Lukas Giner, BSc

A Robust High-Speed Cache Covert Channel in the Cloud

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Asst. Prof. Daniel Gruss

Institute for Applied Information Processing and Communications

Advisor

Dr. Clémentine Maurice

Graz, January 2020

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

In nearly all modern CPU architectures, caches are used to bridge the speed gap between the processor and the much slower main memory. Over the last years, these caches have been shown time and again to be prime candidates for side-channel attacks. One type of cache attack is the construction of a covert channel to exfiltrate data from systems that allow no overt communication. A specific scenario is the extraction of data from one virtual machine to another, attacker-controlled VM in a cloud computing environment. Prior work has demonstrated that this is indeed possible in principle, but, so far, a high-speed channel free of transmission errors and with minimal restrictions for the host system has not been published.

In this thesis, we present our work towards achieving this. First, we define a threat model that is broad enough to encompass a large variety of system configurations by stripping away as many requirements as possible. Next, we systematically identify the challenges implied by this threat model. The main challenges are finding addresses that are congruent in the cache, establishing a connection from one virtual machine to another, and keeping the channel synchronized. To identify congruent addresses, we expand upon previous work and improve their methods. With the *Jamming Agreement*, we present a novel approach to establishing contact and transmitting the initial channel parameters. For synchronization, we develop a two-way communication system with a back-channel, which can keep the channel synchronized, even in the presence of deschedulings or a non-monotonic system clock. Finally, we conduct an extensive analysis of the properties of this channel and conclude that it does indeed fulfill the premise of a robust, high-speed cache covert channel in the cloud.

Kurzfassung

In fast allen modernen CPU-Architekturen werden Caches verwendet, um die Geschwindigkeitslücke zwischen dem Prozessor und dem viel langsameren Hauptspeicher zu überbrücken. In den letzten Jahren hat sich immer wieder gezeigt, dass diese Caches sehr gute Kandidaten für Seitenkanalangriffe sind. Eine Art von Cache-Angriff ist der Aufbau eines verdeckten Kanals, um Daten aus Systemen herausschleusen, die keine offene Kommunikation zulassen. Ein konkretes Szenario ist die Extraktion von Daten von einer virtuellen Maschine in der Cloud zu einer anderen, welche unter der Kontrolle des Angreifers steht. Frühere Arbeiten haben gezeigt, dass dies zwar prinzipiell möglich ist, bisher wurde aber noch kein Kanal mit hoher Übertragungsgeschwindigkeit veröffentlicht, der frei von Übertragungsfehlern ist und zudem minimalen Einschränkungen für das Host-System hat.

In dieser Arbeit stellen wir unsere Anstrengungen in diese Richtung vor. Zunächst definieren wir ein Bedrohungsmodell, das umfassend genug ist, um eine große Vielfalt von Systemkonfigurationen abzudecken, indem wir so viele Anforderungen wie möglich entfernen. Als nächstes identifizieren wir systematisch die Herausforderungen, die dieses Bedrohungsmodell mit sich bringt. Die Hauptherausforderungen bestehen darin, kongruente Adressen im Cache zu finden, eine Verbindung von einer virtuellen Maschine zu einer anderen aufzubauen und den Kanal synchron zu halten. Um kongruente Adressen zu finden, erweitern wir die bisherigen Arbeiten und verbessern deren Methoden. Mit dem *Jamming Agreement* stellen wir einen neuartigen Ansatz zur Kontaktaufnahme und Übertragung der anfänglichen Kanalparameter vor. Für die Synchronisation entwickeln wir ein Zweiwege-Kommunikationssystem mit einem Rückkanal, welches den Kanal auch in Anwesenheit von Prozess-Descheduling oder einer nicht monotonen Systemuhr synchron halten kann. Zu guter Letzt führen wir eine umfassende Analyse der Eigenschaften dieses Kanals durch, und kommen zu dem Schluss, dass er tatsächlich unsere Ziele eines robusten, verdeckten Hochgeschwindigkeits-Cache-Kanals in der Cloud erfüllt.

Acknowledgements

Firstly, I want to thank my advisors Daniel and Clémentine, for asking me to be a part of this project in the first place, and then guiding me through the process. I want to thank Clémentine in particular for taking the time to read many rough drafts, even long after leaving the university, and Daniel for pushing me to complete the rest of my master's.

To my parents, I want to express my sincere gratitude for tolerating my dawdling for all these years and supporting me even when progress was sparse - I know you're quite relieved to see the end of this road.

Lastly, I want to thank Michael Schwarz and Manuel Weber for their part in this work, and all the help and advice they provided.

Contents

Chapter 1

Introduction

CPU caches are an integral part of modern computing and constitute an important area of security research. As they are present on nearly all of today's electronic devices, attacks and exploits based on their architecture can affect many different types of devices, independent of their operating systems.

Cloud computing presents a large group of such affected devices, as it is possible for multiple tenants to be co-located on the same physical CPU [1, 2, 3, 4, 5, 6, 7]. As an ever-increasing number of services handling highly sensitive data populate the cloud, cache attacks are an increasingly interesting attack vector because they break isolation between tenants.

Attacks using CPU caches are based on measuring the cache access timings, which puts them in the category of timing side-channel attacks. They have been a field of research for many years but became more active with the widespread availability of fast multi-core machines in the last few years. Fundamentally, they allow the attacker to either spy on unknowing third-party applications by monitoring their cache accesses [8] or to communicate covertly with a collaborating application. Both scenarios are "stealthy" with regards to commonly monitored ways of transferring data (e.g., RAM or hard drives), which is why an attack that establishes communication over the cache is referred to as a "cache covert channel." These channels will be the focus of this thesis.

Cache attacks can be based on different techniques, such as Flush+Reload, Flush+Flush or Prime+Probe, each requiring different conditions and providing different advantages. Flush+Flush has been shown [9] to be both stealthy and extremely fast, but requires shared memory. As Prime+Probe operates not on single cache lines, but on entire cache sets, it does not need shared memory, at the cost of speed and accuracy.

When bringing cache covert channels to a cloud environment to establish communication between different tenants, several challenges arise. For one, shared memory is not always available between virtual machines, which necessitates the use of the Prime+Probe technique.

In this thesis, we explore the practical applicability of such a channel and show our steps towards a fast and reliable implementation that works in the virtualized environment of the Amazon Elastic Compute Cloud (EC2). We show the challenges in constructing such a channel from the ground up and present practical solutions for them towards a fully usable channel.

The content of this thesis represents a large contribution to a conference paper presented at NDSS 2017 [10].

1.1 Motivation

Cache covert channels across virtual machines or even in the cloud are not new and have in fact been demonstrated multiple times [1, 11, 12, 13, 14]. However, in our opinion all of these lack practicality in one or more of the following ways:

- speed: transferring data on the order of bytes per second
- requirements: VCPUs sharing the same physical core at some points; a consistent `rdtsc` instruction between the colluding virtual machines; page deduplication
- unresolved synchronization problems: using the edit-distance as a measure of error ratio without addressing the correction of these errors
- a significant impairment in the presence of strong sources of noise on the cache

Our intent is to develop a cache covert channel that could rightly be called both "High-Speed" and "Robust". In doing so, we detail what challenges need to be overcome for robustness (*i.e.*, error correction) in particular to become possible.

1.2 Threat Model

While we test our channel on the Amazon EC2 service, it is not bound by specific features of the EC2. Consequently, the method presented in this thesis should be applicable to any system configuration that fulfills the following few requirements.

R1 Shared LLC

The CPU needs to have a shared and inclusive last-level cache.

R2 Availability of 2MB huge pages

2MB huge pages need to be enabled on the hypervisor and virtual machines.

