of all TLBs used in recent Intel microarchitectures according to the reverse-engineering results of Gras et al. [18]. Furthermore, they confirm Intel's statement that since the Nehalem microarchitecture, ITLB (4 KiB) page entries are statically allocated between two logical processors (i.e., between hyperthreads), whereas the DTLB and STLB are a competitively-shared resource [10]. Still, the underlying replacement policy remains unknown.

On a context switch, the virtual address space changes and cached physical address translations in the TLB are no longer valid. The simple solution of flushing the TLB on every context switch comes with a high performance penalty when switching back and forth between processes. For that reason, TLBs support additional identifiers in their entries to differentiate between different contexts. As a result, cached address translations belonging to a different address space can remain in the TLB and will be ignored during lookups (they are nevertheless subject to replacement). This technique, also known as *tagging*, has become common practice in operating systems [39] and hypervisors [61]. Intel provides tagging on process level (Process-Context Identifier, PCID) as well as on VM level (Virtual Processor Identifier, VPID) [11].

2.2 Side-Channel Attacks & Covert Channels

Information security builds upon the principles of confidentiality, integrity, and availability. In modern cloud systems, lots of critical user, business, or even governmental data [42] are stored in databases. As virtualization and containerization become more prevalent in cloud environments [6], isolation has gained increasing importance to protect sensitive data from unauthorized access. An adversary has to overcome this isolation in order to steal data. One common way to do this is by exploiting software vulnerabilities, which in turn can be patched with an update. Therefore, attackers came up with more sophisticated approaches, among which are side-channel attacks. The goal of side-channels is to infer sensitive data from execution artifacts rather than exploiting software bugs. In virtualized environments, we assume that whenever multiple VMs or containers run on a common physical host, some resources (e.g., CPU caches) are implicitly shared and can be leveraged to infer data.

Algorithm 2.1: Binary Square-and-Multiply exponentiation

input: base b, exponent $e = (e_{n-1} \dots e_0)_2$
output: b^e

1: $r \leftarrow 1$
2: **for** $i \ from \ n-1 \ to \ 0$ **do**
3: $\quad r \leftarrow r^2$
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad r \leftarrow r * b$
6: **return** r

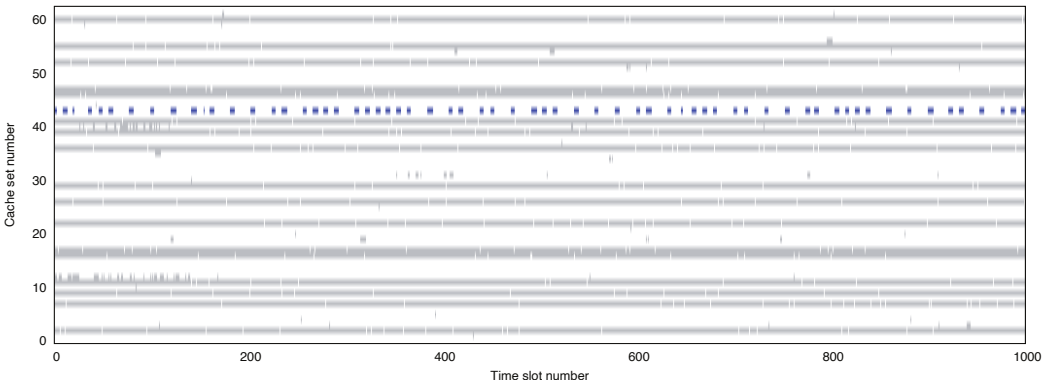


Figure 2.3: Cache activity during Square-and-Multiply exponentiation [34]. The time between the highlighted cache accesses depends on the exponent's bits. The resulting bit pattern is 00100111...

Although side-channels can also be built on the software level (e.g., by snooping network packet sizes to get hints about their contents [54]), research mainly focuses on the hardware level. A common attack is to infer secret keys by monitoring hardware effects such as power consumption [29], electromagnetic radiation [1], or memory access latency (related to caches) [34]. The processor's caches are of particular interest because of their fine-grained resolution of leaked information and low latency, thereby allowing channels with high bit rates. Liu et al. [34] depict an attack on the Square-and-Multiply exponentiation (see Algorithm 2.1) which is used in variations by both RSA and ElGamal encryptions[4]. Leaking the exponent e may lead to recovery of the private key. The core idea of the attack is to monitor whether the multiplication in line 5 occurs or not. Then, the attack deduces the exponent's bits from the condition in line 4. Tracing the cache activity leads to a pattern as depicted in Figure 2.3. While processing a

[4]In fact, modular exponentiation is used. Though being a simplified example, the attack idea remains the same.

binary 1, squaring is followed by a multiply. In contrast, for a binary 0, squaring is the only operation in the loop. Therefore, by observing the time between subsequent squarings (i.e., cache accesses caused by them), we can recover the exponent. Another example is AES, where lookup tables are pre-calculated and stored in main memory in order to improve performance [3]. During encryption, the CPU loads these tables into the cache and traces of cache accesses disclose indices into the tables to the attacker.

From Section 2.1.1, we recall that the L1 and L2 caches are shared between a single core and the L3 cache is shared between all cores. Therefore, attacks leveraging the L1 or L2 cache can only be mounted with co-resident VMs on one core, while the prerequisite for L3-attacks lessens to co-residency on the same CPU. For instance, Inci et al. [24] present a side-channel attack to recover a 2048-bit RSA key from co-resident Amazon EC2 instances using the L3 cache. On process level, Gullasch et al. [21] demonstrate the recovery of a full AES-128 encryption key in "almost realtime" via the L1 cache.

The primary building block of all cache-based side-channel attacks is to monitor cache lines. To do so, a commonly adapted technique is Prime+Probe [14]. It can be segmented into three steps as depicted in Figure 2.4: During the *priming* phase, the attacker manipulates the (shared) cache to be in a known state, for example by forcing the eviction of certain address ranges. To evict entries, the process allocates a block of memory and accesses data within it. After preparing the cache, the attacker waits for a specific time period while the victim operates on the cache. Shortly after, in the *probing* phase, the attacker examines how the victim's activity has changed the state of the cache. For this, the attacker measures the time taken for accessing the same addresses as in the *priming* phase. A threshold determines whether the access resulted in a cache hit or miss, and, depending on the timer's resolution, it is even possible to differentiate between the cache levels. Of course, this presupposes the existence of a high-precision timer, e.g., the timestamp counter of x86 processors[5]. However, Schwarz et al. [51] have shown that even in absence of fine-grained timers, other sources can be used to measure time with sufficient precision. A drawback of the Prime+Probe attack is that the attacker has to know the mapping scheme from virtual to physical addresses in order to target specific cache sets, because cache sets are usually derived from the physical address. Address space layout randomization (ASLR) additionally lowers the chance for a successful attack. Therefore, Yarom and Falkner [62] propose an alternative monitoring approach for caches, Flush+Reload, which relies on sharing pages between the attacker and the victim process rather than knowledge of the actual cache sets. Sharing identical memory

[5]The timestamp counter is a register which counts the number of clock cycles since reset [11]. Its value can be obtained by using the unprivileged `RDTSC` instruction.

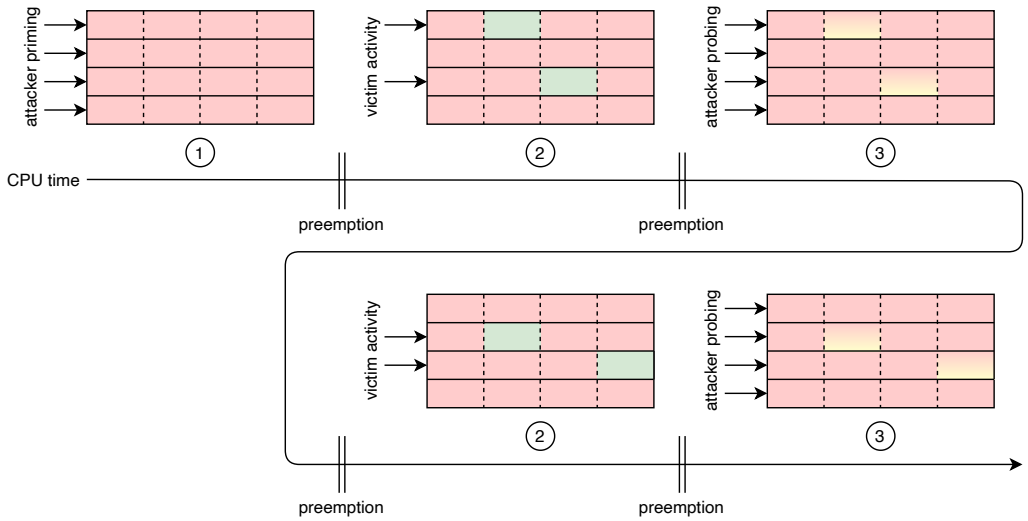


Figure 2.4: The three phases of the Prime+Probe technique. By accessing the cache lines during the probing phase, they will be reset to primed state, making the priming phase unnecessary in subsequent iterations.

pages between processes or virtual machines is common practice to reduce the memory footprint of a system (memory deduplication). During the *flush* phase, the attacker flushes (shared) memory lines from the cache. After a wait phase, the attacker *reloads* these memory lines, measuring the time to load them. If the victim accessed a memory line during the wait phase, the line will be available in the cache and reloading it will take less time than reloading a memory line that has not been accessed. As a further advancement of Flush+Reload, Gruss et al. [19] implement a monitoring mechanism without the need of any attacker-issued memory accesses.

In contrast to side-channel attacks, covert channels consist of a sending and receiving instance which intentionally exfiltrate information over a channel not meant for communication. Covert channels are closely related to side-channel attacks, since a successful side-channel attack can always be converted to a covert channel by causing the hardware state changes intentionally via a sender program. For instance, we can feed the Square-and-Multiply example from Algorithm 2.1 with arbitrary bit strings by interpreting those as exponents. On the receiver side, we observe the same pattern as during a side-channel attack, but interpret it as a bit string rather than a numeric exponent. This however assumes that the adversary is able to execute attacker-controlled code on the victim's machine.

The United States Department of Defense (DoD) defines a covert channel as "any communication channel that can be exploited by a process to transfer

information in a manner that violates the system's security policy" [32]. The concept of a covert channel was first introduced by Lampson [30] in 1973. Since then, lots of different attack vectors have been used in order to bypass software and hardware barriers. The severity of a covert channel is determined by its bit rate. A channel bit rate exceeding 100 bit/s is considered "high" by the DoD, whereas a bit rate of less than 1 bit/s is considered acceptable in most application environments [32]. Nonetheless, the security risk has to be evaluated on a case-by-case basis, since sometimes even small amounts of data contain sufficient sensitive information (e.g., passwords).

Publication	Via	Setup	Bit Rate	Error Rate
Schmidt [50]	Frequency	Native	0.135 B/s	0 %
Kalmbach [26]	Frequency	Virtual	1.125 B/s	0 %
Okamura and Oyama [43]	CPU load	Virtual	0.061 B/s	0 %
Percival [45]	Cache	Native	400 kB/s	–
Gruss et al. [19]	Cache	Native	496 kB/s	0.84 %
Ristenpart et al. [49]	Cache	Cloud	0.025 B/s	–
Maurice et al. [41]	Cache	Cloud	45.09 kB/s	0 %
Pessl et al. [46]	DRAM	Virtual	74.5 kB/s	0.4 %
Gras et al. [18]	TLB	Native	0.875 B/s	0.002 %

Table 2.2: Comparison of state-of-the-art covert channels. The error rate represents the percentage of bit errors in the final received data, that is, after error correction methods have been applied (if applicable). Native = Between Processes, Virtual = Between VMs, Cloud = Between cloud VM instances.

In the remainder of this section, we focus on hardware-based channels. The bit and error rates of these channels largely depend on the chosen resource. In Table 2.2, we offer a comparison of several publications. Kalmbach [26] builds a channel via CPU frequency scaling provided by Intel's Turbo Boost technology. While the current CPU frequency can be easily retrieved from a public interface, it changes sluggishly with the load, resulting in low bit rates. Similar work has been done on AMD by Schmidt [50] with even worse bit rates due to "high, asymmetric latencies". To mitigate these channels, it is sufficient to set the CPU frequency to a fixed value which incurs a performance loss (unless the CPU is being overclocked). Still, a covert channel using just the CPU load is possible as demonstrated by Okamura and Oyama [43]. From side-channel attacks, we know that caches provide high resolution and low latency, thereby enabling channels with higher bit rates. The same techniques we described earlier (e.g., Prime+Probe) can be applied for covert channels. Percival [45] did a first study of a covert channel between two processes on the same core leveraging the L1

cache. They estimate a bit rate of up to 400 kB/s using "an appropriate error correcting code". Moving to cross-core attacks, Gruss et al. [19] achieve a bit rate of 496 kB/s with an error rate less than one percent with their Flush+Flush monitoring mechanism applied to the L3 cache in a native setting. Most challenging are attacks between different CPUs on a multi-processor platform. Irazoqui et al. [25] mounted a cross-CPU side-channel exploiting cache coherency protocols of AMD, Intel, and ARM processors. They did not extend their approach to a covert channel, but successfully leaked a full AES and ElGamal key. Another promising resource for cross-CPU channels is the main memory. Pessl et al. [46] came up with a main memory covert channel capable of up to 74.5 kB/s across VMs on different CPUs with an error proability of just 0.4 %. They exploit timing differences caused by row conflicts in the (shared) DRAM row buffer. In contrast to Prime+Probe and equivalent cache monitoring techniques, their approach does not rely on shared memory. Within virtualized environments, bit rates are usually lower due to interference of the hypervisor and other VM instances. Ristenpart et al. [49] were the first to mount a covert channel between two Amazon EC2 instances via the L1 and L2 cache. However, by using the L1 and L2 cache, it is necessary to achieve co-residency not only on the same physical host, but on the same core as well. Maurice et al. [41] reduced this challenge to co-residency on the same host by using the L3 cache. They construct a covert channel capable of up to 45.09 kB/s across Amazon EC2 instances while eliminating all errors through a multi-layer protocol based on retransmissions and forward error correction. Gras et al. [18] were the first to mount a TLB-based side-channel attack using the Prime+Probe technique known from cache attacks. Beside their main objective of leaking secret keys, they also implement a proof-of-concept covert channel between processes with a bit rate of around 0.875 kB/s in idle state and 0.5 kB/s under heavy interference.

2.3 Error Detection & Correction

Communication over real channels comes with noise which causes errors in transmitted data [23, p. 323ff]. By definition, covert channels are not even meant for communication and consequently need to cope with interference introduced by concurrently running processes. The transmission can be made more robust against errors by appending redundant information (e.g., parity bits) to the data. For instance, error detection codes are used to detect whether a data transmission was successful and to request a retransmission from the sender if not [26, 41]. Corrupt or incomplete data may be recovered on the receiver side using (forward) error correction codes. This is especially important for unidirectional channels which lack a feedback channel and cannot request retransmissions. Computing

and adding redundant parity data causes an overhead in terms of the amount of data to transmit and total transmission time, thereby lowering the effective bit rate. An optimal error detection (or correction) mechanism is capable of detecting (or correcting) as many corrupt bits as possible at minimal overhead. Furthermore, a scalable error detection (or correction) code allows to control the overhead depending on the expected error probability. In the following sections, we recapitulate error detection and correction codes relevant for this thesis.

2.3.1 Berger Code

Berger [4] proposed a code which is able to detect any number of unidirectional errors, that is, errors that only flip ones into zeros or zeros into ones. Channels subject to this error behavior are referred to as *asymmetric channels*. The Berger code is often found in hardware circuits due to its simple implementation, e.g., to facilitate fault-tolerant computer systems [36].

As depicted in Figure 2.5, the code word is formed by the information word concatenated with the number of binary zeros in the information word. Therefore, if any number of unidirectional bit flips occurs, the number of zeros will differ from the parity word. To a limited extend, the Berger code also detects bit flips in both directions, as long as the number of zero-to-one-flips does not match the number of one-to-zero-flips.

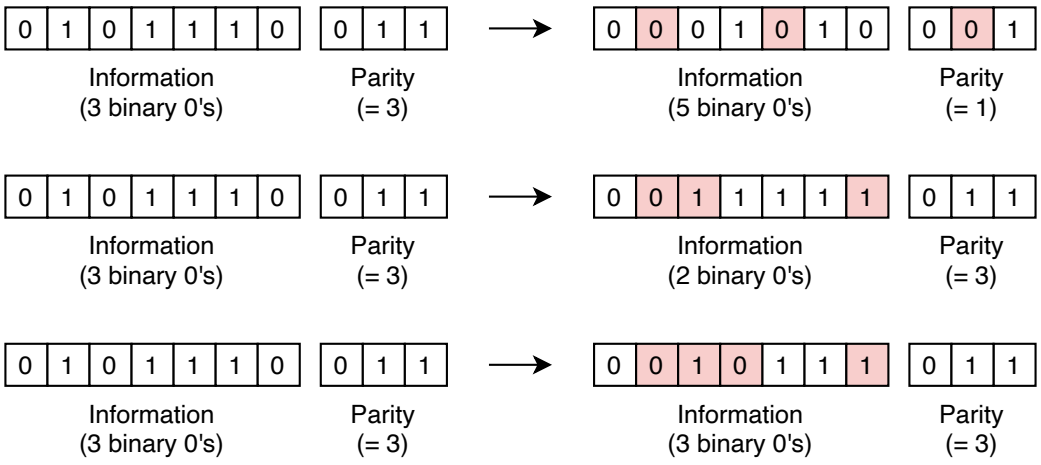


Figure 2.5: The Berger code detects errors by counting the number of zeros. In the first example, only unidirectional bit flips ($1 \rightarrow 0$) occur and the error is detected. The second and third example demonstrate that in some occasions, bit flips in both directions can be detected, while in other cases, bit flips cancel out each other.

Though the length of the Berger code scales with the information word length and cannot be specified manually, the code is of interest because it can be calculated efficiently. On Intel platforms with SSE4.2 extensions, the `POPCNT` instruction returns the number of bits set to 1. Negating the operand prior to the execution of `POPCNT` yields the Berger code of the operand with a throughput of one instruction per clock cycle [12].

2.3.2 Cyclic Redundancy Check

The cyclic redundancy check (CRC) is an error detection mechanism based on the mathematical concept of cyclic codes [47]. CRCs consist of a fixed number of bits computed as the remainder of a polynomial division of the data. The divisor is referred to as *generator polynomial* and has to be chosen in a way to maximize the code's error detection capabilities. Depending on a channel's properties, some generator polynomials perform better than others of the same degree. On retrieval, the CRC is recalculated from the data and if it matches the transmitted CRC, the data is assumed to be correct. CRCs are particularly suited for detecting short burst errors which are characteristic for communication infrastructure [15].

Name	Applications	Generator Polynomial[6]
CRC-1	$=$ *parity bit*, most hardware	0x1 $(x+1)$
CRC-5-USB	USB	0x05 (x^5+x^2+1)
CRC-8-CCITT	ISDN	0x07 (x^8+x^2+x+1)
CRC-16-CCITT	Bluetooth, SD cards	0x1021 $(x^{16}+x^{12}+x^5+1)$
CRC-32	Ethernet, SATA	0x04C11DB7
CRC-32C	iSCSI, ext4, SSE4.2	0x1EDC6F41

Table 2.3: Common, standardized CRCs and their applications. The name's numeric postfix corresponds to the number of parity bits.

An excerpt of common, standardized CRCs is listed in Table 2.3. The higher the number of parity bits, the more reliable the error detection. Efficient CRC computation can be achieved by a series of shift and XOR operations. On Intel platforms with SSE4.2 extensions, the `CRC32` instruction exposes hardware-based CRC-32C computation. For short CRCs, precalculated lookup tables present a fast alternative.

[6] In normal representation: The most significant bit (x^n) is left out in the hexadecimal respresentation (as it is always 1). The hexadecimal polynomial contains only the coefficients $x^{n-1} \dots x^0$. Example: 0x1021 = 0001 0000 0010 0001 $= x^{16}+x^{12}+x^5+x^0$.

2.3.3 Hamming Code

The Hamming code is a simple error detection code, capable of detecting and correcting one-bit errors [23, p. 353ff]. The idea of a single parity bit is extended, so that multiple parity bits are calculated over specific subsets (expressed as a *generator matrix*) of the data bits. After transmission, multiplication of the received code word with a *parity-check matrix* yields the bit error position (known as *syndrome*). If the position equals zero, no error has been detected. The notation (n, k) refers to code words of n bits total length and k data bits, implying $n-k$ parity bits. The number of parity bits depends on the data word length. By appending an additional parity bit, the Hamming code can be extended to detect up to two-bit errors. The extended Hamming code differentiates the following cases:

- syndrome $= 0$, additional parity $= 0 \rightarrow$ **no error**
- syndrome $\neq 0$, additional parity $= 1 \rightarrow$ **correctable (one-bit) error**
- syndrome $= 0$, additional parity $= 1 \rightarrow$ **error in additional parity**
- syndrome $\neq 0$, additional parity $= 0 \rightarrow$ **non-correctable (two-bit) error**

Still, only one-bit errors can be corrected. Due to its simple implementation, the Hamming code is widely used for hardware with low error rates (e.g., ECC memory).

2.3.4 Reed-Solomon Code

In 1960, Reed and Solomon [48] proposed an error correction code based on polynomial interpolation. The key idea is that a polynomial of degree $p-1$ is fixed by p (distinct) data points. By appending an arbitrary amount of points on the polynomial as redundant information, the receiver is able to reconstruct the original polynomial even if some data points get lost or corrupted. This makes the code easily scalable to match the required level of robustness. An example is given in Figure 2.6. Due to its versatility, the Reed-Solomon code is widely used in consumer applications such as CDs or digital television, as well as for NASA's and ESA's space exploration missions (e.g., the Voyager expeditions) [59].

The data points are represented by *symbols*. The Reed-Solomon code encodes blocks of r-bit symbols whereby the block length n is given as $2^r - 1$ symbols [23, p. 398ff]. The notation (n, k) refers to a block of n symbols which contains k data symbols, leaving $n - k =: t$ symbols as redundant information.

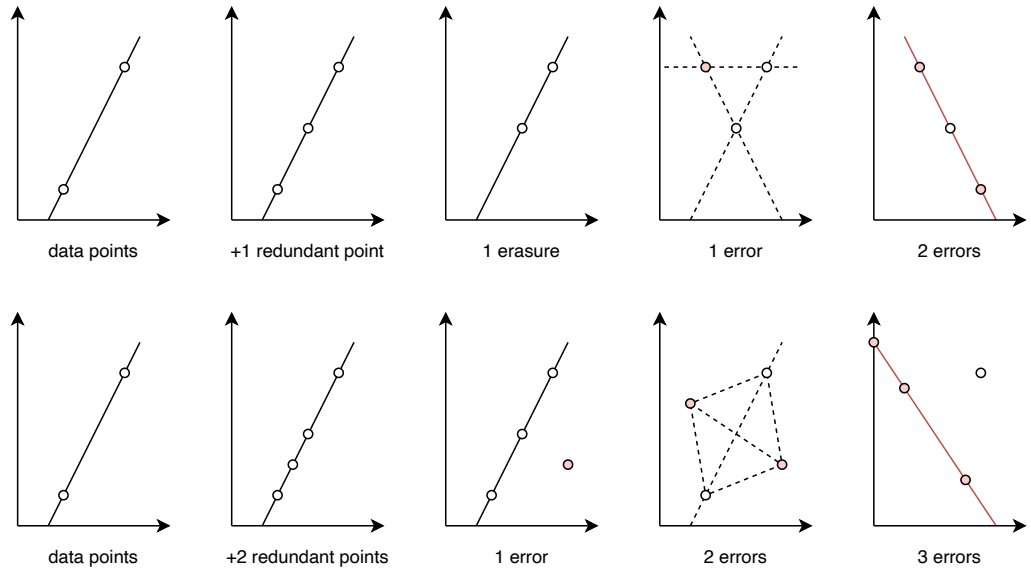


Figure 2.6: Reed-Solomon polynomial interpolation example. The data polynomial (here of degree 1) is fixed by 2 points. By appending 1 redundant point on the polynomial, we are able to correct 1 erasure or to detect 1 error. For 1 error, we can derive 3 possible polynomials, preventing us from correcting the error. 2 errors might remain undetected. By appending 2 redundant points, we can correct 1 error with a majority decision. 2 errors lead to an ambiguity and therefore can be detected but not corrected. 3 errors might fool the majority decision, leading to incorrect results.

In Figure 2.6, we illustrate that t additional symbols allow to:

- compensate for t erasures or
- detect t errors or
- correct $\lfloor \frac{t}{2} \rfloor$ errors.

The error correction capabilities can be further increased by providing information about where erroneous symbols are located. By knowing their offsets within an encoding block, we can treat them as erasures (i.e., discard them) and are able to correct up to t errors per block.

Chapter 3

Analysis

In Section 2.2, we looked at several side channels and covert channels that infer or transfer data across isolation boundaries (e.g., VMs) without having means for explicit communication. One of the most effective methods is to exploit timing artifacts in the processor's cache hierarchy. Caches are of particular interest because of their fine-grained resolution of leaked information and low latency, thereby allowing channels with high bit rates. Maurice et al. [41] demonstrated how an error-free covert channel with a decent bit rate of around 45 kB/s is possible by leveraging the L3 cache in combination with appropriate error correction mechanisms.

To mitigate cache-based channels, it is necessary to enforce isolation even on the microarchitectural level. An obvious solution for cloud providers is to prohibit co-residency of VM instances belonging to multiple tenants (or restrict host sharing to "trusted" tenants). In the hypervisor, a radical way to enforce isolation is by flushing the cache on every guest switch. However, this greatly lowers the effectiveness of caches and degrades overall performance. Therefore, hardware vendors implemented mechanisms to close down cache-based channels right at the microarchitecural level. For instance, modern Intel CPUs implement Cache Allocation Technology (CAT)[1]. Designed to guarantee quality-of-service (QoS) regarding the L3 cache performance [11], CAT can also be used to mitigate cache-based side and covert channels. CAT partitions the cache by ways, thereby protecting entries of a set from eviction due to applications assigned to another partition. However, CAT only offers a limited set of four distinct partitions without the possibility to dynamically balance the cache allocation according to the needs of applications. This makes CAT unsuited for realistic cloud environments. Liu et al. [35] overcome these shortcomings by implementing a finer-grained L3 page-level partitioning of two ways (*secure* and *non-secure* partition) on top of

[1] Launched as part of the Intel Xeon E5-2600 v3 product family (2014) [22].

CAT which dynamically maps sensitive code and data pages to the secure partition instead of assigning whole VMs to each of the four (statically-sized) CAT partitions. Gruss et al. [20] propose another approach to eliminate cache attacks by enforcing data cache quiescence during sensitive operations. Cache monitoring mechanisms, e.g., Prime+Probe, are based on the eviction of cache entries. Using Intel's Transactional Synchronization Extensions (TSX), sensitive operations can be encapsulated in a TSX transaction which gets aborted as soon as concurrent data cache activity occurs.

As techniques such as CAT and TSX are able to prevent cache-level attacks, security researchers are shifting their focus to other implicitly shared resources. Gras et al. [18] were the first to mount a TLB-based side-channel attack, thus bypassing state-of-the-art side-channel protections. In contrast to other implicitly shared resources (e.g., CPU frequency [26]), the Prime+Probe technique applies to the TLB as well and allows leaking of fine-grained information similar to cache attacks. Beside their main objective of leaking secret keys, Gras et al. [18] also implement a proof-of-concept covert channel between processes with a bit rate of around 0.875 kB/s in idle state and 0.5 kB/s under heavy interference. Though they showed that covert channels over the TLB are practical, the bit rate is not yet able to keep up with cache-based channels.

In this work, we aim to considerably increase the performance of TLB-based covert channels in terms of channel bit rate and reliability, thereby demonstrating that holistic techniques for microarchitectural resource partitioning are needed. The following section leads up to the attack scenario that we assume throughout our work. Afterwards, we discuss methods to monitor TLB entries and examine the challenges of synchronization and noise.

3.1 Attack Scenario

Public clouds (such as Amazon AWS, Microsoft Azure, etc.) represent a lucrative target for cyberattacks due to their large-scale use for processing sensitive data [42, 44]. To increase utilization at lower costs, hosting companies consolidate multiple tenants per physical machine by virtualization and containerization [6]. This opens up the possibility to build a covert channel between VMs on the same physical host (*co-resident* VMs). In a probable scenario, an adversary uses an exploit to execute attacker-controlled code on the victim's VM (e.g., within a web service). The attacker wants to exfiltrate sensitive data from the compromised VM and send it to a VM under his control. Direct communication between the VMs, for example over network, is prevented by the cloud provider. Similar to L1 or L2 cache-based channels [45, 49], the TLB's core-exclusivity requires both VMs to reside on the same core.

For an attacker operating in public clouds, this means that he has to ensure co-residency between the VMs on the same physical host. The co-residency problem has been previously examined by Varadarajan et al. [58] for Amazon EC2, Google Compute Engine, and Microsoft Azure. They analyze how various parameters affect the placement of VM instances and develop *launch strategies* to exploit these *placement policies* in order to achieve host co-residency. Among these parameters are:

- cloud provider, data center, region
- time launched, time of the day, days of the week
- instance type, number of VMs in use

Their proposed launch strategies obtain co-residency faster (10x higher success rate) and cheaper (up to $114 less) when compared to a secure reference placement policy. They show that not only when targeting smaller data centers, there is a high chance of co-residency, but also on massive data centers with numerous physical servers, achieving co-residency is practical.

In Section 2.1.1, we introduced the concept of TLB *tagging* to preserve TLB entries during context switches. Without tagging, the TLB has to be flushed on every context or guest switch. This makes the TLB an isolated resource, thereby disabling channels based on the competitive eviction of entries between processes or VMs. However, similar to flushing caches on every context switch, disabling tagging on process or VM level is undesirable due to performance loss. In particular, tagging on process level plays an important role in the proposed mitigation of the Meltdown vulnerability [33]: Where previously a user process had a single page table with entries for both user- and kernel-space mappings, it now has two page tables to strictly separate kernel from user memory. On kernel boundary crossings (e.g., system calls), the operating system has to switch between these two tables. With tagging, the TLB does not need to be flushed for every context switch, keeping the negative performance impact low. Therefore, having tagging on process level enabled is in the interest of the cloud provider and its tenants for performance reasons. This already allows TLB-based covert channels between processes (e.g., in a containerized environment).

To mount a covert channel between VMs, we also require tagging on VM level to preserve entries between guest switches. On Intel platforms, the Virtual Processor Identifier (VPID) can be used to tag TLB entries belonging to a VM. According to Intel, the VPID benefit is very dependent on the workload and memory virtualization mechanism [16]. In process and memory-intensive workloads, they promise more than 2 % performance gain, while worst-case synthetic benchmarks yield up to 15 % better results with VPIDs enabled. We confirm these

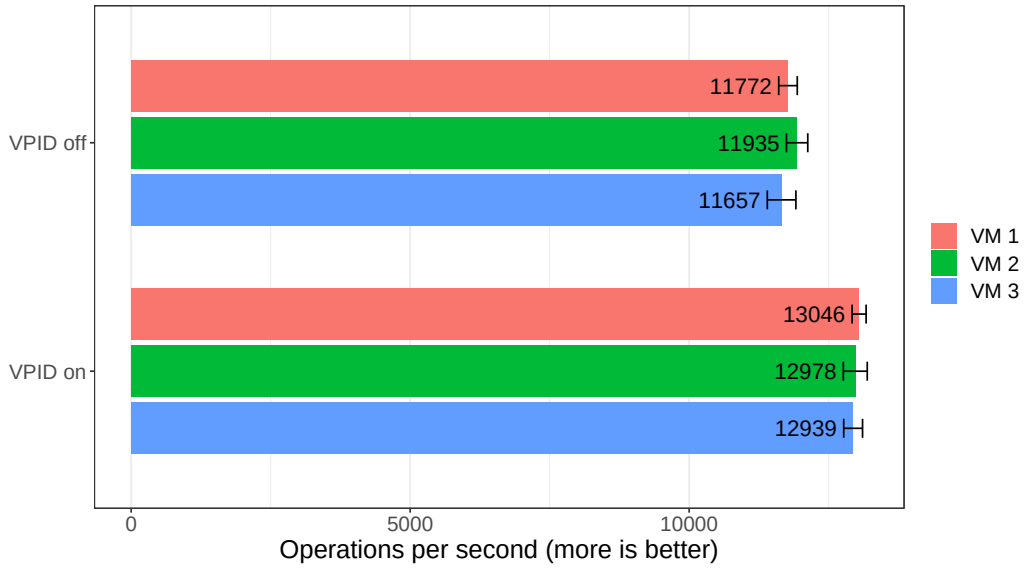


Figure 3.1: Memcached benchmark with VPID tagging being enabled and disabled. Enabling the VPID feature achieves a performance gain of about 10 %, equally for each VM.

results by running a memcached-based benchmark[2] within three single-core VMs on a system with VPID tagging being enabled and disabled. Memcached is a distributed memory object caching system, thereby representing a memory-intensive workload typical for servers. From the results in Figure 3.1, we see that enabling the VPID feature achieves a performance gain of about 10 % in this scenario.

Most challenging for an attacker is that he has to achieve co-residency on the same core. Since he cannot control the core's scheduling, he has to deal with the risk of his VM being scheduled to another core during a data transmission. Even if the attacker VM is not scheduled away, concurrent VMs might get scheduled between the attacker and victim VM, polluting the TLB's state before the data is retrieved from it. However, the chance of a successful covert channel attack is considerably higher if hyperthreading is enabled. Hyperthreading (or Simultaneous Multithreading, SMT) allows to run instructions of multiple threads (usually two) on a single core at the same time [40]. The physical execution resources are shared and the architectural state[3] is duplicated for the two *hyperthreads*. As a result, enabling hyperthreading is highly beneficial because the attacker

[2] Phoronix Test Suite [31], benchmark: pts/mcperf, method: append, 4 connections. Each configuration ran three times. All three VMs are pinned to the same core of an Intel Xeon E5-2630 v4. They are provisioned with 1 core and 4 GiB main memory each.

[3] The architectural state is the part of the CPU which holds the execution state of a process (e.g., control or general-purpose registers).

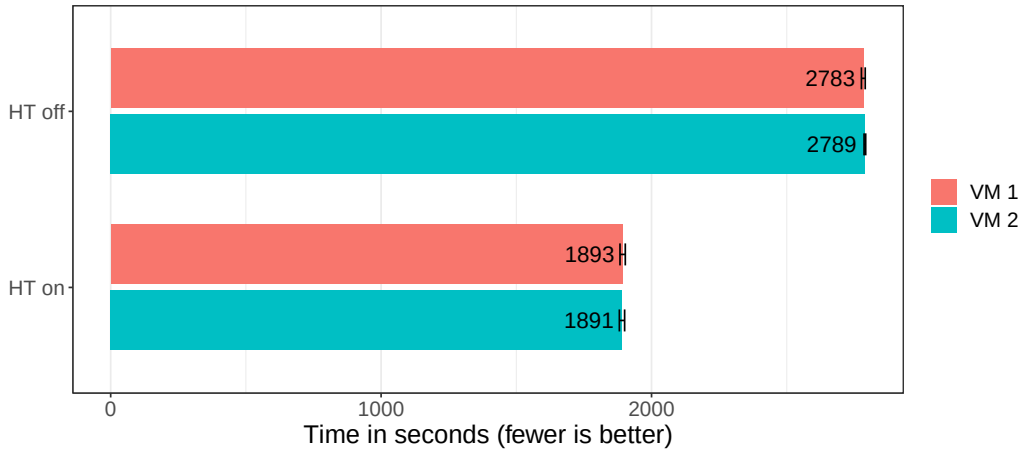


Figure 3.2: Linux kernel compilation benchmark with and without hyperthreading. In both configurations, the computing power is split equally between the VMs. The hyperthreaded system completes the benchmark about 32 % faster.

and victim VM can run simultaneously while sharing the same TLB. We argue that in the cloud providers' interest to offer maximum performance at minimum cost, hyperthreading should be enabled. According to Intel, the first Intel Xeon processors with hyperthreading showed a performance gain of 15-30 % during server-typical workloads [40]. We confirm this by running a Linux kernel compilation benchmark[4] within two single-core VMs on a system with hyperthreading being enabled and disabled. Compiling tasks greatly demonstrate the significance of hyperthreading due to their large degree of parallelization. From the results in Figure 3.2, we see that the hyperthreaded system completes the benchmark about 32 % faster.