R3 Exclusive access to two CPU threads per transceiver

For correct operation, sender and receiver need 2 concurrent threads each. These may be located on the same CPU core via hyper-threading.

Our approach does explicitly *not* rely on page deduplication, root access, a monotonic `rdtsc`, `rdtsc` synchronization between virtual machines or identical clock speeds for all CPU cores.

1.3 Challenges

Based on the threat model we created for a widely applicable cache covert channel, several challenges present themselves.

First and foremost, how can we achieve synchronization between sender and receiver without expecting a reliable, cycle-accurate clock (`rdtsc`) for extended periods of time? From this follows the need to tolerate the descheduling of receiver and sender threads (or the entire VM), even if it cannot be detected with the clock. Another obstacle is that virtual to physical address mapping is not available in virtualized environments, but knowing it to some extent is a requirement for Prime+Probe. Finally, there is no preexisting channel to negotiate parameters before communication.

1.4 Outline

Chapter 2 provides background information on CPU caches and different attack types. Chapter 3 discusses the previously mentioned challenges in greater detail. In Chapter 4, we look at specifics for various parts of our implementation and how they serve to overcome the presented challenges. Lastly, we evaluate the performance characteristics of our channel in Chapter 5 and conclude the thesis in Chapter 6.

Chapter 2

Background

In this chapter, we first give a general overview of how caches operate. We will usually default to the behavior in modern Intel processors when specifics are mentioned, as that is the hardware most Amazon EC2 instances offer [15] and the implementation of this thesis was done on. This is followed by a brief overview of errors in digital communication channels as they pertain to this work.

2.1 Caches

In modern CPU architectures, all accesses to the main memory (RAM) go through the cache, a small set of buffers used to store frequently used data. This process speeds up the CPU's operation by significantly reducing the latency for accesses to cached data. From the perspective of running software, this happens transparently.

A cache consists of *cache lines* which are copies of sequential pieces in the main memory, typically 32, 64 or 128 bytes long. When requested data is found in a cache line, this is called a *cache hit*, when it is not, a *cache miss* has occurred. The cache as a whole is usually a combination of several *levels* of caches, the first, L1 being the smallest. The reason for this is a trade-off between speed, hit rate and die-space. For the most frequently accessed data, the lowest latency possible is desirable. Because access time and physical size go up with increased storage size, for maximum performance, the L1 cache is kept small. A smaller size implies a lower hit rate, which is why slower, but generally larger caches are added to increase the overall hit rate. At the time of writing, many end-user CPUs use 3 levels of caches, while a few (such as the Intel i7-5775C) use very large L4 caches for specialized purposes. The highest level of cache is called the *last-level cache* (LLC). In multi-core

processors, higher level caches, such as the LLC, are often shared between cores, while lower level caches are private to each core.

For its i7 series, Intel estimates [16] these latencies:

- L1 cache hit, 4 cycles
- L2 cache hit, 10 cycles
- L3 cache hit $\approx$ 40 cycles
- L3 cache hit, shared line in other core $\approx$ 65 cycles
- L3 cache hit, modified in other core $\approx$ 75 cycles
- DRAM, $>$ 60ns

The exact difference depends on the clock speed of the CPU, but it is clear that even an L3 hit is significantly faster than a cache miss. As the L1 cache is considerably faster than the L3, it follows that deciding which data is cached in the L1 and which is relegated to the slower L2 or even L3 cache is critically important for performance.

The algorithm controlling this behavior is the *cache replacement policy*. This can be a strategy as simple as *first in first out* (FIFO) or something more complex, like *least recently used* (LRU). When data is stored in a cache line, an identifying tag is saved alongside it. The replaced line is *evicted* from this cache level. Recent Intel CPUs have used a pseudo-LRU policy up to and including the Sandy Bridge architecture, but moved to an undocumented variation of LRU with Ivy Bridge, resembling [17] DIP [18] and DRRIP [19].

The cache replacement policy operates on multiple sets of cache lines. How large they are is determined by the addressing scheme of the cache. In a *fully associative* cache, all cache lines belong to the same *cache set* and data can be cached in any line. This means the cache replacement policy operates on the entire cache at once. The upside of this is that the cache size can be optimally used, according to the employed policy. The downside is that for each access, the entire cache has to be searched to determine if a tag is present. Additionally, any change might require updating the metadata of the replacement policy for each cache line. The opposite of this is a *direct mapped* cache, in which each cache line is its own set, which means that each memory address can only be cached in one specific cache line. While finding data in the cache requires the least amount of hardware complexity with this approach, non-uniformly distributed memory access can cause high cache miss rates in some lines and almost no use in others. The so-called *set associative* cache presents a compromise of these two extremes. The cache is partitioned into sets of n cache lines each, called *n-way set associative*. Each memory address now maps to a specific set in the cache, but the replacement policy can decide which of the n ways to replace. This combines

the high-efficiency use of a fully associative cache with the lower hardware requirements of a direct mapped cache. Figure 2.4 shows the process of evicting a cache line from a set in a set associative cache for an LRU policy and the policy used in Ivy Bridge and onwards.

How addresses are tagged and mapped to cache sets depends on the cache's indexing method. Caches can be indexed or tagged from either virtual or physical addresses, which allows 4 different combinations. For the Intel i7 series, the L1 is virtually indexed and physically tagged, L2 and L3 are physically indexed and tagged [20]. Using a part of the virtual address for indexing or tagging is fast, because the incoming virtual address does not have to be translated, but can also cause shared memory to be cached more than once, as the data can have multiple different cache indices and tags. For small caches such as the L1, this problem is mitigated because they need fewer bits for indexing. Since the main memory is segmented into pages of at least 4kB, the lower 12 bits of virtual addresses are the same as in the physical addresses. Using the physical address for indexing and tagging is slower because the address has to first be translated by the *translation lookaside buffer* (TLB). If the address in question is not cached in the TLB, it has to be loaded from memory, which further slows down this operation. The advantage of using the physical address is that there can be no ambiguity.

In addition to the cache replacement policy, the relationship between different caches also defines what data is stored in which cache. Two cache levels can be in one of three relations: *inclusive*, *exclusive* and *non-inclusive*. An inclusive cache has to contain all the data stored in specific (usually all) lower-level caches. This property also applies the other way, which means that an eviction in an inclusive cache necessarily triggers the eviction of that cache line in all caches it includes. LLCs in Intel CPUs are generally inclusive. If two caches are exclusive to each other, they are not allowed to store the same data. AMD used to favor this strategy for its LLCs, but seems to have adopted an inclusive LLC for its latest architecture [21]. Non-inclusive caches can include data from lower-level caches, but do not have to. Representative of this are L2 caches in Intel's i7 series [22].

Starting with the Nehalem architecture, Intel processors divide the LLC not only into sets but also *slices*. The number of slices equals the number of cores. While all cores have access to the entire LLC, each has a slice associated with it that is closer on the connecting ring bus. Addresses are evenly divided among slices by a hash function of the address. Since Sandy Bridge, this function is undocumented [22], but methods for reverse-engineering it have been published for processors with 2^n cores [23, 24] as well as non-power-of-two cores [25, 26, 7]. In the first case, the hash function consists of an XOR

sum for all n bits required, as shown in Figure 2.2. Figure 2.1 illustrates the whole process.