While we only focus on single-core VMs in this thesis, achieving core coresidency can further simplified by using VMs with multiple cores (vCPUs). The higher the number of vCPUs, the higher is the chance of sharing a core with the victim VM. The attacker detects this by iterating over all vCPUs, listening for data transmissions.

Based on the preceding considerations, we derive our final attack scenario in which the attacker and victim VM reside on the same core (or on adjacent hyperthreads), with VPID tagging and hyperthreading being enabled.

[4] Phoronix Test Suite [31], benchmark: pts/build-linux-kernel. Each configuration ran three times. Both VMs are pinned to the same core of an Intel Xeon E5-2630 v4. They are provisioned with 1 core and 4 GiB main memory each.

3.2 Monitoring TLB Entries

In order to send data over the TLB, we need to intentionally insert entries and afterwards monitor whether they got evicted or are still cached. In Section 2.2, we presented several techniques to observe caches, with Prime+Probe being the most prevalent. Many cache attacks implement the *probing* phase of Prime+Probe by measuring the latency of memory accesses in order to determine the existence of a cache entry [18, 34, 45]. An exemplary implementation using the timestamp counter as a high-precision timer is depicted in Algorithm 3.1. After the first access of a in line 1, the entry is cached. Then, we wait for a specific time to give other processes the chance to evict the cache entry, either unintentionally (side-channel attacks) or intentionally by a sender (covert channels). Finally, we access a again and keep track of the timestamp counter before and after the access. By taking the difference of the two measurements and comparing them to a predetermined threshold, we distinguish a cache hit from a miss.

Algorithm 3.1: Prime+Probe using `RDTSC`

input: memory address a, threshold $thres$
output: $true$ if access hit cache, $false$ otherwise

```
1: access a                                   ▷ priming phase
2: sleep
3: t1 ← RDTSC                                 ▷ probing phase
4: access a
5: t2 ← RDTSC
6: if t2 − t1 < thres then
7:     return true
8: else
9:     return false
```

While this technique is straightforward to implement and does not require elevated permissions, doing time measurements on microarchitectural level is error-prone. The memory latencies for a cache hit and miss are different on every architecture and also underlie small fluctuations within the same platform. This makes choosing an appropriate threshold a tedious task. Nonetheless, Gras et al. [18] successfully used Prime+Probe with timestamp monitoring in their TLB-based side-channel attack and their proof-of-concept covert channel. However, to mount a high-performance covert channel, we propose a novel approach to accurately monitor TLB entries using the *accessed bit* in page table entries (PTEs).

Algorithm 3.2: Prime+Probe using accessed bits

input: memory address a
output: $true$ if access hit cache (TLB), $false$ otherwise

```
1: access a                                   ▷ priming phase
2: accessed bit ← 0
3: sleep
4: access a                                   ▷ probing phase
5: if accessed bit = 0 then
6:     return true
7: else
8:     return false
```

Algorithm 3.2 outlines our Prime+Probe variant. When a virtual memory address is accessed for the first time (line 1), the memory management unit (MMU) walks the page table to translate the virtual into a physical address (assuming hardware-managed TLBs). If a mapping exists, the MMU caches the translation in the TLB and sets the accessed bit of the corresponding PTE. If we reset the accessed bit in the page table (line 2), it will remain unset until the MMU walks the page table again, that is as long as the translation for the corresponding address is cached in the TLB. Only after the respective entry got evicted, an access to the same address (line 4) triggers a page table walk, which sets the accessed bit again. Therefore, by reading the accessed bit in line 5, we get a reliable way to distinguish a cache hit from a miss.

Although the novel accessed bit-based Prime+Probe variant is more accurate compared to timestamp monitoring, it also requires elevated permissions. Access to page tables is restricted to the kernel, which, given our attack scenario, implies that the resulting covert channel is unidirectional (victim $\rightarrow$ attacker) because the attacker has kernel privileges in his own VM only. For bidirectional communication, privilege escalation has to be achieved on the victim's VM with further exploits.

3.3 Challenges

By definition, a covert channel denotes the absence of direct communication. Thus, for a TLB-based covert channel, communication is solely carried out via insertion and eviction of TLB entries. The receiver has to fetch the data from the TLB during a certain time window in which the sender actively "writes" the data to the TLB. In Figure 3.3, we compare different sender and receiver constellations regarding the timing between send and receive operations. For now,

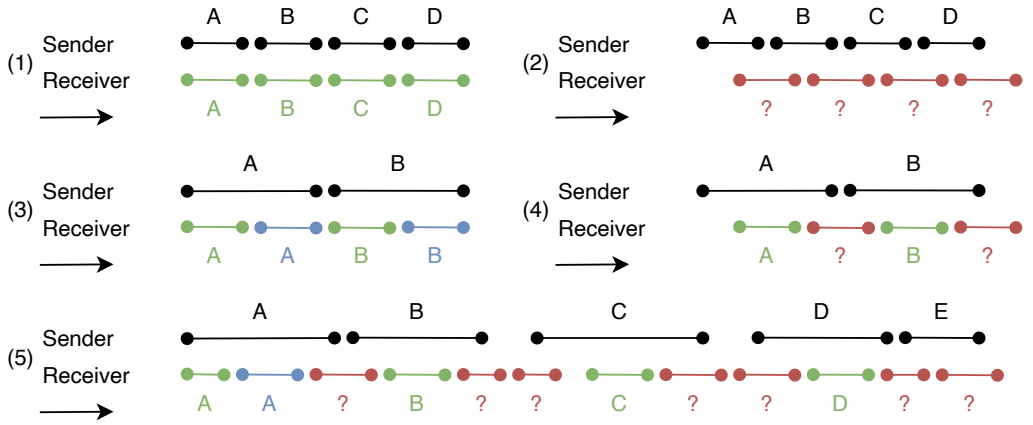


Figure 3.3: Illustration of different sender and receiver timings. For a successful data transmission, the receiver window has to fall completely in the sender window. Successful transmissions in green, redundant (successful) transmissions in blue, erroneous transmissions in red.

we assume that as long as the receiver window falls completely in the sender window, a successful transmission can occur. If a common clock exists, we can synchronize sender and receiver perfectly as shown in (1). However, generally there exists no common clock between VMs of sufficient resolution. For instance, for security and compatibility reasons, the timestamp counter is virtualized and calling `RDTSC` from within a VM is subject to clock deviation, timestamp-counter offsetting, and scaling [8]. Without synchronization, successful data transmission using the same window lengths is no longer possible (2). Therefore, in absence of a common clock, we define windows during which the sender repeatedly writes the same data and at the same time the receiver tries to read the data via sampling. From the Nyquist-Shannon sampling theorem, we know that the sampling frequency has to be more than two times higher than the highest frequency of the original signal [53]. For us, this means that the sender window length has to be more than twice the receiver window length. In an out-of-sync scenario, this guarantees that at least one successful transmission per sender window is possible (4). On the other hand, we might even encounter redundant successful transmissions which can be used for forward error correction (3). On a real system, the sender and receiver window lengths as well as the gaps between consecutive transmissions slightly deviate. If we cannot uphold the sampling theorem under these circumstances, transmission errors will occur (e.g., loss of "E" in Example 5). In accordance with the analysis of Maurice et al. [41], we have to deal with the following types of errors (also see Figure 3.4):

substitution errors The TLB is a shared resource and competitively used by concurrent processes. Therefore, the *priming* entry inserted by the receiver might get evicted unintentionally, causing a bit flip from $0 \rightarrow 1$. If the sender unsuccessfully evicts the entry, we encounter a bit flip from $1 \rightarrow 0$. The interference increases with the overall system load, especially when running memory-intensive tasks.

insertion errors Without further measures, the receiver will continue retrieving data from the TLB, even when the sender is interrupted or descheduled. The received data will either be empty or contain noise caused by concurrent processes.

deletion errors Similarly, when the receiver is descheduled, we are unable to receive data sent during this time.

While substitution errors are individual errors, insertion and deletion errors occur in bursts due to scheduling. Insertion and deletion errors are particularly challenging for unidirectional channels because the receiver cannot inform the sender whether the transmission was successful or to request retransmissions.

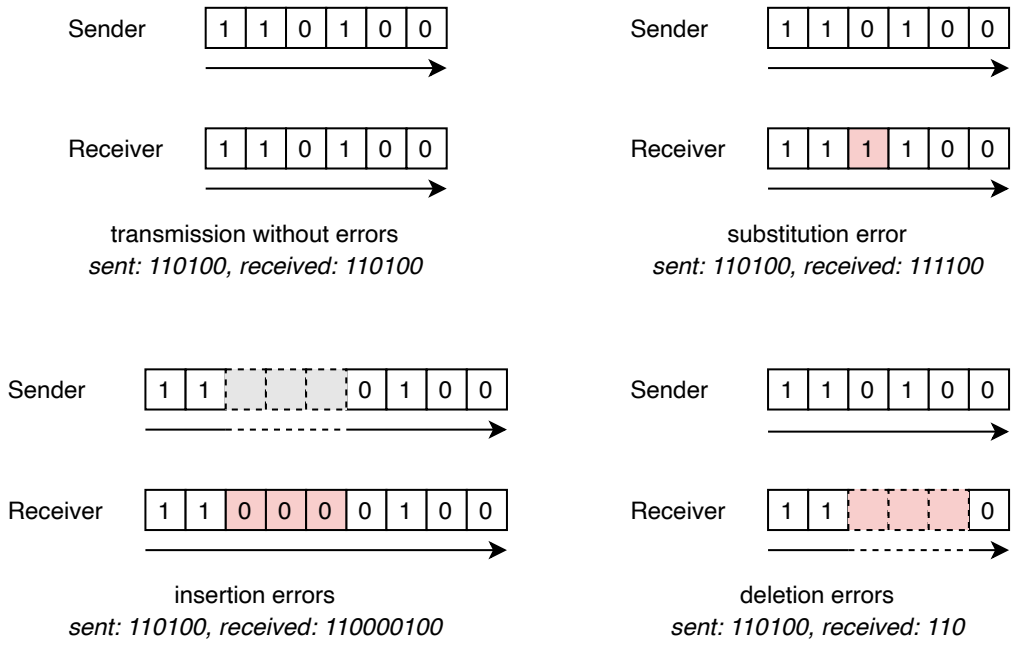


Figure 3.4: Different error types occurring during transmission. Dashed lines represent the time when either the sender or the receiver is not scheduled. As a simplification, only 1 bit is sent per transmission, while in general, each box can be seen as a packet containing multiple bits.

To gain further insight on the noise caused by the TLB usage of concurrently running processes, we repeatedly insert one TLB entry per set and count the number of its evictions while running different tasks for 30 seconds. The measurement and the benchmark process were each running in their own VM, pinned to adjacent hyperthreads on the same core. From the results in Figure 3.5, we summarize that some sets show exceeding utilization (often along

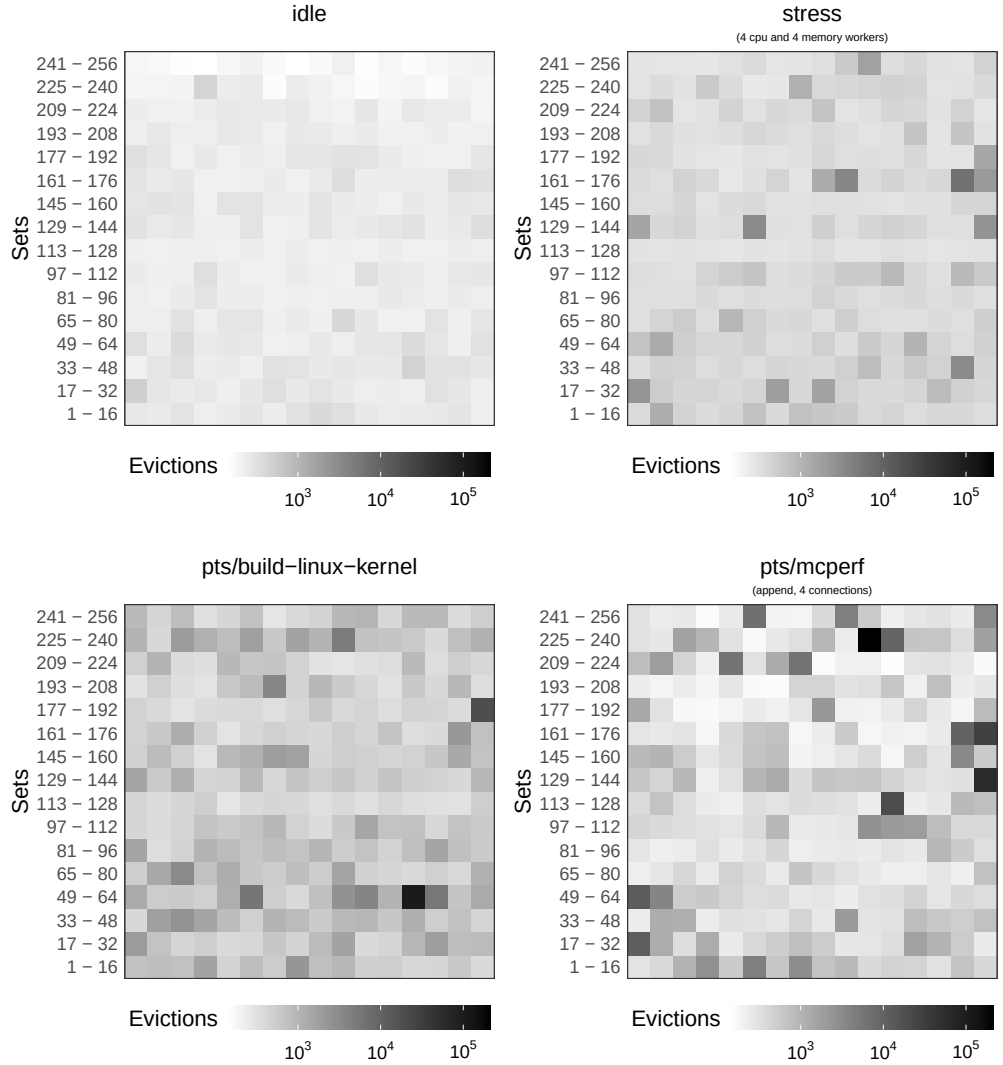


Figure 3.5: Number of evictions visualized for all 256 TLB sets of an Intel Xeon E5-2630 v4 while running different tasks for 30 seconds. Tasks generating a high contrast pattern focus on few sets which show exceeding utilization (often along with their neighboring sets $\rightarrow$ principle of locality).

with their neighboring sets), whereas the overall utilization does not follow a particular pattern. Sets with a high number of evictions are especially prone to substitution errors, that is, bit flips from $0 \rightarrow 1$.

3.4 Conclusion

Caches can be used to build high-performance covert channels with decent bit rates while maintaining a low error rate. However, isolation techniques on microarchitectural level (e.g., Intel CAT) defeat cache-based side-channel attacks and covert channels. Therefore, we shift our focus to the TLB, where these techniques do not apply. Based on the findings of Gras et al. [18], we explore whether high-performance covert channels between VMs, as we know from caches, are possible over the TLB. We argue that achieving co-residency in modern clouds is practical and that VPID tagging as well as hyperthreading are enabled for performance reasons.

To send data over the TLB, we need to intentionally insert entries and afterwards monitor whether they got evicted or are still cached. Similar to the Prime+Probe method from cache attacks, we measure the memory latency with a high-precision timer and use a threshold to distinguish between TLB hits and misses. Furthermore, we propose a novel monitoring technique for TLB entries leveraging the accessed bit in page table entries.

We have to face the challenges of synchronization and noise. To address the synchronization issue in absence of a common clock, we use windows of different length for the receiver and sender in combination with sampling. Due to scheduling and interference of concurrent processes, we encounter insertion, deletion, and substitution errors. To build a reliable covert channel, an appropriate protocol is needed which is able to deal with these challenges.

Chapter 4

Design & Implementation

From the previous chapter, we recall that in order to build a reliable covert channel over the TLB, the issues of synchronization and noise have to be handled by an appropriate communication protocol. Similar to Maurice et al. [41], we lend ideas from wireless protocols and construct a multi-layer protocol consisting of a physical and data-link layer. In this chapter, we describe the design we use to achieve high-performance TLB-based covert channels and outline the overall transmission process.

4.1 Physical Layer

The *physical layer* forms the lowest level of our protocol and is responsible for transmitting raw bits via the TLB. These bits are represented by TLB hits and misses, whereby misses are generated by intentional set evictions. Evicting a set through multiple memory accesses incurs a high CPU utilization which in turn can be detected by monitoring tools or the system admin. We observe that real-world data contains slightly more zero than one bits (see Section A.1). With respect to the attacker's stealth, we define a TLB hit as binary 0 and a miss as 1.

Then, we can send bits over the TLB as follows: The receiver process in the attacker's VM applies Prime+Probe to one memory address. In the majority of cases, we will see a TLB hit because continual memory accesses keep the TLB entry alive. This is interpreted as binary 0 by the receiver. To send a binary 1, the sender process running on the compromised VM actively causes evictions in the shared TLB by accessing a large set of pages. Each new translation will evict one TLB entry, eventually evicting the one belonging to the receiver process. The resulting TLB miss is interpreted as binary 1 by the receiver. In some cases, concurrently running processes or OS-triggered TLB flushes will evict the TLB entry, causing a substitution error ($0 \rightarrow 1$).

As explored in Section 2.1.1, all modern Intel processors employ set-associative TLBs. Given the mapping scheme from memory addresses to TLB sets is known, we are able to flush and monitor each set independently. Therefore, the number of sets corresponds to the number of bits we can send per iteration. According to the TLB organizations we saw in Table 2.1, the number of sets is either 128 or 256, allowing us to send up to 16 or 32 bytes (1 byte $=$ 8 bits) per iteration. We define each group of 16 or 32 bytes as a *packet* of size n. However, we cannot use the whole n bytes for actual data only. In order to detect substitution, insertion, and deletion errors, we need to include additional metadata in each transmission. Per packet, we reserve h bytes for transmission metadata (*header*) and use the remaining $n - h$ bytes for actual data (*payload*). In practice, we use $h = 2$, leaving 14 or 30 bytes for the payload depending on the number of sets in the TLB (128 or 256).

To detect missing or redundant transmissions, each packet is assigned an advancing sequence number. As stated in Section 3.3, the sender window needs to be more than twice as long as the receiver window. Therefore, without a sequence number, the receiver might retrieve the same packet multiple times (insertion error). On the other side, if the sequence numbers of two subsequently received packets do not follow the ascending order, we lost a packet (deletion error). Reorderings cannot occur in our channel because packets are sent sequentially and the TLB only holds one packet at a time. We assign s bits of the header for the sequence number, giving us a total of 2^s possible numbers. In practice, we allot one header byte for the sequence number ($s = 8$). However, we declare only 254 of the 256 possible sequence numbers as valid, reserving 0 (0x00) and 255 (0xFF) to handle two special cases: By ignoring packets with a sequence number of 0x00, we prevent insertion errors when the sender does not send data (e.g., is descheduled). In this case, the receiver's probing entries do not get evicted, resulting in 0 bits due to TLB hits (Example 3 in Figure 4.1). While this is true for idle systems, substitution errors caused by interfering processes might flip a bit in the sequence number, thereby making the packet "valid". Occasionally, the operating system triggers TLB flushes which evict all probing entries, resulting in 1 bits due to TLB misses (Example 4 in Figure 4.1). Therefore, we ignore packets with a sequence number of 0xFF. Valid sequence numbers range from 1 (0x01) to 254 (0xFE) as depicted in (1) and (2) in Figure 4.1.

Since we build an unidirectional channel, we cannot issue retransmission requests. This is problematic with regard to our scenario where substitution errors are likely to occur, resulting in corrupt packets. To overcome this issue, we send each packet a large number of times under the assumption that at least one transmission succeeds without errors. The actual number of times (i.e., the sender window length) depends on the level of interference, but will always be higher than the theoretical minimum of two times the receiver window, thereby

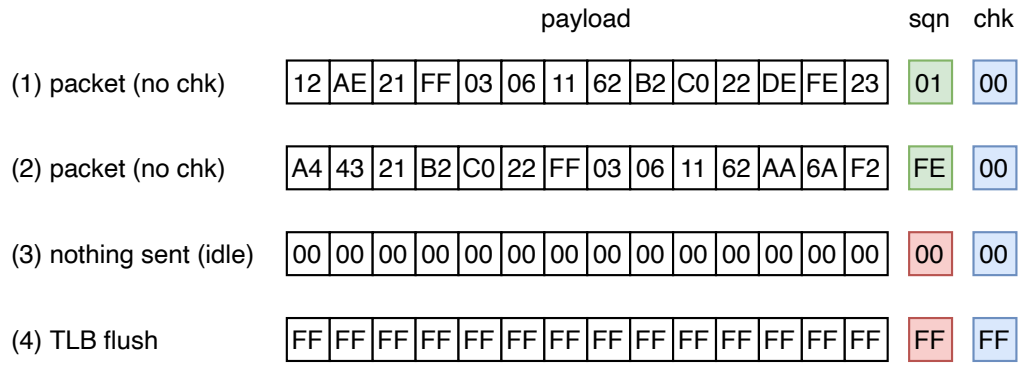


Figure 4.1: Packets on the physical layer of 16 bytes length. The payload occupies 14 bytes, sequence number (sqn) and checksum (chk) 1 byte each. TLB states not intended by the sender are declared invalid by design.

decreasing the maximum bit rate (see Section 3.3). To sort out the packets with substitution errors, we allot the remaining header byte for a checksum computed over the payload and the sequence number. Protecting the sequence number against bit flips is particularly important to reliably detect deletion and insertion errors as stated above. We implement the following checksum algorithms in order to compare them during our evaluation (see Chapter 5) in respect of their behavior under noisy conditions:

- *Berger code*: We use the `POPCNT` instruction to efficiently calculate the number of binary zeros as depicted in Section 2.3.1. Assuming a maximum of 30 bytes for the payload and 1 byte for the sequence number, the maximum value of the Berger code is 248 (= 31 zero bytes) and therefore can be represented by one byte.

- *CRC-8*: We employ a cyclic redundancy check code (see Section 2.3.2) of 8 bits length. For efficiency reasons, we use a lookup table precalculated from the generator polynomial 0x31[1] to compute the one-byte CRC.

- *Custom XOR*: We use this lightweight checksum to analyze whether stronger checksums with higher computational overhead provide better error detection in our application at all. Packets are written iteratively to the TLB, beginning with the first byte of the payload and ending with the sequence number and checksum. We presume that if the first and last byte of a packet are correct, the bytes in between are likely to be correct as well. Therefore, we XOR the first byte of the payload with the sequence number.

[1] As found in the Sensirion SHT75 humidity sensor [52].

In addition, we can approach substitution errors with error correction codes, such as the Hamming code which is capable of correcting one bit (see Section 2.3.3). We experimented with applying an extended $(8, 4)$-Hamming code to the physical layer. Every four data bits are accompanied by four parity bits, allowing the correction of one erroneous bit every eight bits with a total parity of 50 % per packet. However, we observed that whenever substitution errors occur, they quickly exceeded the Hamming code's correction capabilities. Therefore, we decided against using an error correction code on the physical layer, but rather stick to the redundancy introduced by larger sender windows and use an additional layer to deal with the remaining errors.

To summarize, the payload, sequence number, and checksum form a packet on the physical layer. The physical layer we designed is not reliable with regard to correctness and completeness. Substitution errors can occur in such a way that the checksum is still correct. On the other side, under heavy interference, it is unlikely to receive a correct packet during the sender window. If we choose to dismiss erroneous packets, we cannot recover deletion errors due to the lack of a feedback channel.

4.2 Data-Link Layer

We introduce a *data-link layer* on top of the physical layer which provides stronger error detection capabilities and includes redundancy for forward error correction. As the required level of robustness differs for different application scenarios, we use the Reed-Solomon code[2] in our data-link layer due to its scalability (see Section 2.3.4). To keep the computational overhead low, we define the symbol size $r = 8$, which also allows us to intuitively interpret bytes as symbols. The resulting block length is 255 bytes.

Next, we map each block of 255 bytes to packets of the physical layer. While intuitively, we might fill each packet's payload with 14 (or 30) bytes from one block, this approach greatly undermines the scalable error correction capabilities of the Reed-Solomon code. By doing so, we spread each block over 19 (or 9) consecutive packets and the loss of a few packets (e.g., due to a burst error) quickly exceeds the correctable limit as we have to recover 14 (or 30) bytes per lost packet. Instead, we spread each block over 255 consecutive packets, embedding one block byte in each packet as depicted in Figure 4.2. This way, the number of parity bytes is directly reflected in the number of packets containing parity and we can choose their amount depending on the required level of robustness. For instance, if we define 64 of 255 bytes within a block as parity, we send

[2] Using the libfec implementation by Phil Karn [27].

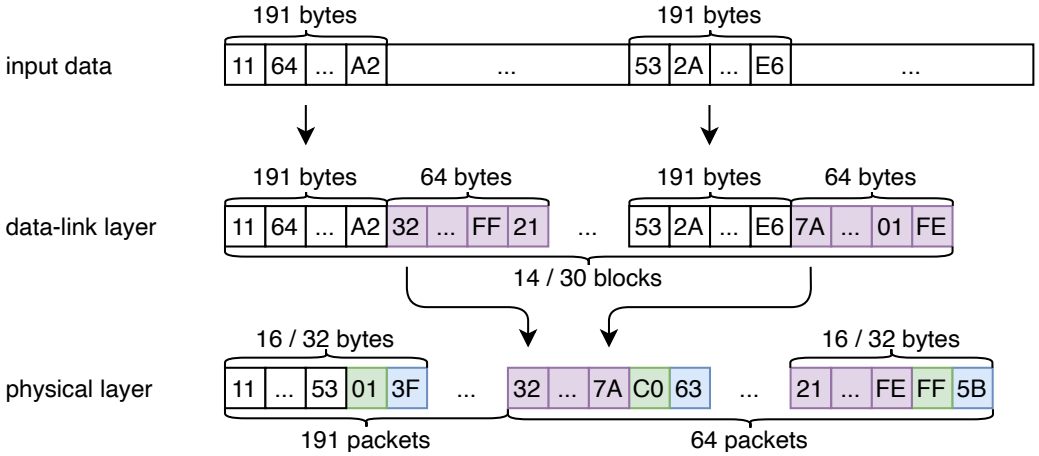


Figure 4.2: Illustration of the multi-layer protocol. On the data-link layer, subsequent segments of the input data are embedded in Reed-Solomon blocks. On the physical layer, we spread each block over 255 packets, thereby sending 14 (or 30) blocks in parallel. Parity bytes are colored purple. The physical packets are structured as depicted in Figure 4.1 with the sequence number representing the offset within a block.

191 packets of data symbols followed by 64 packets containing parity symbols. In Section 2.3.4, we looked at the error detection and correction capabilities of the Reed-Solomon code: With 64 bytes parity per block, we are able to correct up to 32 and detect up to 64 erroneous bytes at unknown positions. Furthermore, we can correct up to 64 errors if we know their offsets within a block (erasures). By spreading each block over 255 packets, these numbers translate directly to packets: The 64 packets containing parity allow correction of up to 32 and detection of up to 64 corrupt packets every 255 packets. Because the block offsets are implicitly encoded in the sequence number, we treat lost (or discarded) packets as erasures and are able to recover up to 64 missing packets every 255 packets.

Since we only use one byte of the packet payload per block, we transfer 14 blocks in parallel to take full advantage of the payload size in the case of a 14 bytes payload, respectively 30 blocks for a 30 bytes payload. It is important to mention that overall, compared to the consecutive mapping we first mentioned, we still send the same amount of parity with the data. However, by distributing the blocks over more packets, we increase the robustness and use the Reed-Solomon code to its full potential.

In Section 4.1, we defined only 254 valid sequence numbers to detect the special cases of idle state and TLB flushes. With the data-link layer enabled, we declare 0xFF valid as well to accommodate the need for 255 sequence numbers to represent all byte offsets within a block. Since we cannot use 0xFF for TLB flush

detection anymore, we use the result of a bitwise AND operation[3] to determine whether all bits of a packet are one. Though a small chance exists that we dismiss valid packets (with 0xFF bytes only), we can afford dropping these packets intentionally because of the Reed-Solomon error correction.

4.3 Transmission Process

The first step to transmit data via the TLB is to allocate memory on the sender and receiver side (see Figure 4.3). The TLB caches translations from virtual to physical memory at page-level, that is, each memory address within a single page points to the same TLB entry. Therefore, we need a pool of pages to generate TLB entries in specific sets. The base address of a page has to be chosen according to the mapping scheme (see Section 2.1.1), so that the receiver can insert one priming entry per set and the sender is able to evict all entries of a set. To evict a TLB set, the sender has to access at least n pages pointing to the same set, where n is the number of ways in a set. As a result, for a TLB organized in 256 sets with 6 ways each, 256 pages are required for the receiver and at least 1536 pages for the sender. Assuming 4 KiB pages, this translates to 1 MiB allocated memory on the receiver side, respectively 6 MiB on the sender side. We use the `mmap` system call to allocate memory at a fixed virtual address. On Linux, the read-only memory we allocate is mapped to the zero page[4] and therefore does not count to the memory consumption of the process which is beneficial to remain unnoticed on the victim system.

After allocating memory, we are ready to send packets. Depending on whether we use the data-link layer or not, we write the data directly in the payload or wrap it into Reed-Solomon blocks as described in Section 4.2. To send a packet, we iterate over its bits and for each binary 1, we evict the receiver's priming entry by evicting the whole TLB set. The sender window translates to the number of repetitions of this procedure for each packet. At the same time, the receiver repeatedly polls and processes packets using Prime+Probe. We iterate over all TLB sets and set the associated packet bit depending on whether we encountered a TLB hit or miss. We use the monitoring methods from Section 3.2. Timestamp-based Prime+Probe monitors TLB entries by measuring the access latency with `RDTSC` in combination with a predetermined threshold. To implement accessed bit-based Prime+Probe, we use a custom Linux kernel module to read and reset the accessed bit of a page table entry. We discard packets with an invalid sequence number or a non-matching checksum. When

[3] For efficiency reasons, the calculation is done on 64-bit words.

[4] A page filled with zeros. The Linux kernel uses it to save memory by mapping virtual pages containing only zeros to the same physical frame [56, p. 755].

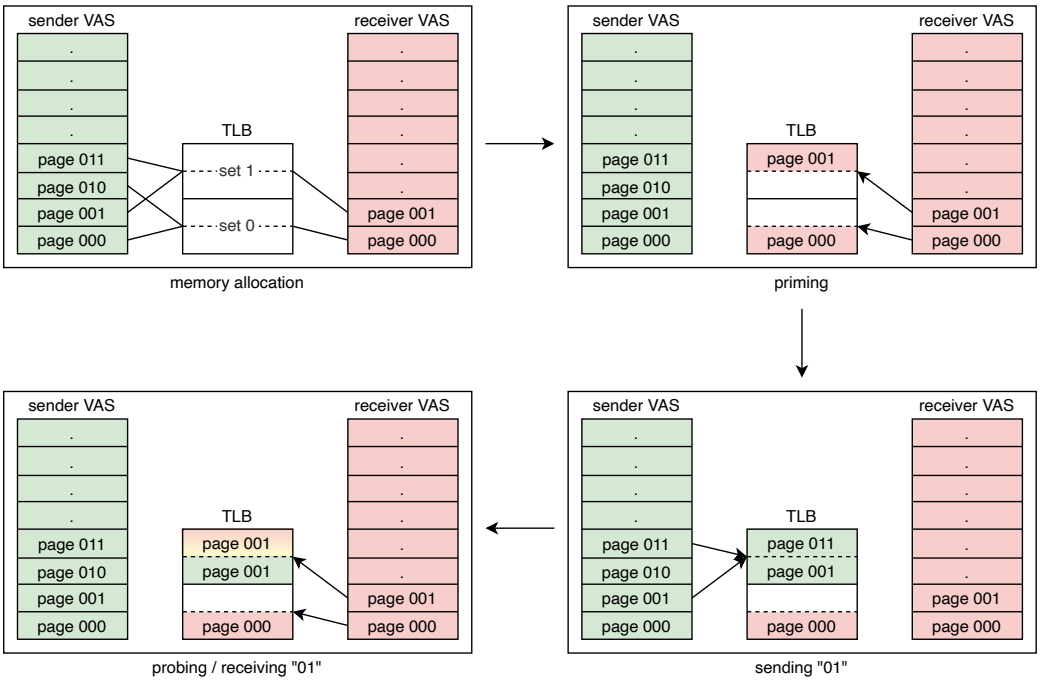


Figure 4.3: Simplified transmission process. VAS = virtual address space. The TLB has 2 sets with 2 ways each and uses a linear mapping (set number = page number modulo 2). The green and red color can be seen as TLB tagging. First, the sender and receiver allocate memory so that the sender is able to evict all TLB sets, whereas the receiver only needs to insert one priming entry per set. Each set maps to exactly one bit to be sent. To send a binary 1, the sender evicts all entries of the corresponding set including the receivers' priming entry. To send a binary 0, the sender leaves the corresponding set untouched. During the probing phase, the receiver "reads" the sent bits by interpreting TLB hits and misses as binary 0's and 1's.

receiving a correct packet, we write its payload to the output file. However, the latency of the I/O system call is critical for the receiver because in our scenario with single-core VMs, the receiver process cannot receive packets while the file is being written. Therefore, we buffer the payloads and flush the buffer at a low frequency to keep receiver preemptions at a minimum. In a scenario with multi-core VMs, it is possible to write the data asynchronously without interrupting the receiver at all. To initiate the end of a transmission, the sender sends a particular data stop packet. To make the data stop signal more robust, we send the packet for a longer sender window and require the receiver to retrieve multiple data stop packets before leaving the receiver loop. If the data was sent Reed-Solomon encoded, we decode it and correct errors after the transmission has ended.

4.4 Conclusion

In this chapter, we designed a two-level protocol which we implement in our high-performance TLB-based covert channels. The physical layer is in charge of transmitting raw packets via the TLB. A TLB hit represents a binary 0, a TLB miss a binary 1. To overcome the issues of synchronization and noise (see Section 3.3), we send each packet a large number of times and include a sequence number and checksum to distinguish between successful and erroneous transmissions. The data-link layer builds on top of the physical layer and leverages the Reed-Solomon code to provide stronger error detection and forward error correction which is essential in our unidirectional channel scenario.

Before transmitting data, we allocate pages with regard to the TLB mapping scheme, so that we are able to generate TLB entries in specific sets. The receiver needs to insert one entry per set (priming entry), while the sender requires multiple pages mapping to each set in order to force the eviction of the priming entry by filling the whole set with its own entries. The sender constructs packets as specified by our protocol and repeatedly evicts TLB sets accordingly. Meanwhile, the receiver polls packets using Prime+Probe and writes the payloads to an output file until the sender signals the end of the transmission.

In our theoretical design, we left several decisions open such as the actual checksum algorithm, the amount of parity symbols for the Reed-Solomon code, or the TLB monitoring mechanism. In the following, we conduct experiments to assess our protocol design considering the objective of achieving high-performance TLB-based covert channels, thereby identifying the most reliable configurations for different scenarios.

Chapter 5

Evaluation

Based on our findings from Chapter 2 and our analysis in Chapter 3, we proposed a two-layer covert channel design in Chapter 4. Our design involves several parameters such as the TLB monitoring mechanism (i.e., timestamps vs. accessed bits), the checksum, and the sender window length. In this chapter, we evaluate this design with respect to our goal of building a high-performance TLB-based covert channel. We begin with an explanation of the methodology which we use throughout this chapter. Next, we study how different configurations of our design perform when transmitting real-world data such as documents and multimedia contents. To evaluate the robustness of our channel, we introduce different levels of interference while monitoring the bit and error rate. We conclude by discussing our results.