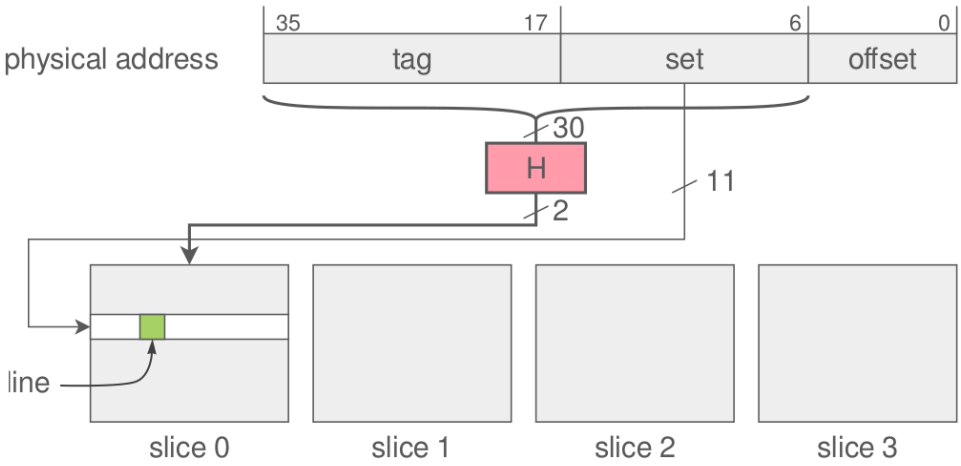


Figure 2.1: Addressing an Intel last-level cache with the slice function H [24].

		Address Bit																													
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0						⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 cores	o_0					⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1				⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕	
8 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕	
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕			⊕			⊕	⊕				⊕		

Figure 2.2: The slice function for 2, 4 and 8 slices [24].

2.1.1 Cache Attacks

From the properties described in the previous section, two "standard" techniques for side-channel attacks using the cache have emerged. Both are based on the fact that a cache miss takes a measurably longer time than a cache hit (see Figure 2.3). These small timing differences can be measured with the `rdtsc` (Read Time-Stamp Counter) instruction, which can be called from user mode.

To show the potential danger of these attacks, they have been used (or proposed) for cryptographic key-extraction [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 9, 7], key-logging [9], bypassing kernel ASLR [23], cross-VM application fingerprinting [37] and more.

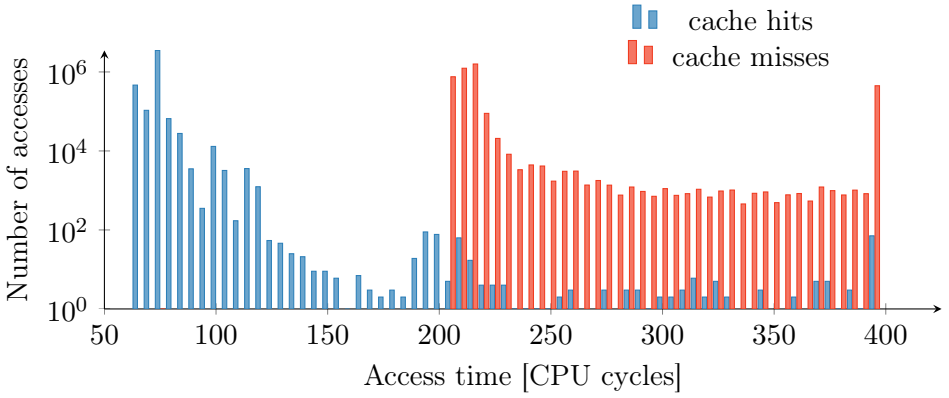


Figure 2.3: Latency for accessing cached and uncached memory locations.

Prime+Probe Named by Osvik et al. in 2006 [31], this method was first described by Percival in 2005 [30]. *Prime+Probe* (P+P) takes its name from the 2 steps performed by the attacker to extract information from a target process:

1. A cache set is first *primed* by evicting all of its contents by filling it with data from the attacking program. To reliably and efficiently evict a set requires some knowledge of the cache replacement policy. At the very least, this requires one access each to n addresses within the cache set, given an n-way set associative cache. More complex eviction strategies have been found that are effective on non-LRU policies [38] (Figure 2.4).
2. The same set is later *probed*, which means accessing the cache lines inserted in the priming phase and measuring the total time it takes to read them.

If the target program has used this particular set in the meantime, a number of the attacker's lines will have been evicted and the measured time will be larger than a previously calibrated threshold. One drawback of this method is that an attacker cannot know which memory access caused the evictions, or even which program. Another is the need for physical addresses to construct eviction sets for caches that are not virtually indexed. An advantage, on the other hand, is that this approach requires *no shared memory* with the victim.

Flush+Reload In 2014, Yarom et al. presented *Flush+Reload* (F+R), an approach to cache side-channels that greatly improves on the precision of Prime+Probe. Gullasch et al. already employed this technique in 2011 [39], but did not emphasize it. Flush+Reload is based on the `clflush` instruction, which evicts a given address from all cache levels. Again, as the name

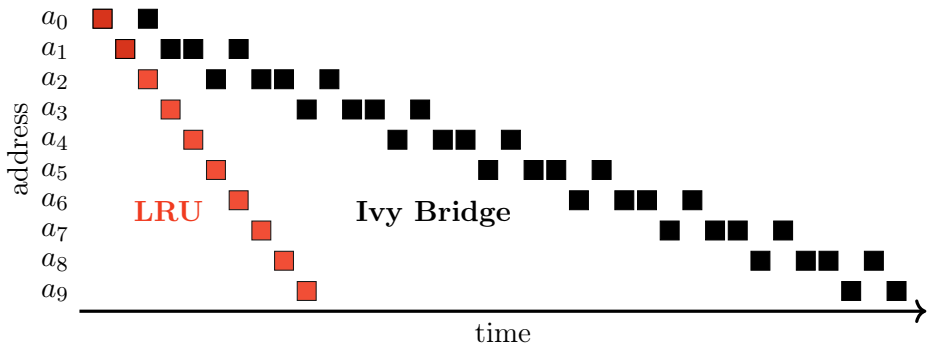


Figure 2.4: Access patterns for eviction of a line from a cache set for an LRU policy vs. the Ivy Bridge policy. Addresses a_0 to a_1 belong to the same cache set.

suggests, 2 steps are taken by an attacking program. An address in a memory region that is shared with the victim process is first *flushed* and then *reloaded* by accessing it. Timing this single access allows the attacker to determine with high confidence whether the victim process has used this specific address in the meantime or not. The granularity of this attack is thus the length of a cache line, usually 64 bytes.

Evict+Reload, Flush+Flush These are variations on Flush+Reload. The principle of removing an address from the cache, then checking for its reinsertion later stays the same. For CPUs or environments without access to the `clflush` instruction, *Evict+Reload* (E+R) [8, 40] is a possibility. Here `clflush` is replaced with an eviction of the entire set, similar to the prime step in Prime+Probe. *Flush+Flush* (F+F) [9] uses the fact that the `clflush` instruction, conversely to a simple access, takes distinguishably *longer* for its operation when the data to be flushed is in fact cached.

As mentioned in Section 2.1, the inclusion property of a cache level also applies in case of evictions. This means that in current Intel CPUs, eviction of an address from the cache on one core also causes this line to be evicted in the private caches of all other cores, as the inclusive LLC ties all caches together. Consequently, all described attacks can function across cores.

2.1.2 Cache Covert Channels

When we change the previous scenario from an *attacker-victim* relation to *attacker-collaborator*, we can create what is referred to as a *cache covert channel*. The monitored process works with the spy process by transmitting information to it through intentional cache manipulation. Such channels

authors	method	speed	BER	cache	CC	CVM
Percival, 2005	P+P	2.24MB/s	25%	L1	✗	✗
Percival, 2005	P+P	500kB/s	25%	L2	✗	✗
Ristenpart et al., 2009	P+P	0.025B/s	n/a	L2	✗	✓
Xu et al., 2011	P+P	0.4B/s	9%	L2	✗	✓
Maurice et al., 2015	P+P	94B/s	5.7%	LLC	✓	✓
Liu et al., 2015	P+P	75kB/s	1%[1]	LLC	✓	✓
Martineau, 2015	F+R	576kB/s	n/a	LLC	✓	✓
Gruss et al., 2016	F+R	298kB/s	0.00%	LLC	✓	✗
Gruss et al., 2016	F+F	496kB/s	0.84%	LLC	✓	✗

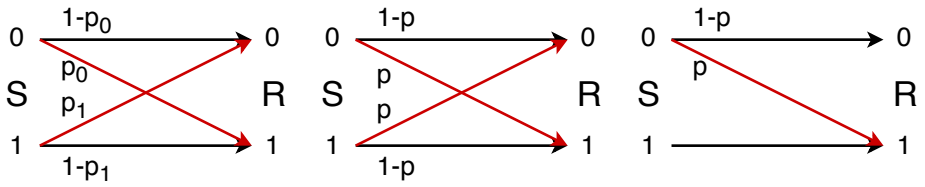
Table 2.1: A comparison of cache covert channels in transmission method, speed, bit error ratio (BER), cache level, cross-core (CC) and cross-VM (CVM) capabilities. [1]edit-distance

could be used to overcome isolation measures, be they between processes isolated by the operating system or in different virtual machines in the cloud. Since cache accesses are usually unmonitored, these channels are covert with regards to traditional information flow control mechanisms.

Since Flush+Reload based techniques require some form of shared memory, Prime+Probe is the approach to use in a cloud environment such as the Amazon EC2. After finding a solution to the mapping uncertainty in virtualization (see Chapter 3, C3&C4), priming and probing is used slightly differently from the attack scenario. To transmit a bit to the receiver, the sending process either evicts prearranged set(s) or does nothing to transmitting either 0 or 1. The receiving process will now access its probe data and measure the time this takes, thereby automatically caching them for the next read. The prime and probe steps are effectively split up between the 2 processes. How many lines are used for probing is a trade-off: using only one line is prone to eviction through noise while using too many can increase total probing time, hide misses or even cause self-eviction.

When the communicating parties are in the same operating system or memory deduplication is enabled, Flush+Reload provides significantly higher speeds. Since the used memory region is now shared and targeted directly, the challenges with address mapping do not arise. The concept is the same as for a Prime+Probe based channel, except Flush+Reload targets specific addresses instead of entire cache sets. An address is either flushed or accessed to transmit 1 bit; then the receiver only needs to time its access to this address and infer the bit's value. Evict+Flush as well as Flush+Flush work in a similar way.

Table 2.1 lists previously published cache channels.



(a) binary asymmetric channel (b) binary symmetric channel (c) Z-channel

Figure 2.5: Transition probabilities for different types of errors in binary channels.

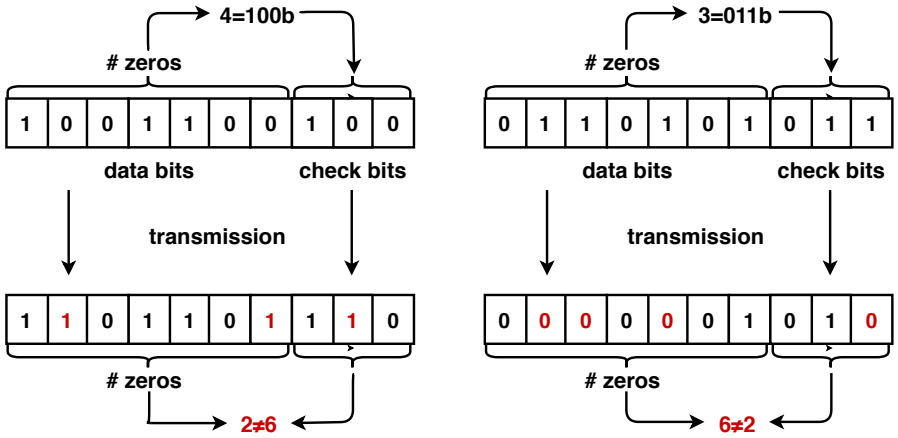
2.2 Error Detection Codes

Error detection codes (EDC) refer to encoding schemes a sender can apply, which will enable the receiver to recognize whether the encoded data has been corrupted. As the name implies, a pure EDC will only be able to detect errors, not correct them. Error-correcting codes (see Section 2.3) are a superset of EDCs since they can both identify and (possibly) correct errors.

There are many different approaches to EDC; one criterion for which to use is the type of error one expects on a particular channel. We can distinguish several types by their bit-flip probabilities, *i.e.*, how likely a bit is to transition from one state to the other during transmission. The most generic description is that of a *binary asymmetric channel*, in which bits flip from 0 to 1 with a probability p_0 and from 1 to 0 with a probability of p_1. The probabilities for correct transmission are then $(1-p_0)$ and $(1-p_1)$ respectively. Two special cases for a binary asymmetric channel are *binary symmetric channels* and *Z-channels*. Binary symmetric channels have a crossover probability of $p = p_0 = p_1$ for a bit to flip in either direction, thus the probability for a correct transmission is $(1-p)$. For Z-channels either $p_0 = 0$ or $p_1 = 0$, which means that bit errors can only occur in one direction. Figure 2.5 visualizes these 3 types.

The errors Z-channels create can be called *unidirectional*, as they can only flip in one direction. It is not important which direction this is and might even change from word to word.

Among the multitude of existing EDCs, we want to examine the Berger code more closely in the next section because of its significance in this thesis.



(a) 0-to-1 bit-flips

(b) 1-to-0 bit-flips

Figure 2.6: Generation of the Berger code and subsequent error detection for both cases of unidirectional errors.

2.2.1 Berger Code

Named after its inventor J.M. Berger [41], the *Berger code* can be used to detect any number of unidirectional bit-flip errors in an encoded word.

For n bits of data, the Berger code requires $k = \lceil \log_2(n+1) \rceil$ check bits. The check bits are simply the binary number of zeros in the data word, which is then encoded by simply pre- or appending the check bits. When all unidirectional errors need to be detected, and all 2^n symbols can occur in the word, the Berger code is optimal. As an EDC, it provides no possibility for correction.

We can easily reason why all unidirectional errors can be detected: assuming only 0-to-1 bit-flips occur within an encoded word, every flip in the data bits decreases the number of zeros by one. Every flip in the check bits increases the number of expected zeros in the data bits. We can see that no matter where a bit-flip happens, it can only increase the difference between the number of zeros stated by the check bits and the actual number of zeros in the data bits. The 1-to-0 case can be derived in the same way. Figure 2.6 illustrates both scenarios.

2.3 Forward Error Correction

Error Correction Codes (ECC) provide redundancy to noisy data transmissions, such that a certain amount of errors may be detected and corrected by

the receiver without retransmission, i.e., error correction with only forward transmission.

They can be split into two general groups, block codes and convolutional codes. Block codes operate on blocks of data with a predetermined size, while convolutional codes work with data streams of arbitrary length.

2.3.1 Reed-Solomon Codes

Reed-Solomon codes (RS codes) were proposed in 1958 by Reed et al. [42] and have since become a widespread and well established forward error correction scheme. They are optimal block codes, in the sense that they are maximum distance separable, which means that for fixed parameters n and k, they provide the maximum error correction [43].