5.1 Methodology

To reproduce a realistic public cloud scenario, our evaluation platform is equipped with an Intel Xeon E5-2630 v4 processor. This and other Intel processors of the Broadwell E5 product line are also found in Google's [17] or Amazon's [2] cloud computing portfolio. It features 10 physical cores (20 threads due to hyperthreading) with a base clock of 2.20 GHz and maximum turbo frequency of 3.10 GHz [9]. As stated in Section 2.1.1, processors following the Broadwell microarchitecture employ a set-associative TLB with multiple levels. The STLB holds 1536 4 KiB page translations, organized in 256 sets with 6 ways each. Virtual addresses are mapped to sets by XORing a subset of the address bits (complex-mapped). Our system is equipped with 64 GiB DDR4 main memory with an operating speed of 2133 MHz. We use an internal SATA III solid state disk of 256 GB size for secondary memory. All components are mounted on a Supermicro X10SRi-F server motherboard.

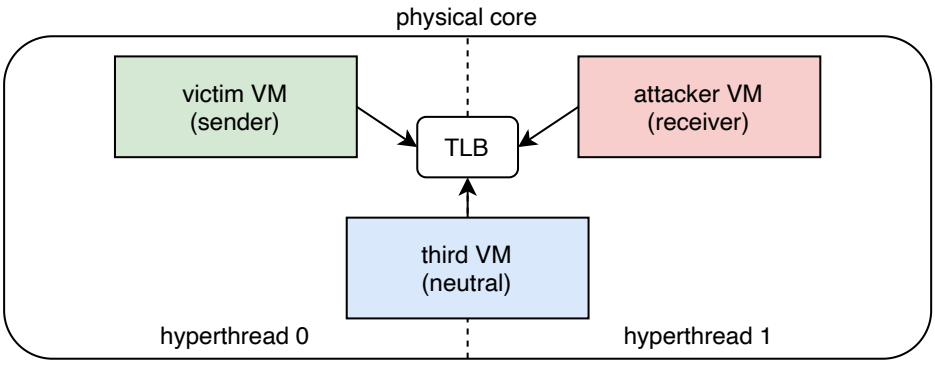


Figure 5.1: The VM setup used in our evaluation. Victim and attacker VM are pinned to adjacent hyperthreads. A third (neutral) VM is pinned to the same core, but can be freely scheduled between the hyperthreads.

The host machine and all VMs run Ubuntu Server 20.04 LTS [37] with Linux kernel 5.4. We choose Linux Kernel-based Virtual Machine (KVM) [28] as virtualization solution. We followed the default install instructions and modified neither kernel nor KVM. All VMs are allotted 1 vCPU, 4 GiB main memory, and 32 GB disk space each. Attacker and victim VM are pinned to the adjacent hyperthreads of the 5th core. To simulate a multi-tenant environment, we retain a third VM (pinned to the same core) which we use to introduce additional interference. Figure 5.1 illustrates the VM setup. We connect to the VMs via SSH and automate the test scenarios with scripts.

The general flow of a single test run is as follows: If we want to introduce interference, we start the respective processes on the VMs and wait a fixed amount of seconds to make sure the processes are up and running. Then, we start the receiver process on the attacker VM followed by the sender process on the victim VM. After the transmission completed, we compare the received file to the sent file and compute the metrics. We also define a timeout to catch incomplete transmissions. Each test is run 10 times to ensure statistical reliability.

A configuration is uniquely defined by the following set of parameters:

Architecture Specifies the TLB organization we built the channel for. This also determines the maximum packet size. We implemented support for Intel Broadwell and Skylake processors.

Packet Size By default, we use all available TLB sets. However, to study the implications of having less TLB sets available, we allow manual control of the packet size in multiples of 8 bytes (32/24/16/8). The higher the packet size, the higher is the maximum bit rate.

Sender Window This defines how often each packet is (repeatedly) sent. Longer sender windows offer more redundancy and increase the channel's reliability, but also decrease the maximum bit rate.

Number of Evictions Specifies the number of evictions per set during the send operation. We have to evict at least as many entries as there are ways in a TLB set to ensure that we evict the receiver's probing entry. Choosing a higher value increases the channel's reliability at the cost of the maximum bit rate.

TLB Monitoring We monitor TLB entries either via timestamps or accessed bits (see Section 3.2).

RDTSC Threshold When using timestamp-based TLB monitoring, this defines the threshold in CPU cycles to be used to distinguish between a TLB hit and miss.

RDTSC Window To increase the reliability of timestamp-based TLB monitoring, we can probe each entry multiple times. If at least one probe during the window exceeds the threshold value, we interpret this as a TLB miss (binary 1).

Checksum The checksum we use for packets on the physical layer. As stated in Section 4.1, we evaluate Berger Code, CRC-8, and a custom XOR error detection code.

Amount of Reed-Solomon Parity The number of bytes in each Reed-Solomon block reserved for parity. According to Section 4.2, this corresponds to the number of parity packets (i.e., packets containing parity bytes only) per 255 packets.

Except for the sender window, all parameters are implemented as compile-time constants to leverage effects caused by compiler optimizations such as loop unrolling.

We transfer different types of data during our evaluation: To represent textual data, we use a plain text file holding the German version of the Book of Genesis [38] with a total size of 202 kB. As image data, we take an (uncompressed) bitmap image file (shown in Figure 5.2) with a total size of 3.2 MB. We also transfer an MP3 audio file of 1.1 MB size to examine whether our channel is robust enough to send compressed data (where data loss is critical). If not stated otherwise, we transmit the text file.

To introduce different levels of interference, we run benchmarks concurrently to the covert channel. VM instances are often used as web servers and therefore,

Figure 5.2: The (original) image used in our evaluation.

we select the nginx-based pts/nginx benchmark of the Phoronix Test Suite [31] to represent the average scenario. From the same suite, the pts/pmbench benchmark is a Linux paging and virtual memory benchmark, making it a memory-intensive workload. Configured with a single worker thread and a read-write ratio of 50 %, it forms the worst-case scenario we assume. In addition, we wrote a microbenchmark ("disturb") which exhibits a deterministic memory access pattern. It allocates a 128 MiB byte array and iterates repeatedly over its entries with a 4 KiB stride, thereby accessing the underlying pages in ascending order. If not stated otherwise, we run these benchmarks on the third (neutral) VM.

5.2 Physical Layer

We start by evaluating the physical layer individually with regard to the bit rate and the error rate. We only consider the *effective* bit rate, that is, the number of correct bytes sent over the transfer duration. The transfer duration is defined as the time in seconds between the first received packet and the end of the receiving process (including Reed-Solomon decoding if applicable). To calculate the error rate, we take the percentage of correct packets received regarding the number of packets sent and subtract it from 100 %. Both metrics are calculated after error correction has been applied. To summarize, the bit rate determines the transmission speed, whereas the error rate indicates the quality of a transmission.

In the left plots of Figure 5.3, we vary the sender window and the number of evictions in idle state using accessed bit-based TLB monitoring. CRC-8 is used as checksum. We see that increasing the sender window from 80 to 100 increases the bit rate by 88 % while reducing the error rate by 66 %. From sender window 100 to 120, the error rate is reduced by another 62 % while the bit rate stays

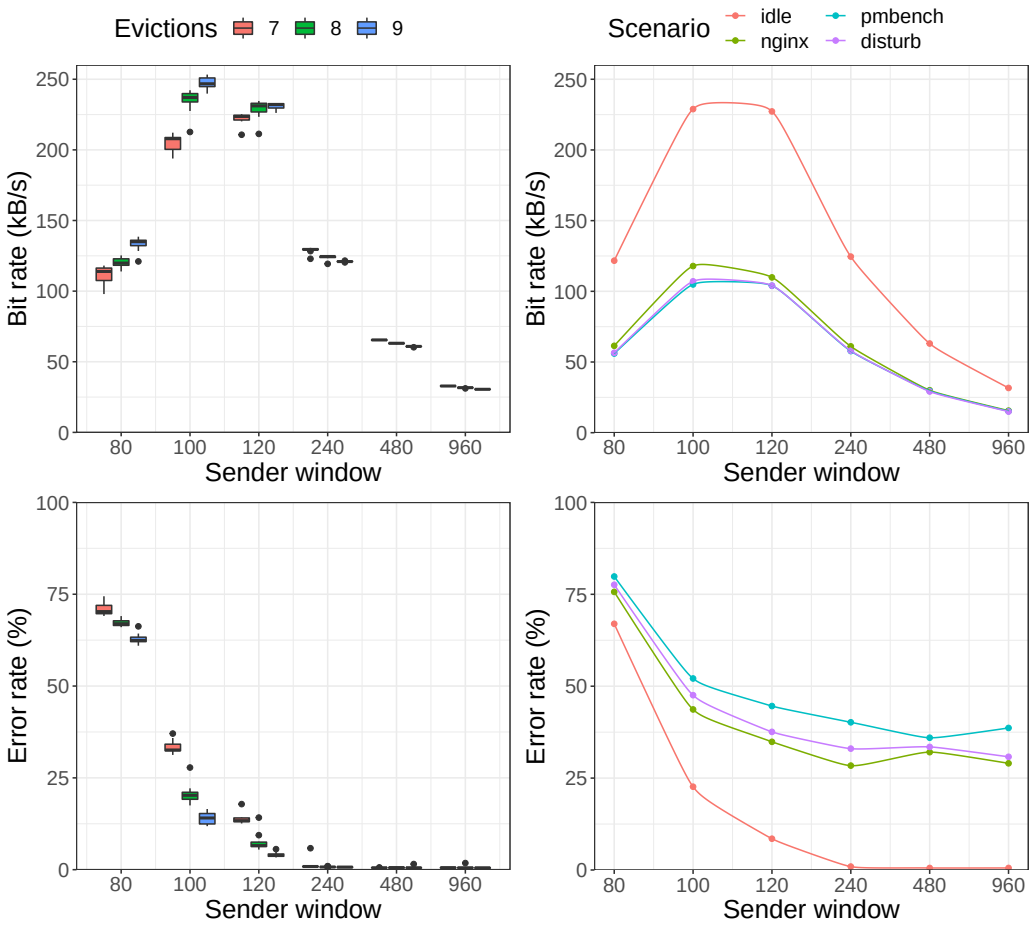


Figure 5.3: Effect of sender window and number of evictions on bit rate and error rate using accessed bits. The left plots show the idle scenario only. The points in the right plots are averaged over all eviction counts. An increase of the sender window leads to less errors which initially increases the bit rate, until a certain point of redundancy has been reached. Beyond this point, the bit rate falls without any improvement of the error rate.

about the same. The maximum bit rate of 253.3 kB/s is reached with a sender window of 100 and 9 evictions per set at an error rate of 12.2 %. A further increase of the sender window causes the bit rate to fall linearly with the sender window (i.e., 2x sender window $\rightarrow$ 0.5x bit rate). For sender windows 240 and above, the error rate is below 1 % and does not improve any further. In particular, we did not achieve an error-free transmission. Increasing the eviction count has a positive effect for sender windows lower than 120, whereas for sender windows higher than 120, the bit rate decreases with a higher number of evictions.

As stated in Section 5.1, longer sender windows offer more redundancy and increase the channel's reliability, but also decrease the maximum bit rate. For we only consider the effective bit rate, the bit rate gain we get from a more reliable transmission (with less errors) outweighs the cost of more send repetitions per packet when we increase the sender window from 80 to 100. However, for sender windows above 240, the amount of redundancy saturates the channel, that is, we send packets with more repetitions than needed for a stable transmission and reduce the bit rate without any benefit. A higher number of evictions per set increases the amount of time needed for each packet bit to be written. Regarding the receiver's sampling (see Section 3.3), larger sender window lengths improve the channel's performance in unsaturated state. On the other hand, choosing a higher eviction count introduces additional redundancy which is unnecessary once the channel is saturated. Therefore, we have to balance the sender window and the number of evictions to find the optimum.

In the right plots of Figure 5.3, we see the bit rate and error rate averaged over all eviction counts with interpolating curves for different scenarios. Compared to idle state, we only get about half the bit rate when running another process along our covert channel due to the split of processing resources. While the correlation between sender window and bit rate are equal for all scenarios, we get a different behavior regarding the error rate. For sender windows 240 and above, the error rate does not decrease significantly and stays above 25 %. We identify two counteracting reasons for this: Firstly, substitution errors occur in such a way that the checksum is still correct because the checksum's detection capabilities are limited. Secondly, heavy interference causes errors in all packet repetitions of a sender window. While increasing the sender window leads to a higher chance of receiving a packet without errors, we also increase the risk of wrongly accepting corrupt packets (see bump in error rate of the nginx scenario for a sender window of 480).

In Figure 5.4, we compare the different checksums on the physical layer in idle state. In all sender windows, we observe that Berger Code yields unreliable results for an eviction count of 7. If the number of evictions is higher than 7, the Berger Code provides robustness similar to CRC-8. However, for a sender window of 120, CRC-8 slightly benefits from an eviction count increase, whereas the performance of Berger Code already decreases. We measured the calculation times of each checksum implementation and realized that CRC-8 takes the most time, followed by Berger, with the fastest being the custom XOR checksum[1]. Therefore, we conclude that faster checksums reach channel saturation with a lower sender window than slower checksums. We do not see this effect for the

[1] In our implementation, custom XOR is about 2x faster than Berger and Berger is about 3x faster than CRC-8. The numbers might differ for other implementations.

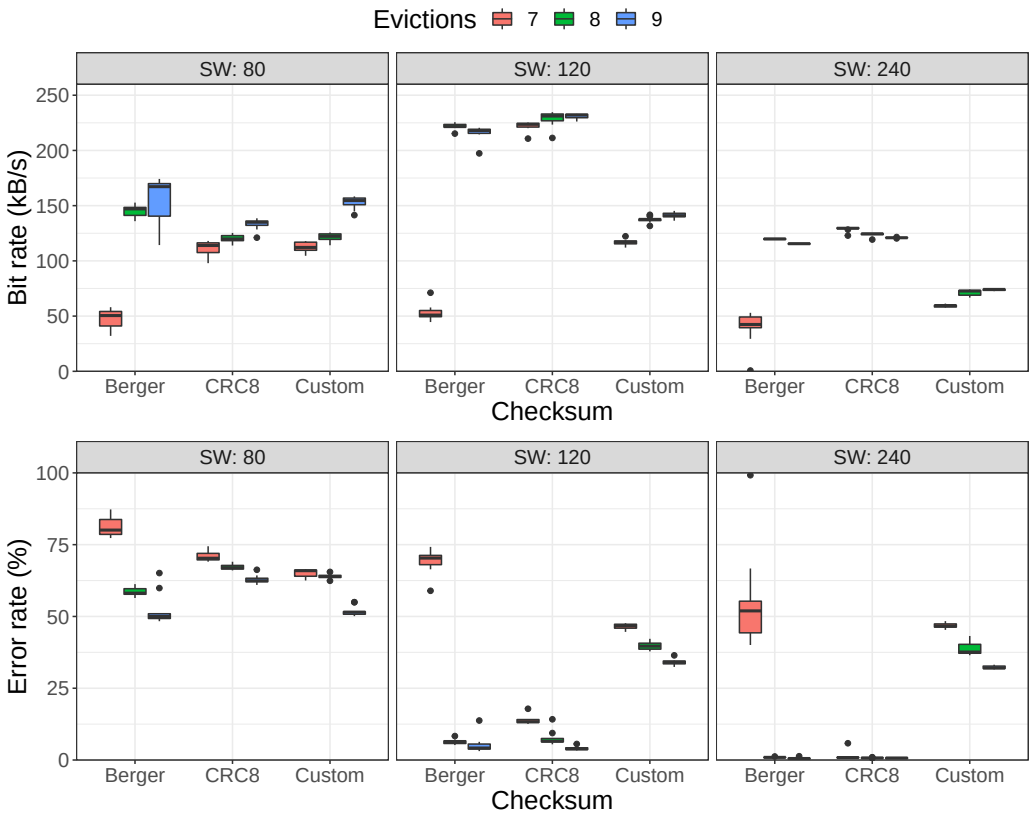


Figure 5.4: Effect of checksum and number of evictions for different sender windows using accessed bits. Berger Code reaches channel saturation with a lower sender window than CRC-8. The custom XOR checksum provides the weakest detection capabilities and therefore yields a high error rate even for long sender windows.

custom XOR checksum because its weak detection capabilities lead to a large amount of wrongly accepted packets.

Moving to timestamp-based TLB monitoring, we examine the effect of the eviction count and the RDTSC threshold for different sender and receiver windows in Figure 5.5. The measurements are done in idle state with CRC-8 being used as checksum. Similar to the accessed bit-based channel, the interplay of eviction count and sender window defines when the channel is saturated. However, we empirically found that the kind of redundancy needed for a reliable channel is different: The eviction count is about twice as high than theoretically needed (i.e., the number of ways per TLB set), leading to larger sender window lengths. As a result, we need to introduce less redundancy through packet repetitions (sender window). On the receiver side, we require a (RDTSC) receiver window

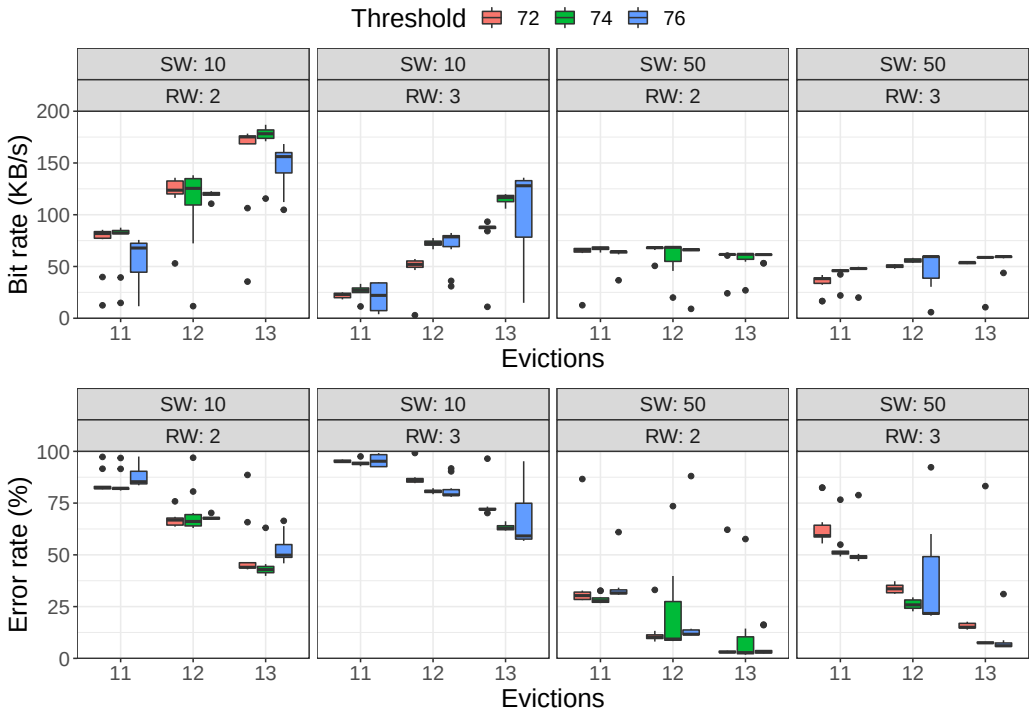


Figure 5.5: Effect of number of evictions and RDTSC threshold for different sender windows (SW) and receiver windows (RW) using timestamps.

of at least 2 to achieve a reliable transmission. Reducing the receiver window to 1 leads to bit rates in the single-digit range. When increasing the receiver window from 2 to 3, we cannot gain further robustness but rather increase the error rate due to our implementation: The chance of measuring at least one TLB miss increases with higher receiver windows, eventually leading to packets with binary 1's only. We can counteract this effect by choosing a higher threshold to lower the sensitivity of each measurement. For instance, increasing the threshold from 74 to 76 for a receiver window of 3 achieves a better channel performance on average. However, we also see an increase in the spread of the measurements due to the lower sensitivity, as TLB misses occasionally remain undetected. On the other side, a threshold chosen too low leads to an increased number of substitution errors ($0 \rightarrow 1$) caused by oversensitivity to TLB misses. We achieve a maximum bit rate of 186.9 kB/s with a sender window of 10, a receiver window of 2, a threshold of 74, and an eviction count of 13 at an error rate of 40.2 %. We achieve a minimal error rate of 1.7 % with a sender window of 50, a receiver window of 2, a threshold of 74, and an eviction count of 13 at a bit rate of 62.3 kB/s. During the rest of our evaluation, we set the receiver window to a fixed value of 2 and the threshold to 74 for timestamp-based channels.