RS-Codes operate on fixed-size blocks (codewords), comprised of n symbols: k data symbols and $n-k$ parity symbols, noted as RS(n, k). The length of codewords is determined by the symbol size s, $n = 2^s - 1$, e.g. RS(4096,4055) for 12-bit symbols with $\approx 1\%$ parity. Codewords can be smaller than this if the encoder pads unused symbols with zeros before encoding and does not transmit them. The decoder can then add the same amount of zeros before decoding. For errors in unknown positions (r), up to $r \leq \frac{n-k}{2}$ symbols can be detected and recovered. When the position of errors is known, so-called erasures (s), RS codes can recover up to $s \leq n-k$ symbols. In total, $2s + r \leq 2(n-k)$ symbols can be correctly recovered.

Chapter 3

Analyzing Challenges

This chapter serves as a broad overview of our contributions to the field in the form of challenges we faced when implementing our cross-VM cache covert channel. This list is not exhaustive, but the challenges discussed are those that, to the best of our knowledge, had so far not been overcome for our specific requirements.

3.1 Challenge C1: Virtualized Timers

One of our early design decisions was not to make any assumptions about the `rdtsc` instruction besides its short-term accuracy. This means that we do not need it to be continuous or even monotonically increasing at all times, but we do expect close-to-native performance and accuracy over the short periods of time it takes to measure cache latency. Importantly, we also expect no fixed relation from one VMs time stamp counter to another's. The reason for this decision is that different hypervisors may virtualize `rdtsc` differently and each have various options to choose from. Building a channel based on any one particular configuration would restrict its generality, which is something we wanted to avoid.

This creates a problem: how do we synchronize sender and receiver when their perceived time might move backward and forward in relation to each other? What if time "jumps" for one party, but not the other? Clock drift on its own is something that many everyday communication channels deal with. One of the solutions, for example, is the use of a self-clocking signal. However, the discontinuities in the clock we can have might be many times larger than the period of such a self-clocking signal, which would defeat its purpose. We could find no analog to this particular circumstance. A consistent source of such discontinuity is discussed in C2.

We will find a solution to this challenge in Section 4.2.

3.2 Challenge C2: Scheduling Difficulties

Besides small inaccuracies in a virtualized `rdtsc` instruction, the biggest problem stems from the fact that two counters in different VMs cannot be expected to be synchronized. If one VM is descheduled for whatever reason, we cannot assume the counter will be the same relative to the other VM after it is resumed, as it would be if the instruction were run natively. Of course, this is reasonable in the context of virtual machines, as they aim to present an environment that looks "native" to programs within it.

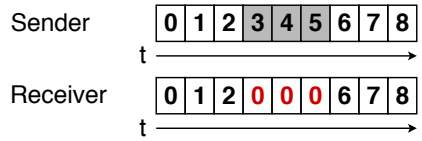
When creating a cache covert channel in a native environment, the simplest way to synchronize a sender/receiver pair is to partition time into small segments (on the order of several thousand cycles) that correspond to a specific position in the transmitted data. This introduces a more manageable problem: scheduling on the OS level. When sender and/or receiver are descheduled, they still know exactly what data to read/write when they resume their operation. The data sent or read in between is lost or wrong. In this scheme, when sender or receiver are descheduled, it results in *substitution* errors. They can be both detected via `rdtsc` and possibly corrected with ECC (Section 2.3). When we transfer this scheme to communication between VMs, their scheduling can cause *insertion* or *deletion* errors, in addition to substitution errors caused by process scheduling. Figure 3.1 depicts the differences of these error types.

The difficulty with insertion and deletion errors is that synchronization is implicitly lost. Because we cannot rely on `rdtsc` to find out if, when or how often this happened, the data cannot trivially be restored. The challenge is thus to find a transmission scheme that can tolerate the descheduling of VMs and processes and produce correctable results. Conceivably, the 2 main strategies here are avoiding insertion and deletion errors entirely or encoding the data heavily enough such that it can be restored.

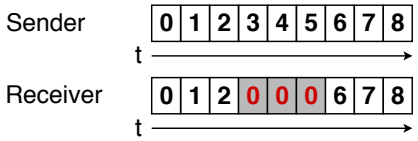
Again, this challenge will be solved in Section 4.2.

3.3 Challenge C3: Address Mapping in Virtualization

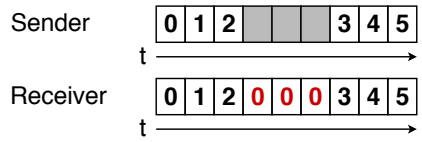
As mentioned in Section 2.1, the last-level cache uses physical addresses for indexing its cache sets and determining the cache slice. Since user space programs operate entirely on virtual addresses, the first step for a native



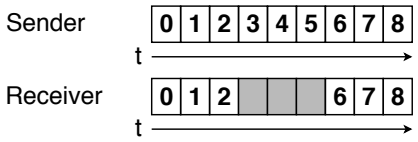
(a) *Substitution* errors caused by sender-**process** descheduling. Missed time is tracked.



(b) *Substitution* errors caused by receiver-**process** descheduling. Missed time is tracked.



(c) *Insertion* errors caused by sender-**VM** descheduling. Missed time is not tracked.



(d) *Deletion* errors caused by receiver-**VM** descheduling. Missed time is not tracked.

Figure 3.1: Transmission of the byte sequence `012345678` for 3 different error types (substitution, insertion, deletion). Grey background signifies a descheduled thread. Note the difference between substitution and insertion/deletion errors in received length.

Prime+Probe channel is to find out how its virtual addresses map to physical addresses. From there, sets of addresses that are located in the same cache set can be created. On Linux systems, a process with root privileges can read this information from the virtual file */proc/[pid]/pagemap*. In a virtual machine, this is not possible, even if the process could somehow obtain root privileges. That is because the hypervisor adds another layer of indirection; VMs are applications that operate in a virtual memory space themselves. What an operating system in a VM perceives as machine addresses are in fact virtual addresses already, consequently the pagemap file cannot provide the correct information here.

This challenge is, therefore, the question of how to determine virtual addresses' cache set indices and slices without access to the physical addresses.

We present our solution in Section 4.3.

3.4 Challenge C4: First Contact - Establishing the Channel

Once we have overcome all other obstacles, what remains is the very beginning of communication in our channel. To start transmitting data, the sender needs to know when the client is ready, and vice versa. Additionally,

both need to know what cache sets to use for communication and in which order they are to be used.

Exchanging this information is, by definition, already communication, and herein lies the challenge: How can sender and receiver exchange information before these basic parameters of cache channels are established?

This question will be answered in Section 4.4.

Chapter 4

Implementation

4.1 Native Prime+Probe

The starting point for this work is a native Prime+Probe channel as described in Sections 2.1.2, C2 and C3. For now, we will still use `rdtsc`, root access and shared memory, because this channel will serve as a testbed to study the characteristics of Prime+Probe channels and as a benchmark for the following sections.

We begin by translating the virtual addresses in a block of memory with the pagemap file to get the physical addresses and create cache sets from the reverse-engineered hash function [24]. For this we use shared memory, which enables one party, in this case the sender, to determine the necessary amount of addresses in the same cache sets and let the receiver know by writing the determined probe addresses to the shared memory. Since we use fixed-size blocks of cycles (transmission windows) for each word that is transferred in parallel (as mentioned in C2), all that is required to synchronize this channel is a common starting-cycle count, which is also determined by the sender and given to the receiver via the shared memory.

After this initial setup, the data transfer is straightforward. Starting with the arranged initial cycle count, the sender continuously evicts all cache sets corresponding to 1-bits in the currently transmitted word. Meanwhile, the client repeatedly reads all sets with its probe addresses and counts hits and misses on all sets. At the end of every transmission window, the receiver decides for each set if it was a 0 or 1 by comparing the number of hits and misses. Now sender and receiver start the same procedure for the next word until a predefined file length has been received. For this to work, the sender and receiver would ideally read and write in a perfectly interleaving pattern, so that every read of the receiver is preceded by a corresponding

write. Since perfect interleaving is unrealistic, it is enough to ensure that the sender writes more often than the receiver reads. Listings 4.1 and 4.2 show shortened versions of the sender and receiver.

```
1  inline void evictSet(volatile uint64_t** addrs)
2  {
3    for (int i = 0; i < 15; ++i)
4    {
5      *addrs[i];
6      *addrs[i + 1];
7      *addrs[i];
8      *addrs[i + 1];
9    }
10 }
11
12 void writeWord(size_t end, uint32_t data)
13 {
14   while(rdtsc() <= end)
15   {
16     for (int i = 0; i < WORD_SIZE; i++)
17     {
18       if (data & (1 << i))
19       {
20         evictSet(eviction_addresses);
21       }
22     }
23   }
24 }
```

Listing 4.1: Sender

```
1  inline void accessSet(volatile uint64_t** addrs)
2  {
3    for (int i = 0; i < 3; ++i)
4    {
5      *addrs[i];
6      *addrs[i+1];
7      *addrs[i+2];
8      *addrs[i];
9      *addrs[i+1];
10     *addrs[i+2];
11   }
12 }
13
14 uint32_t readWord(size_t end)
15 {
16   int hit[WORD_SIZE];
17   int miss[WORD_SIZE];
18
19   size_t time, time_tmp, delta[WORD_SIZE];
```

```
20    int reads = 0;
21
22    while(rdtsc() < end)
23    {
24      time = rdtsc();
25      for (int i = 0; i < WORD_SIZE; ++i)
26      {
27        accessSet(probe_addresses);
28        time_tmp = rdtsc();
29        delta[i] = time_tmp - time;
30        time = time_tmp;
31      }
32      for (int i = 0; i < WORD_SIZE; ++i)
33      {
34        if (delta[i] < MIN_CACHE_MISS_CYCLES)
35          hit[i]++;
36        else
37          miss[i]++;
38      }
39    }
40
41    uint32_t output = 0;
42
43    for (int i = 0; i < WORD_SIZE; ++i)
44    {
45      output |= ((hit[i] < miss[i]) << i);
46    }
47
48    while (rdtsc() < end);
49
50    return output;
51  }
```

Listing 4.2: Receiver

For the number of addresses to evict a cache set, we choose the number of ways in the LLC (here 16). We evict the set using the alternating eviction pattern proposed by Gruss et al. [38] (see Figure 2.4). For probing we use 5 addresses in total, with the access pattern that can be seen in Listing 4.2. The intent is to get a reliable measurement to detect eviction and, at the same time, ensure the probe lines are cached again for the next read. This pattern and the number of lines used proved to be suitable and to provide good resistance to unintentional eviction by noise. Since we expect a lot of slowdowns in future steps, we optimize this channel for speed while also maintaining reasonable bit error ratios (BER).

4.1.1 A Statistical Analysis of Errors in Prime+Probe

With this native Prime+Probe channel as a baseline, we can analyze the types of errors it produces in normal operation and how they vary with factors such as stress on memory by other programs. The conclusions we draw from this will allow us to make sensible decisions regarding the design of our channel with respect to the challenges we have to overcome.

We choose the data word size to be 8 bits; not only because it is convenient, but also since we can see no significant speedup for larger word sizes. An additional 4 code bits, which we will use at a later point (Section 4.2.2), bring the total word length to 12 bits. The raw capacity of this channel is determined only by the chosen amount of cycles used for each word. For this experimental setup, we set the transmission window size to 50000 cycles. On a processor with an `rdtsc` rate of 3.5GHz, this leads to a raw data channel capacity of 547kB/s (820kB/s including the code bits). In this transmission scheme, this parameter essentially controls the trade-off between noise resistance and speed.

The data used is gathered in 3 series of 100 runs each, where one run is a transmission of 1MB of *JPEG* data. The first series is generated on a system with minimal program activity besides the transfer, hence we will refer to it as "quiet." To have a repeatable level of noise in the cache, we generate series 2 and 3 with the addition of the benchmarking tool *stress* [44] with the options "-m 1" and "-m 2" respectively (noted series m1/m2 in the remainder). *Stress* starts the given number of threads and performs repeated memory accesses, each thread using one CPU core at full capacity.

When talking about a substitution error in the following, this does not technically refer to exactly one accidental cache eviction (or non-eviction), it refers to the result after the reading process in the client, as described in the previous section. The statistics collected refer only to the 8 bits of data in each word, unless stated otherwise.

Figure 4.1 shows the average bit error ratio for each run in all 3 series. The effect of the stress tool can clearly be seen here — where on a quiet system the channel encounters error ratios as low as 0.01%, even series m1 already suffers BERs up to 15%. Looking at the large differences between runs of the same series, one might suspect large variance inside the runs is hidden in the average BER values. However, as Figures 4.2a and 4.2b demonstrate, this is not the case. Both runs are very stable over the course of their entire transmission, yet completely different, even though they were recorded within 45 seconds of each other. The graphs also show the BER of the best and worst cache sets for each run, which show a vast difference from each other in Figure 4.2a. It is this difference that explains why one run can have

a low error ratio while the next does not, even though seemingly nothing in the environment has changed: since the cache sets are chosen from memory assigned by the operating system at the start of a transmission, the sets used are essentially random. In a high-noise environment, the probability of having many low-noise sets is reduced and thus the expected error ratio rises.

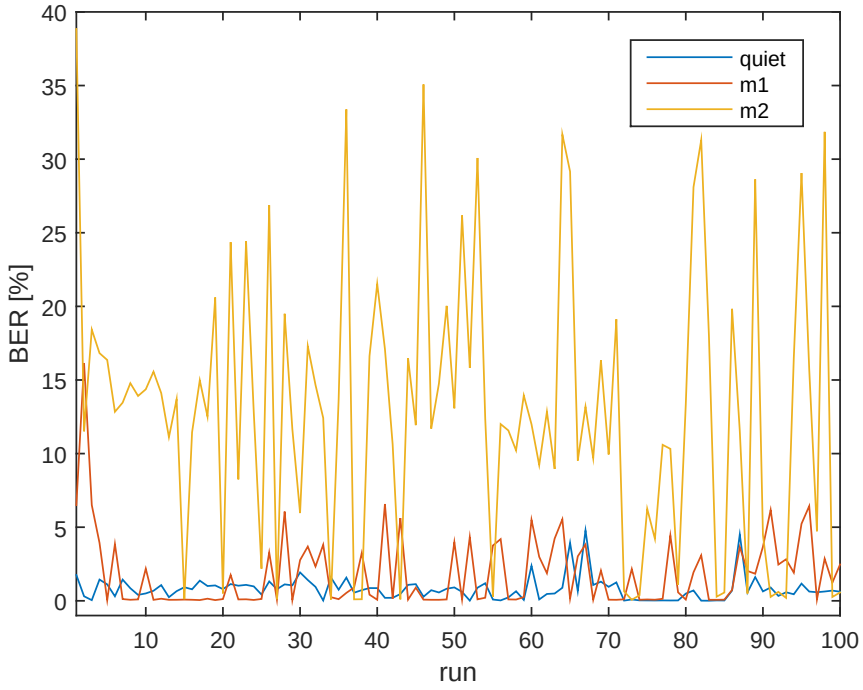
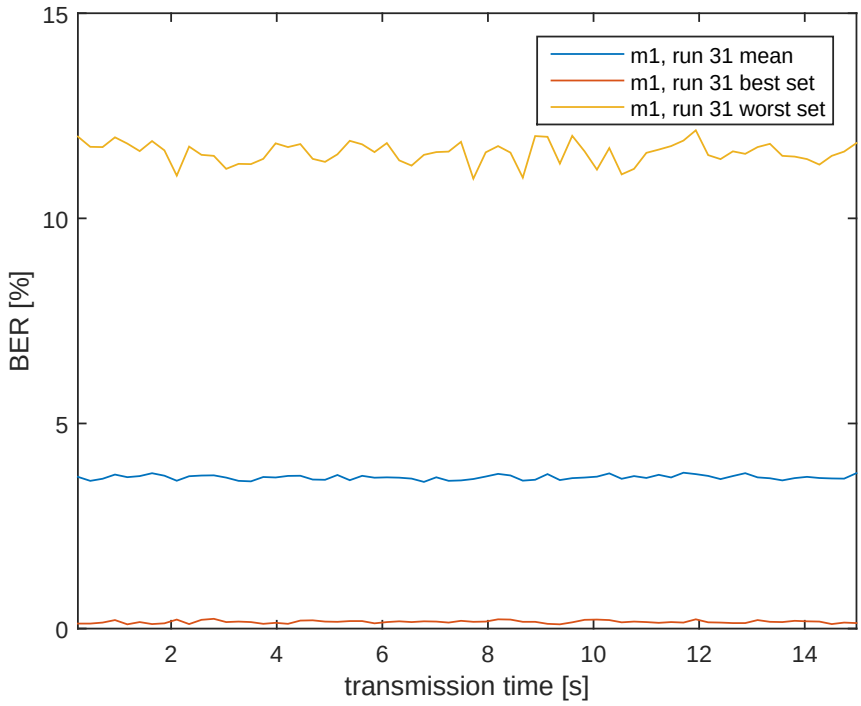