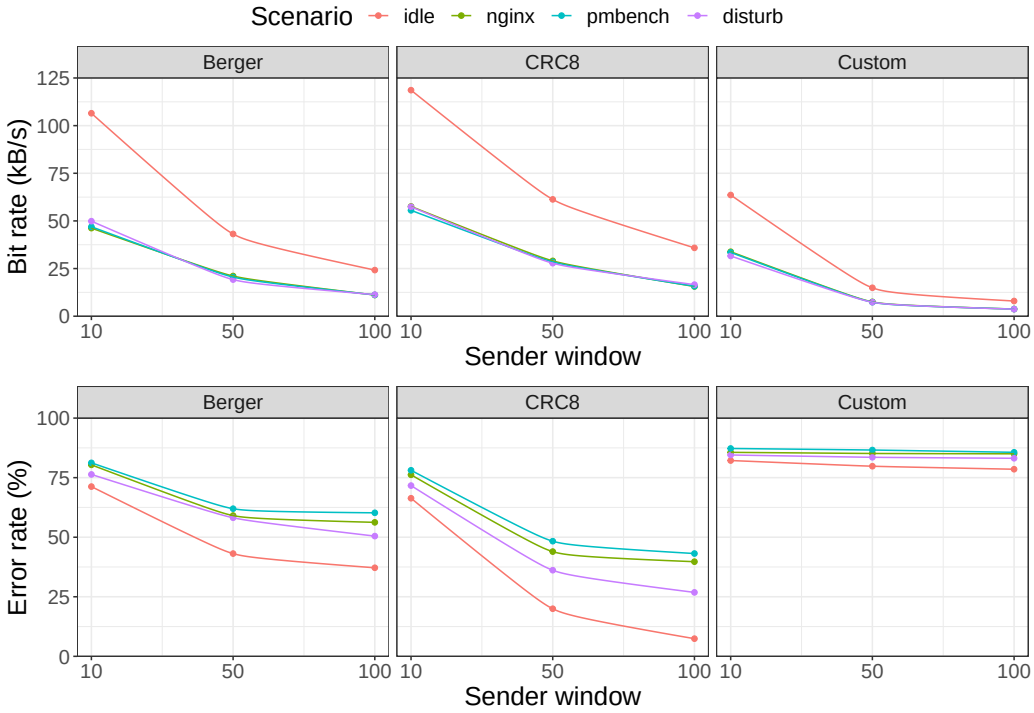


Figure 5.6: Effect of checksum for different sender windows and scenarios using timestamps. CRC-8 is a stronger checksum than Berger, whereas the custom XOR checksum is the weakest. The weaker a checksum is, the less we benefit from an increase of the sender window.

In Figure 5.6, we compare the different checksums on the physical layer in various scenarios using timestamps. Per interpolation point, we average the bit rate and error rate over all eviction counts. Similar to our observation for the accessed bit-based channel, the bit rate drops by half when other processes are running concurrently. Because doing time measurements on microarchitectural level is error-prone and incurs a high amount of substitution errors, the reliability of the timestamp-based channel strongly correlates with the error detection capabilities of the checksum. The custom XOR provides weakest error detection, therefore, all packets are accepted as long as their sequence numbers are valid. Regardless of the scenario, an increase of the sender window does not improve the channel performance and the error rate stays above 75 %. In contrast, the error rate for CRC-8 decreases when increasing the sender window which implies that many packets are discarded due to a mismatching checksum at first and by increasing the send repetitions, we receive more correct packets. Of course, this effect diminishes when increasing the level of interference introduced by concurrent processes. The Berger Code performs equal to the CRC-8 for a sender

window of 10. For higher sender windows, more corrupt packets pass through the Berger Code unnoticed. We conclude that CRC-8 is a stronger checksum than Berger, whereas the custom XOR checksum is the weakest.

While evaluating accessed bit-based and timestamp-based channels, we observe that different eviction techniques are required to achieve maximum bit rates for both channel types. In timestamp-based channels, we use Algorithm 5.1 which intuitively implements the required sender behavior. We evict each TLB set belonging to a packet bit which is set to 1 and repeat the whole procedure as specified by the sender window. To evict a TLB set, we access as much pages as specified by the number of evictions. Given an eviction count of 9, we allocate 9 pages per set during the sender initialization and to evict a TLB set, we access the pages belonging to the set as follows: 1, 2, 3, 4, 5, 6, 7, 8, 9. However, if the TLB has 6 ways per set, we start evicting our own entries after page 6 has been accessed. The entries we evicted cause TLB misses during the next send iteration, whereas normally only entries which were replaced by concurrent processes (e.g., the receiver) cause TLB misses. These ongoing TLB misses significantly slow down the send operation, leading to a more stable transmission at the cost of maximum bit rate. Interestingly, for accessed bit-based channels, we achieve the same stabilizing effect without introducing costly TLB misses. As seen in Algorithm 5.2, we limit the page number to the number of ways per TLB set with a modulo operation (line 5). For a TLB with 6 ways per set and an eviction count of 9, we allocate 6 pages per set during the sender initialization and access them as follows: 1, 2, 3, 4, 5, 6, 1, 2, 3.

Algorithm 5.1: Send packet in timestamp-based channels

input: packet $p = (p_0 \dots p_{n-1})_2$, where $n = number\ of\ sets$

```
1: for w from 0 to sender window − 1 do
2:     for s from 0 to number of sets − 1 do
3:         if p_s = 1 then
4:             for e from 0 to number of evictions − 1 do
5:                 access e-th page pointing to set s
```

Algorithm 5.2: Send packet in accessed bit-based channels

input: packet $p = (p_0 \dots p_{n-1})_2$, where $n = number\ of\ sets$

```
1: for w from 0 to sender window − 1 do
2:     for s from 0 to number of sets − 1 do
3:         if p_s = 1 then
4:             for e from 0 to number of evictions − 1 do
5:                 access (e mod number of ways)-th page pointing to set s
```

Compared to accessed bit-based channels, the amount of outliers is significantly higher for timestamp-based channels and while for accessed bits, the outliers were close to the other measurements, they represent drastic performance losses in case of timestamps. Apart from the outliers, the required higher eviction count is another indicator that timestamp-based channels are less reliable than their accessed bits counterpart. In our unidirectional scenario where retransmission requests are not possible, this makes accessed bit-based channels more suitable to achieve the goal of building a high-performance covert channel. Nonetheless, the physical layer by itself is not sufficient to accomplish an error-free transmission as we demonstrate with the following example: We send a plain text file (Book of Genesis) with a sender window of 120, eviction count of 8, and CRC-8 checksum using accessed bits in idle state. The resulting bit rate is 232.8 kB/s at an error rate of 6.5 %. The most common errors are lost packets, resulting in lost sentence fragments, and wrongly accepted packets which insert garbled characters into sentences. Examples are given in Figure 5.7. However, natural text contains a decent amount of redundancy by itself and therefore, we are able to recover most of the sentences by looking at their context.

lost packet

```
[snd] 1. Und es begab sich darnach, daß sich der Schenke des Königs in
Ägypten und der Bäcker versündigten an ihrem Herrn, dem König von
Ägypten.

[rcv] 1. Und es begab sich darnach, daß sich der Schenke des Königs in
Ägypten und der Bäcker versündigtvon Ägypten.
```

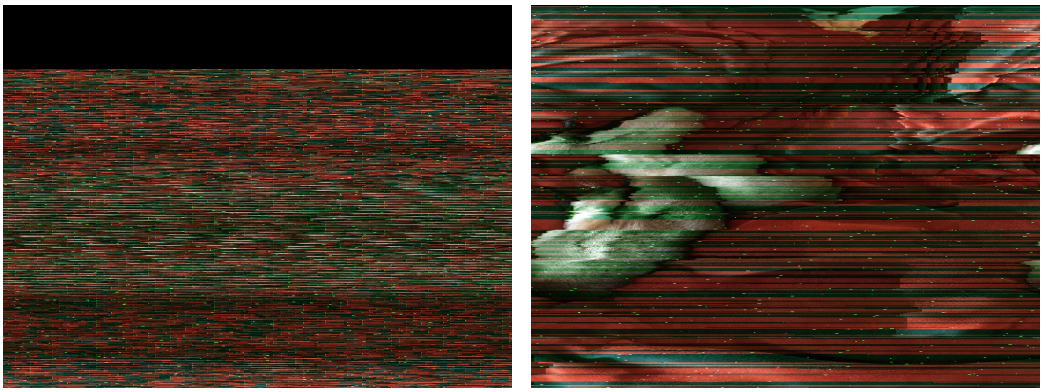
corrupt packet

```
[snd] 2. Und Pharao ward zornig über seine beiden Kämmerer, über den
Amtmann über die Schenken und über den Amtmann über die Bäcker,

[rcv] 2. Und Pharao ward zornig über seine beiden a?d,oro{o über
weonemvminon?Kämmerer, über den Amtmann über die Schenken und über den
Amtmann über die Bäcker,
```

Figure 5.7: Effects of lost and wrongly accepted (corrupt) packets on a plain text file.

This is not possible for all kinds of text (e.g., password files) or other kinds of data such as images. Using the same covert channel configuration, we send the sample image with a resulting bit rate of 177.0 kB/s at an error rate of 17.4 %. The resulting output image is shown on the left in Figure 5.8. For the original image is unrecognizable in the output image, we increase the sender window to 240, resulting in a bit rate of 107.7 kB/s at an error rate of 0.8 %. The resulting output image is shown on the right in Figure 5.8 and even though only about



(a) Error rate: 17.4 %. (b) Error rate: 0.8 %.

Figure 5.8: The sample image sent via the physical layer. Deletion errors manifest themselves in a missing line of pixels, thereby shifting all following pixels. Corrupt packets (e.g., substitution errors) cause green pixel bursts.

25 kB of the total 3.2 MB are corrupted, the image is severely distorted. Due to the lack of retransmission requests or forward error correction techniques, the physical layer will never achieve an error rate of 0 % on a real (error-prone) system which underlines the necessity of the data-link layer.

5.3 Data-Link Layer

In Section 5.2, we evaluated the physical layer and concluded that further measures are necessary to build high-performance TLB covert channels. We already anticipated these results and proposed a data-link layer in Section 4.2. Next, we evaluate to what extent the stronger error detection capabilities and forward error correction increase the channel performance. We continue using the same metrics of bit rate (speed) and error rate (quality).

In the left plots of Figure 5.9, we varied the amount of Reed-Solomon parity bytes during the nginx scenario for an accessed bit-based channel. The physical layer is configured with an eviction count of 8 and CRC-8 as checksum. We notice a significant drop in the bit rate when enabling the data-link layer, especially for shorter sender windows. For a sender window of 120, the bit rate reduces to approximately a quarter at an error rate being twice as high compared to using the physical layer only. For longer sender windows, the error rate of the physical layer stays constant at around 30 % as determined by the checksum's detection capabilities. In contrast, depending on the parity amount, we are able to gradually lower the error rate with our two-layer approach. The gradient of the reduction is proportional to the amount of parity. With a parity of 96 and a

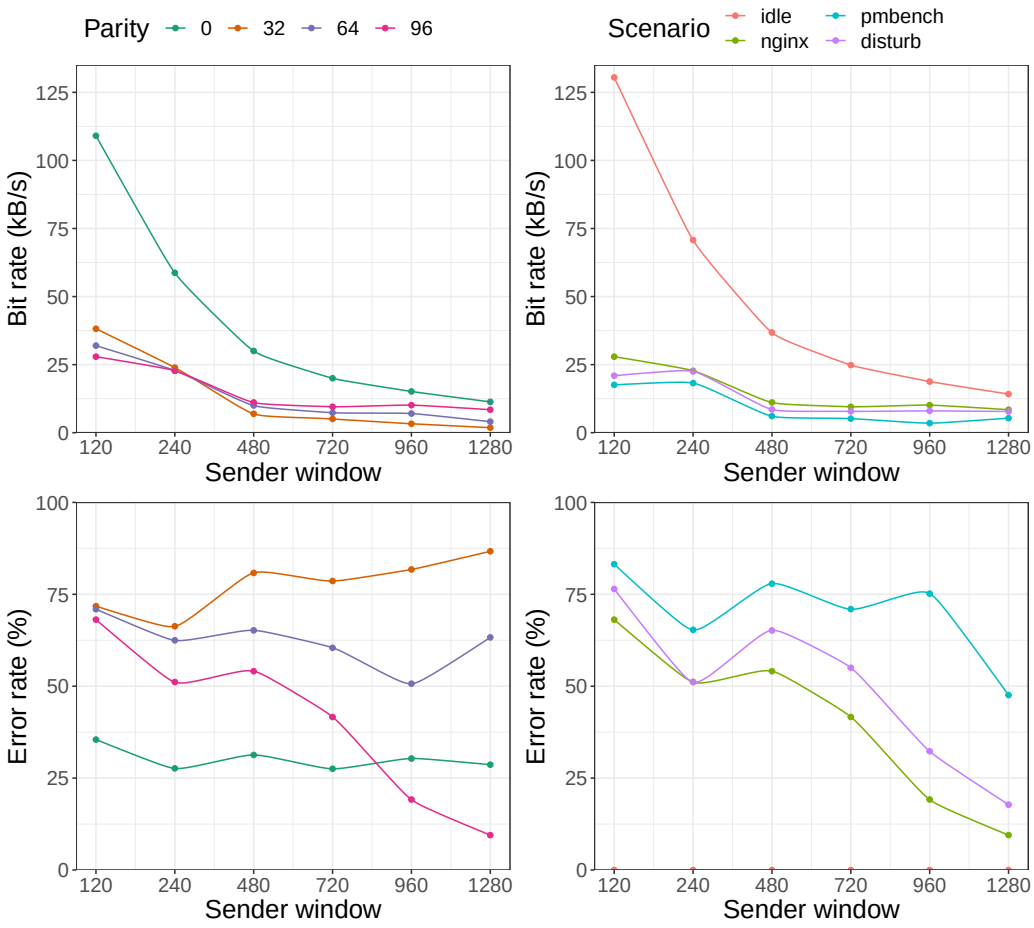


Figure 5.9: The left plots show the effect of the sender window and amount of parity on the bit rate and error rate during the nginx scenario using accessed bits. The right plots show the channel with 96 bytes parity for different scenarios. An increase of the sender window reduces the error rate until too many corrupt packets are wrongly accepted by the physical layer. Depending on the level of interference and amount of parity, we might not be able to achieve a lower error rate than without the data-link layer.

sender window of 1280, we achieve a minimum error rate of 9.5 % at a bit rate of 8.5 kB/s. The minimum error rate for a parity of 64 is 50.7 % with a sender window of 960, respectively 66.3 % for a parity of 32 and a sender window of 240. However, for a parity of 32 or 64, the error rate starts rising again when choosing a sender window beyond these minima. As a result, even though a parity of 96 provides most redundancy and therefore theoretically offers the lowest bit rate, the effective bit rate is higher than for a parity of 32 or 64 due to less errors.

The high increase in the error rate when using the data-link layer is caused by distributing Reed-Solomon blocks over 255 physical packets, thereby transmitting multiple blocks in parallel (see Section 4.2). While the spreading makes the channel robust for packet errors within the Reed-Solomon's correction capabilities, the downside of this approach is that when exceeding this limit, we immediately get multiple corrupted blocks. For a packet payload size of 30 bytes and 64 parity bytes per block (i.e., 191 data bytes), we transmit 30 blocks in parallel, leaving us with 5730 corrupt data bytes when loosing more than 64 packets every 255 packets on the physical layer (or in the case of wrongly accepting more than 32 corrupt packets). When increasing the sender window from 120 to 240, we loose less packets and are able to correct more errors. For these shorter sender windows, the allowed number of erasures is the relevant factor which determines how many packets we may loose or discard intentionally (e.g., due to a checksum mismatch). Regarding erasures, we reach the optimum at a similar sender window independent of the amount of parity (240 in the nginx scenario). The longer the sender window, the more corrupt packets pass the checksum undetected, eventually making Reed-Solomon error correction impossible. In this case, the tolerated number of errors (at random positions) is the limiting factor and therefore higher parity amounts allow longer sender windows.