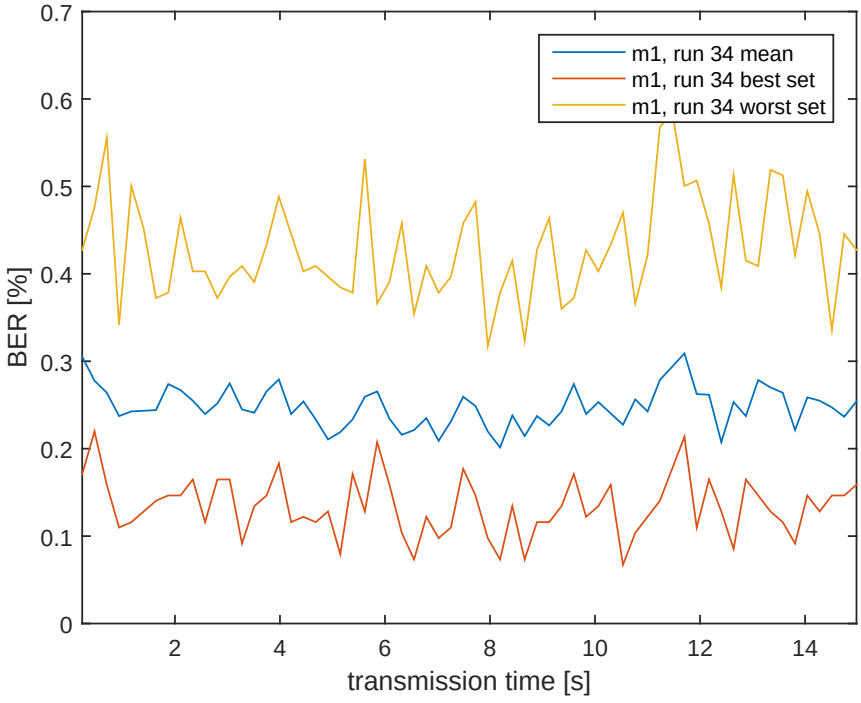
Figure 4.1: Average bit error ratio of each run for all 3 series.

To be able to characterize this channel in terms of the types discussed in Section 2.2, we look at how and when substitutions occur. In particular, how often do non-unidirectional substitutions occur in words, *i.e.*, out of all defective data words in a transmission, how many have both 1-to-0 and 0-to-1 substitutions? Figure 4.3 answers this question for our data set. It shows that, on average, 99.6% of all words contain only unidirectional errors, and even in the worst cases this rarely drops below 98% (with 1 outlier in series m1 and another in m2, at 93.75% and 91.6% respectively). We can observe only a weak correlation between BER and non-unidirectional errors: the high-noise series shows a higher average than the low-noise series (0.4% vs. 0.35%), but no direct correlation within a series.

Within this non-unidirectional subset of errors, we can also look at the ratio of 0-to-1 versus 1-to-0 substitutions, see Figure 4.4. Between 50-65% of these words contain the same number of 0-to-1 substitution as 1-to-0 substitution.



(a) run 31, series m1



(b) run 34, series m1

Figure 4.2: Average bit error ratios of two different runs with stress -m 1 as well as their respective best and worst cache sets over the course of the transmission.

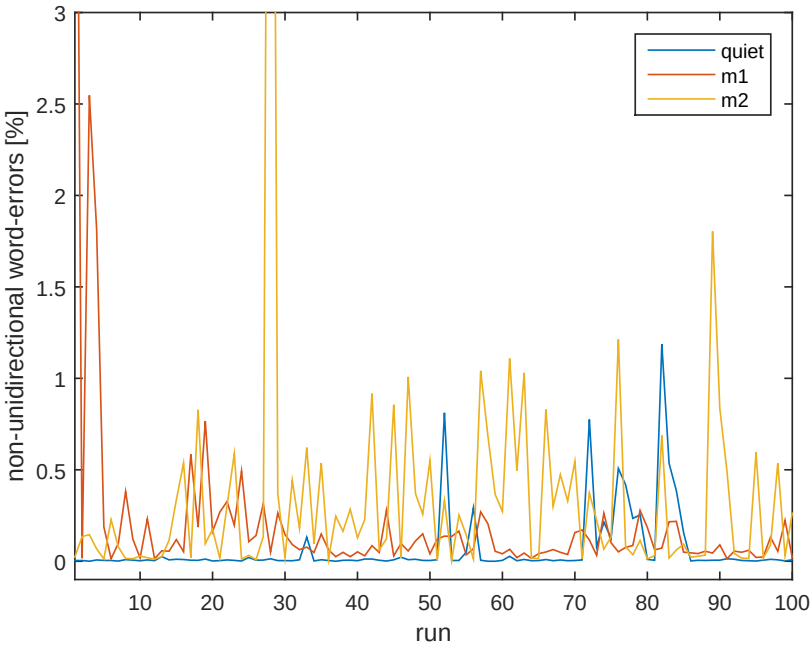


Figure 4.3: Percentage of non-unidirectional bit errors in words out of all words with errors, for each run and all 3 series.

The remaining possibilities show a bias towards 0-to-1 substitutions, on the whole, the distribution is fairly independent of the noise level.

In conclusion, this channel behaves like a Z-channel in most circumstances, and like a biased binary asymmetric channel otherwise. This characteristic will be useful in Section 4.2, as we choose an error detection code.

4.2 Replacing the `rdtsc` Instruction in Synchronization

To address challenges C1 and C2 simultaneously, we devise a message request scheme to replace synchronization via absolute `rdtsc` counts. While relatively simple in abstract terms, this solution comes with its own set of problems that need to be solved.