The right plots of Figure 5.9 show the channel with 96 bytes parity for different scenarios. In idle state, the error rate is 0 % regardless of the sender window at a maximum bit rate of 130.5 kB/s for a sender window of 120. The physical layer by itself (using the same configuration) achieves a bit rate of 228.6 kB/s at an error rate of 7.6 % for a sender window of 120. While in idle state, the bit rate gets less degraded compared to the benchmark scenarios, we still incur a considerable overhead through the Reed-Solomon code. The level of interference introduced by the various benchmark scenarios affects the overall effectiveness of the data-link layer. When increasing the sender window from 120 to 1280, we see a total reduction of 86.1 % in the error rate for the nginx benchmark, a reduction of 76.8 % for disturb, and 42.8 % in case of the pmbench scenario.

We compare the checksums when using the data-link layer in combination with accessed bits for idle state and the nginx scenario in Figure 5.10. For lower amounts of redundancy (short sender windows, low parity amount), we get a slightly better performance using the custom XOR checksum. As soon as we introduce more redundancy through longer sender windows or more parity, Berger and CRC-8 yield a similar or even better error rate. Nonetheless, the results show wide spreads in the error rate, especially when interference is present. Between Berger and CRC-8, the only significant difference can be seen for a sender window of 100 and a parity amount of 64 in idle state. In this case, Berger achieves about half the error rate compared to CRC-8. However, if we introduce noise or increase the amount of redundancy, both codes yield similar bit and error rates.

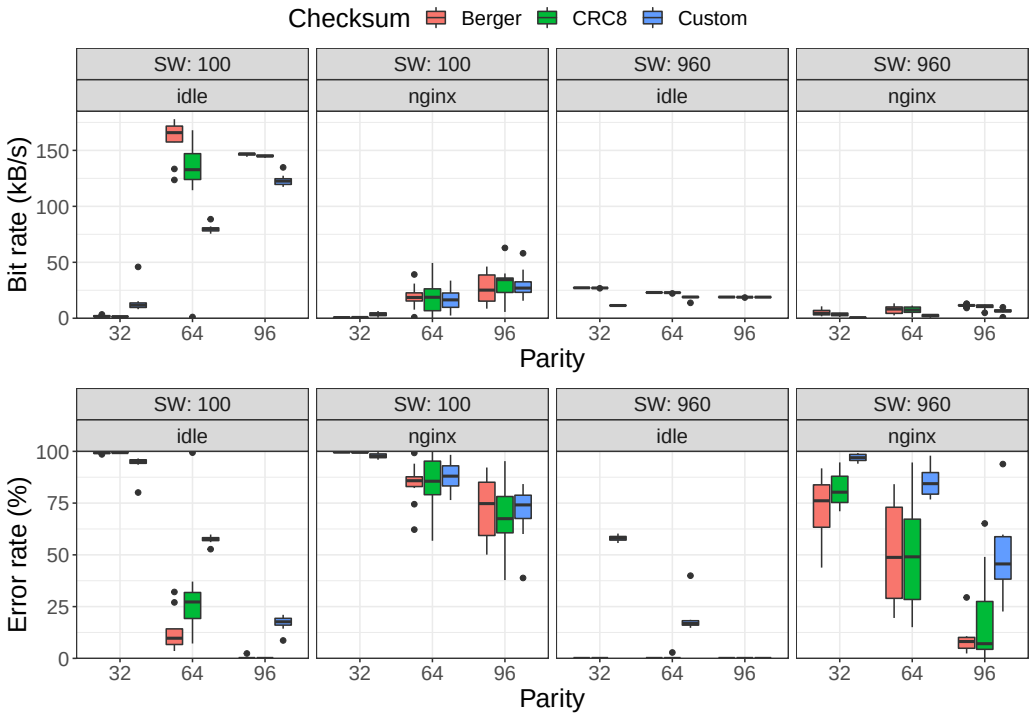


Figure 5.10: Comparison of the checksums for different parity amounts and sender windows (SW) in idle state and during the nginx scenario using accessed bits. For higher amounts of parity (long sender windows, low parity amount), the Berger code and CRC-8 yield approximately equal results, but outperform the custom XOR checksum due to the custom XOR's weak error detection capabilities.

We repeat the same comparison for timestamps using appropriate sender windows with an eviction count of 13 in Figure 5.11. For a sender window of 10, the custom XOR checksum is significantly better than CRC-8 or Berger code. We achieve a maximum bit rate of up to 197.8 kB/s at an error rate of 0.9 % with a parity amount of 64. In contrast, except for the Berger code in combination with a parity amount of 96, using CRC-8 or Berger code results in no (correct) packets being received at all. For a sender window of 100, the CRC-8 yields the lowest error rates, especially when interference is present. In idle state, the error rate stays below 1 %. During the nginx scenario, we encounter a wide spread in the error rate, ranging from 2.9 % to 100 % in the case of a parity amount of 96. The Berger code performs worst for a sender window of 100, requiring a higher parity amount to keep up with the other checksums in idle state and barely receiving correct packets during the nginx scenario.

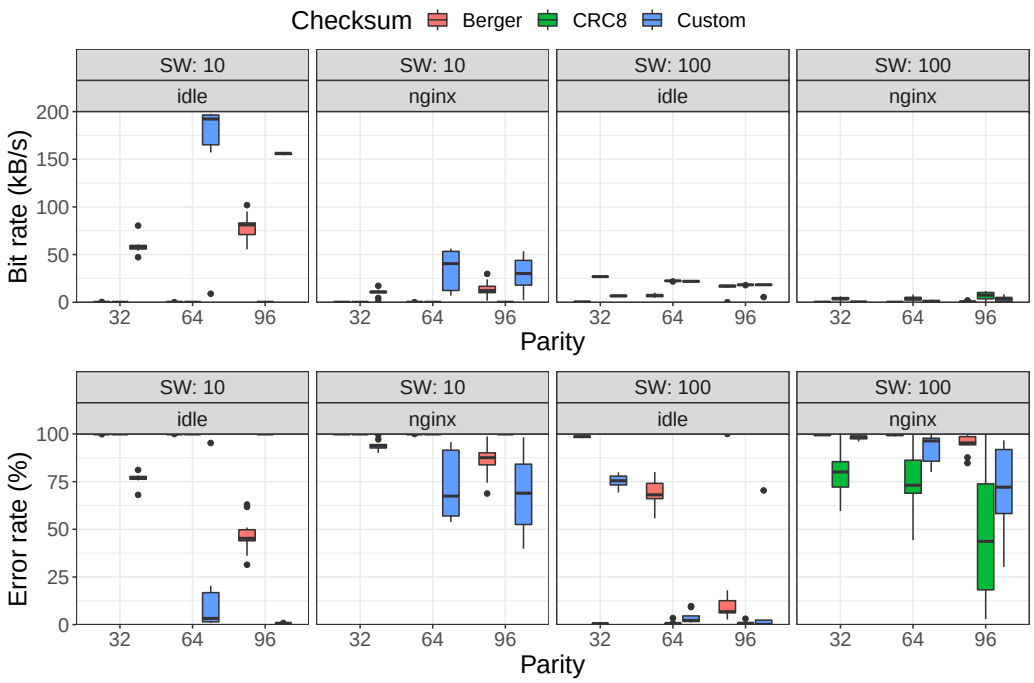


Figure 5.11: Comparison of the checksums for different parity amounts and sender windows (SW) in idle state and during the nginx scenario using timestamps. For short sender windows, the amount of total (and therefore also corrupt) packet repetitions is lower, making a weaker checksum the better choice due to its faster computation.

We draw the following conclusion regarding the interplay between the checksum strength and the Reed-Solomon capabilities: The larger the sender window, the more corrupt packets are wrongly accepted by the checksum, especially if many packets are corrupted due to interference. If we choose a large amount of parity, we are able to tolerate more of these corrupt packets on the data-link layer. However, a larger amount of parity requires more packets to be sent in total, which in turn increases the risk of receiving more corrupted packets, leading to the wide spread we noticed with regard to the error rate. Alternatively, we can lower the amount of wrongly accepted packets by choosing a stronger checksum. On the physical layer, we already saw that CRC-8 provides the best protection against errors, followed by the Berger code, with the custom XOR checksum being the weakest error detection code. On the other side, for short sender windows, the amount of total (and therefore also corrupt) packet repetitions is generally lower, making a weaker checksum the better choice due to its faster computation. The remaining errors (lost or corrupt packets) are handled by the data-link layer in accordance with the parity amount.

Based on the previous considerations, we derive the following four configurations as candidates for high-performance TLB channels in Table 5.1. The "-idle" configurations are meant for idle environments, whereas the "-ix" configurations tolerate a certain amount of interference. We evaluate these candidates not only by simultaneously running a benchmark on a third (neutral) VM, but also while running a benchmark on the sender or receiver VM (see Figure 5.12). Running a benchmark on the receiver VM in parallel to the covert channel never yielded an error rate below 99 % regardless of the configuration and therefore, we omitted these results. The reason for this is that the receiver process has to share its processing time with the benchmark and regularly gets descheduled, incurring a much higher packet loss than we are able to correct in the data-link layer. Consequently, the attacker has to ensure that no unnecessary processes run concurrently on his receiver VM. The same effect does not happen when running a benchmark on the third VM because it is pinned to the core rather than to a single hyperthread (see Figure 5.1) and therefore only interrupts the receiver VM when being scheduled on the same hyperthread.

Parameter	AB-idle	TSC-idle	AB-ix	TSC-ix
TLB Monitoring	accessed bits	timestamps	accessed bits	timestamps
Sender Window	120	10	960	100
Eviction Count	8	13	8	13
RDTSC Threshold	–	74	–	74
RDTSC Window	–	2	–	2
Checksum	Berger	Custom	CRC-8	CRC-8
RS Parity	32	96	96	96

Table 5.1: Candidates for high-performance TLB channels.

In idle state, we get best results using the idle configurations. AB-idle achieves about 200 kB/s at an error rate of 0 % and TSC-idle a bit rate of 155 kB/s at an error rate of less than 1 %. The configurations meant for interference can be used in idle state as well, but offer much lower bit rates (around 20 kB/s) without further decreasing the error rate. Of course, as soon as interference is present, the idle configurations lack the amount of redundancy required to deal with the errors. Although the interference configurations achieve an error rate of 0 % in some cases, we cannot guarantee an upper limit. Especially when running a benchmark on the sender VM, we notice a wide spread regarding the error rate. At first, descheduling the sender process is uncritical since we do not continue receiving until the sequence number and checksum are correct. To a limited extent, even idle configurations are able to handle this kind of interference as seen at AB-idle during the nginx scenario. However, TLB activity of the benchmark

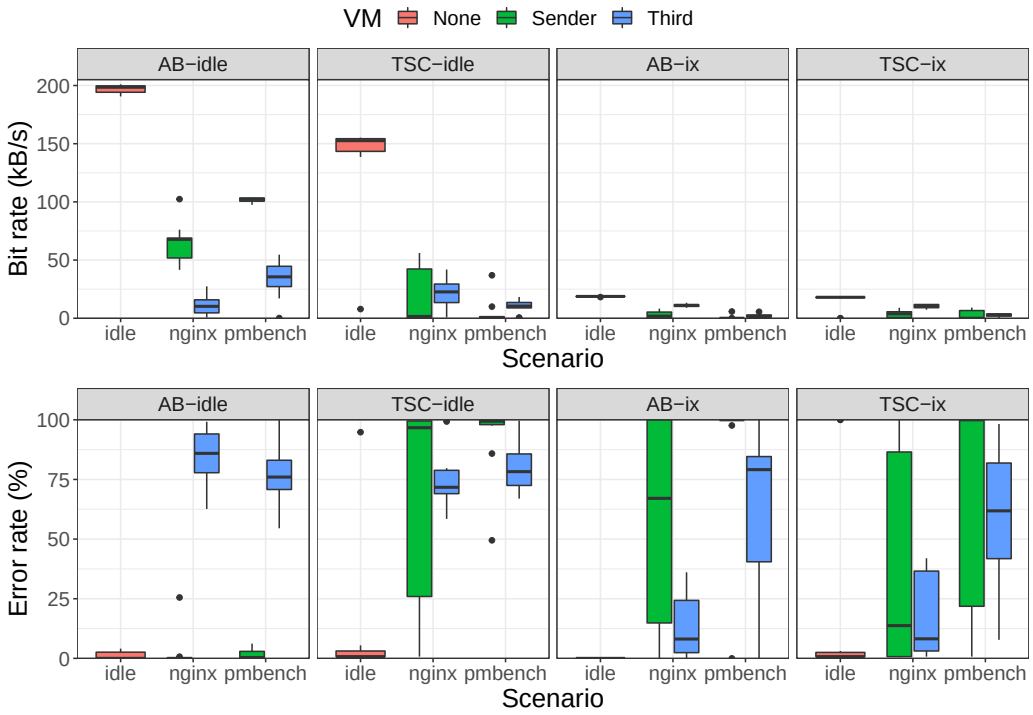


Figure 5.12: Performance of the high-performance TLB channel candidates in idle state and while running a benchmark on the sender VM or a third (neutral) VM. The idle configurations perform better in idle state, whereas the interference configurations are better suited for the benchmark scenarios. Running the benchmark on the sender VM is similar to idle state with regular send interruptions.

might accidentally create "valid" packets and for the receiver and sender VM are pinned to adjacent hyperthreads, there is no chance of loosing these packets as it would have been the case if the third VM gets scheduled in place of the receiver VM. We see this in an increased error rate when running the pmbench on the sender VM rather than on the third VM. A possible solution to this is using a stronger checksum to validate packets (e.g., CRC-32).

In Figure 5.13, we transfer different types of data in idle state and under interference. The plain text file is the smallest file with a size of 202 kB, followed by the MP3 audio file (1.1 MB), and the bitmap image file being the largest (3.2 MB). Bigger files have a higher transfer duration, thereby being longer exposed to errors. For this reason, the higher amount of parity in TSC-idle compared to AB-idle is required for an error-free transmission of the bitmap file. The MP3 file represents an atypical case in this plot. Especially for timestamp-based channels, the error rate is significantly higher (and widely spread) than for the

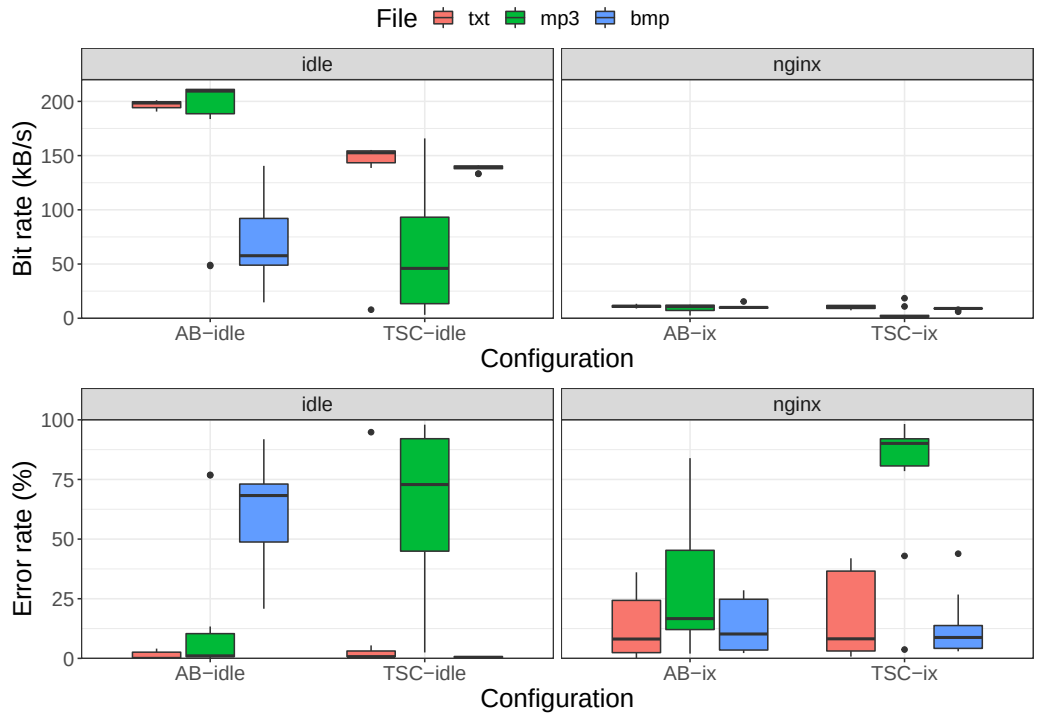
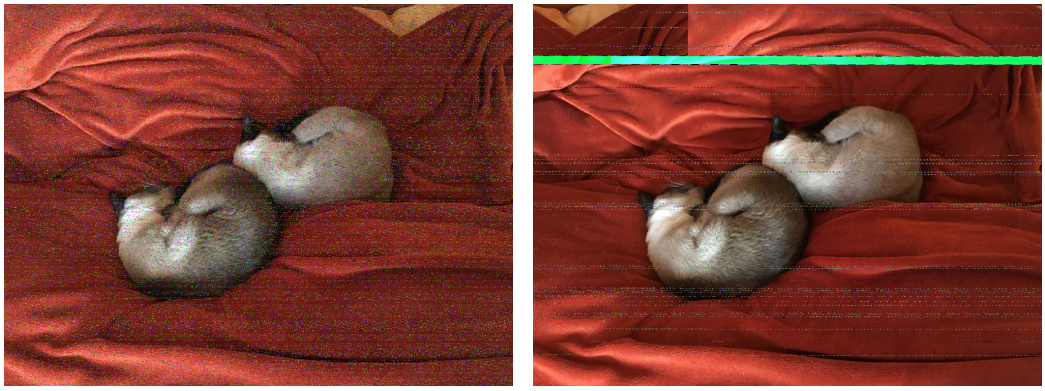


Figure 5.13: Sending different files in idle state and under interference.



(a) AB-idle (idle), error rate: 68.4 %. (b) AB-ix (nginx), error rate: 4.3 %.

Figure 5.14: Output image files with data-link layer enabled. Errors mainly occur as noise, whereas the overall structure is largely preserved.

other two files. We notice that the binary representation of the MP3 file contains regular blocks of zeros, leading to quick send iterations in those file segments because no evictions have to be made. We suspect that either the shorter send iterations or the checksum calculated over a payload of zeros cause this behavior.

Nonetheless, when listening to the MP3 file, we were able to identify the original composition even at an error rate as high as 90 %. The audio contains artificial glitches and tempo deviation, but the data-link layer helps preserving the overall file structure. We illustrate this in Figure 5.14 for the bitmap file. Whereas an error rate of 0.8 % already led to a heavily distorted image when using the physical layer only (see Figure 5.8), the Reed-Solomon code considerable improves the results, even when interference is present.

Though the STLB of our evaluation platform is organized in 256 sets, allowing us to send packets of 32 bytes size (30 bytes payload), other processors come with different TLB organizations. For instance, the STLB of Intel Skylake processors is organized in 128 sets which reduces the maximum packet size to 16 bytes (14 bytes payload). Furthermore, we might arbitrarily choose a smaller packet size to act more like a permissible program in order to circumvent monitoring tools. We compare different packet sizes for the idle configurations in Figure 5.15. There is a linear correlation between the packet size and maximum bit rate. Therefore, TLBs with fewer sets are disadvantageous from an attacker's perspective. However, in case of the Intel Skylake or Ice Lake platform, the actual amount of total TLB entries has not been reduced, but rather the sets were

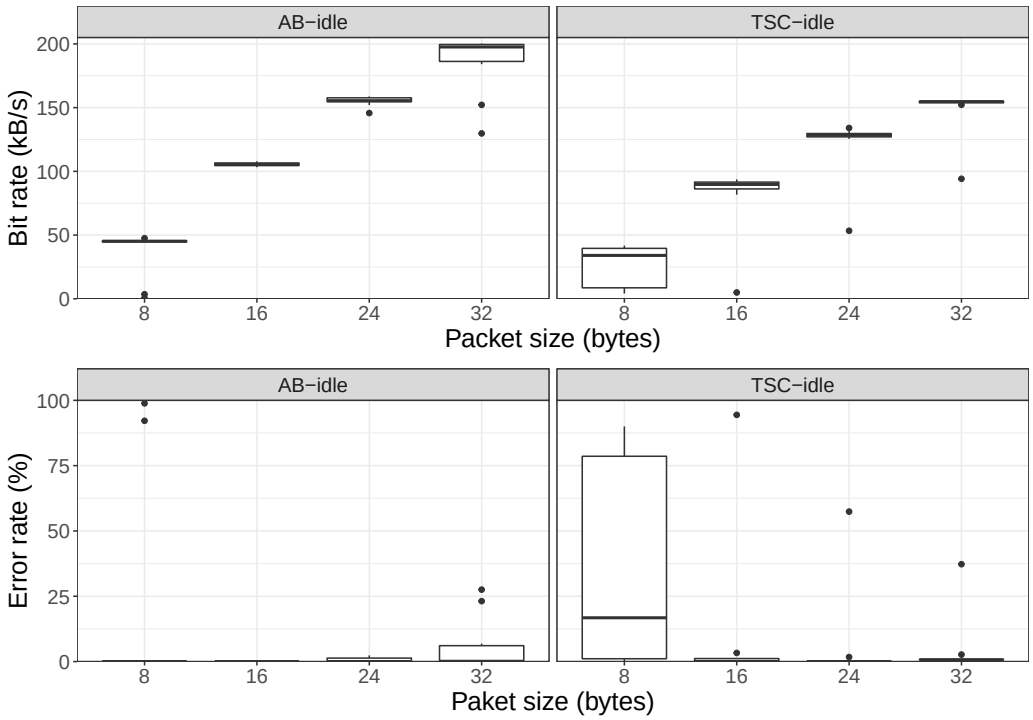


Figure 5.15: Effect of the packet size on the bit and error rate in idle state. The bit rate is linearly correlated to the packet size.

equipped with more ways. If it is possible to gain fine-grained control over the entries in each set, we could send multiple bits per set, possibly achieving similar or even higher bit rates. This has to be studied in future research. We observe that a packet size of 8 bytes yields unreliable results using TSC-idle. Since smaller packets mean that less bits have to be written to the TLB sets in each iteration, thereby shrinking the sender window, this might indicate that we send too fast to uphold the sampling theorem.

5.4 Discussion

In this chapter, we evaluated how the parameters of our two-level protocol affect the transmission speed and quality of the covert channel. Most importantly, the question rises whether the data-link layer is worth its performance reduction. If speed is more important than keeping the file structure, using the physical layer by itself is a viable choice since we achieve bit rates of around 250 kB/s under optimal conditions. However, lost packets cannot be recovered and this is critical for some kinds of data such as images. Even in idle state, minor errors make error rates of 0 % extremely unlikely without any form of forward error correction. Therefore, we drew the idea of introducing a second layer from the covert channel design proposed by Maurice et al. [41]. They use Reed-Solomon to correct minor errors for their physical layer transmits data with a fairly low error, but still is not error-free. This is comparable to our situation in idle state. When interference is present, they handle corrupt packets by issuing retransmission requests which is unfeasible in an unidirectional channel setting. In our case, the errors propagated by the physical layer quickly exceed the capabilities of the Reed-Solomon code, making guarantees for a certain error level impossible. Nonetheless, the data-link layer provides much stronger error detection capabilities compared to the checksum on the physical layer. If a data block is regarded as correct by the Reed-Solomon code, we have a high certainty that this data is equal to the original. In our design, we decided to keep corrupt fragments as well since correct information might be found between corrupt bytes. The main advantage of using the data-link layer is that the file structure is preserved to a large extent, even in noisy environments. Due to the Reed-Solomon code's block structure and our way of distributing blocks over many packets, errors mainly occur as noise which is tolerable for some kinds of data such as images.

Furthermore, it has to considered whether accessed bits or timestamps are used to monitor TLB entries. Without the data-link layer, the noise introduced by measurement inaccuracies on microarchitectural level cause channels based on timestamps to be less reliable. This drawback diminishes with additional error correction via the data-link layer, but still, error rates tend to be higher if the

bit rates ought to be in the range of accessed bit-based channels. Nonetheless, timestamp-based monitoring allows bidirectional channels since they do not dependent on manipulating the page table entries. This way, we could implement retransmission requests similar to Maurice et al. [41] and significantly improve robustness under interference. For each received packet, the receiver writes an acknowledgment in a reserved subset of the TLB sets (ACK packet). Manually choosing a sender window becomes no longer necessary as the sender only continues with the next packet after receiving an acknowledgment. In combination with the Reed-Solomon code, this should guarantee an error-free transmission. Ideally, we choose a hybrid approach leveraging accessed bit-based monitoring on the receiver side (more reliable, controlled by us) and timestamp-based monitoring on the sender side (lower requirements regarding reliability).

Whether or not we achieved our goal of building a high-performance TLB-based covert channel depends on the application scenario. In contrast to the proof-of-concept TLB covert channel of Gras et al. [18], we significantly increased the bit rate and, in idle state, outperform most of the covert channels listed in Table 2.2. Even though the cache-based covert channels of Percival [45] or Gruss et al. [19] allow higher bit rates, they were not designed to send data across VMs and can be mitigated with state-of-the-art cache isolation techniques such as Intel CAT or TSX. On the other hand, when concurrently running processes cause interference, the weaknesses of our unidirectional approach become visible. While in some cases, almost no errors are contained in the received data, we cannot speak of a high-performance channel in general.

Chapter 6

Conclusion

The main objective of this thesis was to study whether high-performance covert channels in terms of channel bit rate and reliability are possible over the TLB. Therefore, we designed a two-layer protocol to deal with the challenges of synchronization and interference caused by concurrently running processes. We presented a novel TLB monitoring method using accessed bits and in combination with the error detection and correction techniques in the two layers propose several candidates for fast and robust channels. The prerequisites to establish a channel across VMs are that the VMs reside on the same core (or adjacent hyperthreads) and VPID tagging is enabled.

Our evaluation has shown that in idle state, the criteria of high-performance covert channels is met for we achieve a bit rate of up to 200 kB/s at an error rate of 0 %. However, as we increase the level of interference, error-free transmissions are less likely and the bit rate drops under 20 kB/s. The unidirectional design we chose does not allow retransmission requests which might solve this problem. Even though this disqualifies our candidates from being high-performance covert channels in noisy environments, we still are able to exfiltrate data in presence of state-of-the-art cache isolation techniques. As little as 1 kB of leaked passwords is enough to cause immense damage and therefore, we urge vendors to carefully consider all (implicitly) shared resources when designing isolation techniques.

The source code for the covert channels described in this thesis is available at https://github.com/deermichel/tlbchannels.

6.1 Future Work

In Section 3.1, we briefly mentioned that achieving core co-residency can be further simplified by using VMs with multiple cores (vCPUs). The higher the number of vCPUs, the higher is the chance of sharing a core with the victim

VM. This is crucial if the victim VM also has multiple vCPUs for we might not be able to pin the sender process to a specific vCPU. The protocol needs to address regular core changes, and, if we want to extent this scenario to allow multiple sender VMs at the same time, requires support for sender identification.

At the end of Section 5.3, we saw that using our current design, TLBs with fewer sets are disadvantageous from an attacker's perspective for the bit rate correlates linearly with the number of sets. However, TLBs with less sets tend to have more ways per set. If we are able to reliably evict a specific number of entries in each set and monitor this number on the receiver side, we could encode multiple states per set, possibly increasing the bit rate or providing more robustness.

The key drawback of our design was the lack of retransmission requests due to the unidirectional requirement posed by the accessed bit-based TLB monitoring technique. However, timestamp-based monitoring is feasible on the sender side as well and should be sufficiently reliable for receiving acknowledgments by the receiver. Ideally, we pair both monitoring techniques in a hybrid approach to benefit from the robust and fast transmission using accessed bits on the receiver side. In combination with the Reed-Solomon code, this should guarantee an error-free transmission and forms the foundation for high-performance TLB covert channels under noisy conditions.

While we focused on Intel processors in this thesis, the covert channel should technically work on AMD processors as well. For instance, the Zen 2 architecture employs a two-level TLB hierarchy, of which the L2 data TLB is organized in 128 sets with 16 ways each [13]. The Address Space ID (ASID) represents the equivalent to the VPID on Intel platforms and can be used to tag TLB entries on VM level [5].

Rather than implementing the sender program in a low-level language such as C, we thought about the possibility to write the sender using JavaScript. Being able to mount a successful attack from within a browser would greatly lower the bar for an adversary, though without direct memory access, the channel is more complicated to implement. Specific pages can be accessed by allocating a large consecutive raw binary buffer (ArrayBuffer) and accessing entries with a 4 KiB stride. Since the base address of the array cannot be manually specified, the receiver has to somehow detect the TLB set representing the first bit of each packet.

Appendix

A.1 Binary Data Analysis

From a statistical perspective, the number of one and zero bits in a sufficiently large amount of binary data is about the same, making up around 50 % of the total bits each. We study this distribution on a commodity macOS system with 123.8 GB of data. For this real-world data, we observe that the total number of binary 1's is slightly lower (44.9 %) than the number of zero bits (55.1 %). We list the results for the most common file extensions in Figure A.1.

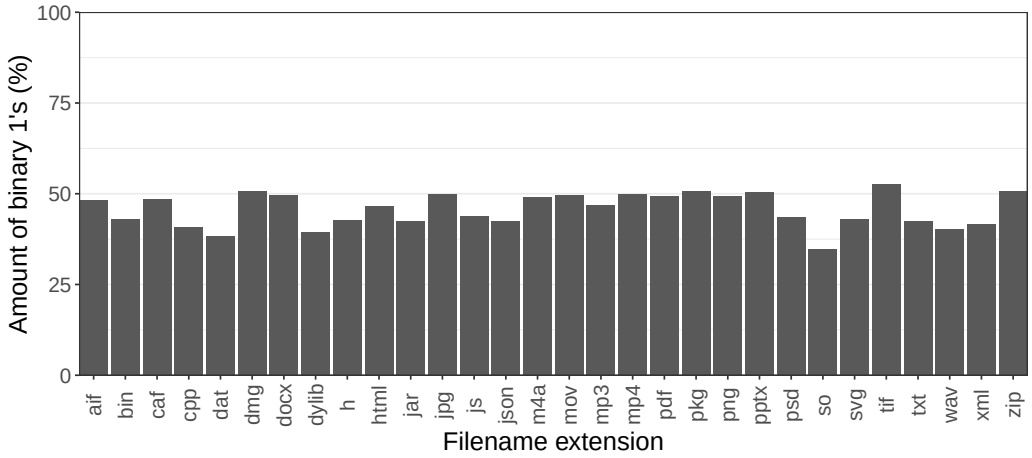


Figure A.1: Amount of binary 1's by file extension. While data with predominantly textual content tends to contain more zero than one bits, non-textual data, such as multimedia data or disk images, tends towards an equal distribution.

Bibliography

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In *International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45, 2002.

[2] Amazon. Amazon EC2 Instance Types. `https://aws.amazon.com/ec2/instance-types/?nc1=h_ls`, 2020. [Last Accessed: July 30th, 2020].

[3] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. Side Channels in the Cloud: Isolation Challenges, Attacks, and Countermeasures. 2017.

[4] Jay M Berger. A note on error detection codes for asymmetric channels. *Information and Control*, pages 68–73, 1961.

[5] Sebastian Biemueller. ASID Management in Xen AMD-V. In *Xen Summit Spring 2007*, 2007.

[6] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2016-2021. White Paper. 2018.

[7] Intel Corporation. TLBs, Paging-Structure Caches, and Their Invalidation. 2008.

[8] Intel Corporation. Timestamp-Counter Scaling for Virtualization. 2015.

[9] Intel Corporation. Intel Xeon Processor E5-2630 v4 Product Specification. `https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e5-2630-v4-25m-cache-2-20-ghz.html`, 2016. [Last Accessed: June 30th, 2020].

[10] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2019.

[11] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. 2019.

[12] Intel Corporation. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2020. [Last Accessed: June 30th, 2020].

[13] Advanced Micro Devices. Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors. 2020.

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[15] Kazuhisa Fujimoto. Checksum and Cyclic Redundancy Check Mechanism. In *Encyclopedia of Database Systems*. Springer, 2016.

[16] Gideon Gerzon. Intel Virtualization Technology Processor Virtualization Extensions and Intel Trusted execution Technology, 2007.

[17] Google. Compute Engine, Documentation, CPU platforms. https://cloud.google.com/compute/docs/cpu-platforms#intel_cpu_processors, 2020. [Last Accessed: July 30th, 2020].

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

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

[20] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, 2017.

[21] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[22] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From concept to reality in

the Intel® Xeon® processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[23] Dirk W Hoffmann. *Einführung in die Informations- und Codierungstheorie*. Springer, 2014.

[24] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, page 898, 2015.

[25] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 353–364, 2016.

[26] Manuel Kalmbach. *Frequenz-basierter Covert Channel mithilfe der Intel Turbo Boost Technology*. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2019.

[27] Phil Karn. DSP and FEC Library. `http://www.ka9q.net/code/fec/`, 2007. [Last Accessed: July 30th, 2020].

[28] Linux kernel community. Linux Kernel-based Virtual Machine. `https://www.linux-kvm.org/`, 2020. [Last Accessed: July 31st, 2020].

[29] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, pages 5–27, 2011.

[30] Butler W Lampson. A Note on the Confinement Problem. *Communications of the ACM*, pages 613–615, 1973.

[31] Michael Larabel and Matthew Tippett. Phoronix Test Suite. 2020. URL `https://www.phoronix-test-suite.com/`. [Last Accessed: April 1st, 2020].

[32] Donald C Latham. Department of Defense Trusted Computer System Evaluation Criteria. *United States Department of Defense*, 1986.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[35] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing . In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[36] J-C Lo, Suchai Thanawastien, TRN Rao, and Michael Nicolaidis. An SFS Berger check prediction ALU and its application to self-checking processor designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 525–540, 1992.

[37] Canoncial Ltd. Ubuntu. `https://ubuntu.com/`, 2020. [Last Accessed: July 31st, 2020].

[38] Martin Luther. Die Bibel: 1. Buch Mose. *Projekt Gutenberg-DE*, 1912. URL `https://www.projekt-gutenberg.org/luther/bibel/bibel.html`. [Last Accessed: July 31st, 2020].

[39] Andy Lutomirski and Ingo Molnar. x86/mm: Implement PCID based optimization: try to preserve old TLB entries using PCID. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=10af6235e0d327d42e1bad974385197817923dc1`, 2017. [Last Accessed: April 2nd, 2020].

[40] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 2002.

[41] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, pages 8–11, 2017.

[42] Phil Muncaster. Data Leak Exposes Thousands of US Defense Contractor Staff. *Infosecurity Magazine*, 2019. URL `https://www.infosecurity-magazine.com/news/cloud-data-leak-thousands/`. [Last Accessed: June 27th, 2020].

[43] Keisuke Okamura and Yoshihiro Oyama. Load-Based Covert Channels between Xen Virtual Machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 173–180, 2010.

[44] Dan O'Sullivan. Cloud Leak: How A Verizon Partner Exposed Millions of Customer Accounts. *UpGuard Blog*, 2017. URL `https://www.upguard.com/breaches/verizon-cloud-leak`. [Last Accessed: July 20th, 2020].

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

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, 2016.

[47] William Wesley Peterson and Daniel T Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, pages 228–235, 1961.

[48] Irving S Reed and Gustave Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, pages 300–304, 1960.

[49] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, 2009.

[50] Tim Schmidt. *Covert Channel based on AMD Precision Boost 2*. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2019.

[51] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267, 2017.

[52] Sensirion. Application Note: CRC Checksum Calculation. `https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/2_Humidity_Sensors/Application_Note/Sensirion_Humidity_Sensors_SHT1x_SHT7x_CRC_Calculation.pdf`, 2011. [Last Accessed: July 30th, 2020].

[53] Claude Elwood Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, pages 10–21, 1949.

[54] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, 2001.

[55] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Pearson, 10th edition, 2016.

[56] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4th edition, 2015.

[57] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. `https://cacheoutattack.com/`, 2020. [Last Accessed: April 1st, 2020].

[58] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, 2015.

[59] Stephen B Wicker and Vijay K Bhargava. An Introduction to Reed–Solomon Codes. *Reed-Solomon Codes and Their Applications*, pages 1–16, 1994.

[60] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud. *IEEE/ACM Transactions on Networking*, pages 603–615, 2014.

[61] Sheng Yang. Extending KVM with new Intel Virtualization technology. In *KVM Forum*, 2008.

[62] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.