As illustrated in Figure 4.5, this method introduces a second, backward communication channel from the receiver to the sender. Its sole purpose is to let the sender know when one word has been correctly received and the sender may transmit the next. This is achieved by transmitting a sequence number with a word of data; when it is read correctly, the back-channel changes its message to the next sequence number, thereby acknowledging

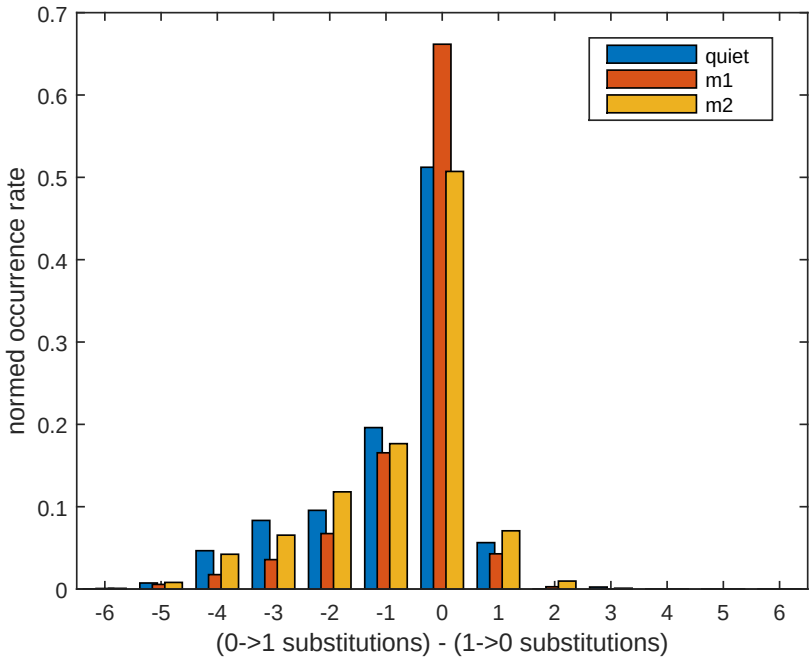


Figure 4.4: Distribution of substitution errors in non-unidirectional words, for all 3 series.

the correct reception. Unlike in Figure 4.5, however, this process is not sequential, but parallel. Neither sender nor receiver waits for a reaction before transmitting; both do so continuously. This solves challenge C2 since a descheduling of one or both virtual machines (or threads) has no influence on their internal sequence counters and thus the synchronization of the channel. For the same reason, it also solves challenge C1, as any discontinuities in the virtualized `rdtsc` instructions will have no consequences besides possible bit errors, should an irregularity occur during the reading of a word.

On the receiver side, the sequence number that is read allows to decide if the currently transmitted word is indeed that which was requested. Because the back-channel has the same problems with descheduling and noise as the data channel, it is possible that the sender will not receive the request to transmit the next word for some time. During this time, it is up to the receiver to detect this and reject the data. Being part of the transmitted word, the sequence number itself is also subject to substitution errors, and so an old sequence number can change to be the expected number instead. To prevent the receiver from accepting such a word as correct, we extend the word with an error detection code, which covers the sequence number and the data. Only when the sequence number and the error detection code are in order is a word accepted and the request number increased. For the same reason, the sequence number request on the back-channel is also encoded.

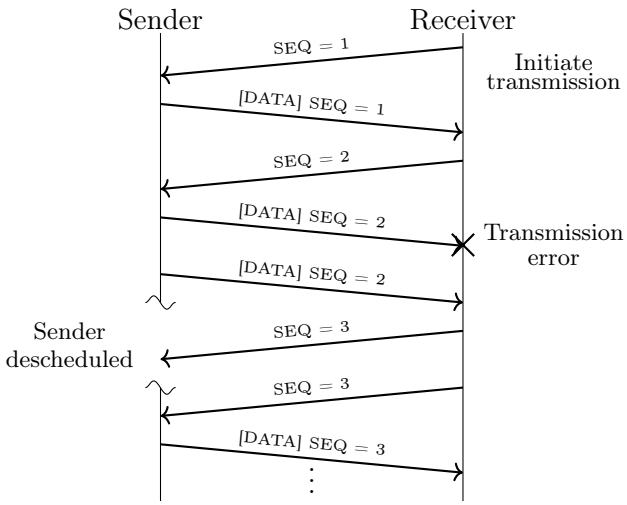


Figure 4.5: Retransmission due to scheduling interrupts or substitution errors in transfer [10].

4.2.1 Reading with a Sliding Window

While the sender works in the same way as it did in the `rdtsc`-synchronized implementation (that is, write without stopping until signaled to move to the next word), the receiver now does not have a predefined window of time in which to read. So as to not endlessly accumulate the number of 0 and 1 reads for a bit, we implement a sliding window. The window size determines how many reads are considered for the value of a bit. As long as the current word has not been accepted (either because of the sequence number or an EDC error), reads are added to and removed from this circular buffer. The advantage of a limited buffer is that it allows reads of old data or burst errors to be discarded after a while. With a small window size, this can happen faster, lowering the average amount of reads required for a word in a quiet cache. The buffer also fills more quickly, which allows some words to be read correctly within the first 3 reads when there is virtually no noise. However, because noise related errors in bits are not linked, they do not necessarily occur at the same time in each bit. In a very noisy environment, a small window means that a part of the bit-reads might always be correct, but how many such reads are in the window at the same time can change with each read. A larger window size increases the chances of correctly reconstructing each bit in the buffer at the same time, allowing it to overcome small clusters of errors, even if they occurred at different times. Figure 4.6 depicts such a high-noise scenario where a window size of 3 is too small to reconstruct the correct 3-bit word at any point in the sequence of reads, but a larger size succeeds right away. While the repeating error pattern is unlikely to

occur exactly this way in a real transmission, it demonstrates the advantage of larger windows as transmission quality declines.

Figure 4.6: Example of a 40% bit-error read-sequence in a 3 bit word, represented as correct (✓) and incorrect (✗) reads. A window of an odd size >3 can reconstruct the correct word at any point in this sequence of 10 reads, while a window of size 3 cannot.

To perform as well as possible in a changing environment, we allow this window size to adapt dynamically with the number of EDC errors per word. Starting with a minimum size of 1, after a certain amount of EDC errors, this is increased by 2, up to a limit. This allows our channel to take advantage of a low noise environment when possible, yet still be reliable in the presence of heavy noise.

4.2.2 Detecting Errors

To know when a word was correctly transmitted, some kind of error detection code is necessary. There are several factors we consider in our decision:

- encoding and decoding speed: as a cache channel's performance largely depends on how many CPU cycles it can use to send or receive data, a fast algorithm is required
- length: achieving a high bandwidth is a goal of this channel, so the code to data ratio should be as small as possible. Moreover, the total word length needs to be small as well, see Section 4.5
- detection capability: because of the design of our channel, the code needs to be able to detect *all* errors in a word

Because descheduling of the sender in combination with accidental eviction through noise can often cause any number of bits in a word to be wrong, codes that only detect up to a certain number of substitution errors cannot be used.

While it would be convenient to have some error correction capability as well, forward error correction codes necessarily add more redundancy, and are often not suited for correcting on the level of single words *and* detecting all errors at the same time. The self-correcting design of this system is another reason why we use a pure EDC on the word-level.

Because overall this channel behaves almost entirely like a Z-channel, the Berger code, as introduced in Section 2.2.1, fits our needs very well. With a code length of $k = \lceil \log_2(n+1) \rceil$ for n bits of payload it is quite small but still detects all errors. Because encoding and decoding consist only of counting the number of zeros in the payload, it is also very fast.

As per the statistical analysis in Section 4.1.1, we know that, in an average of 99.6% of all words that contain errors, the substitution errors are completely unidirectional, and so the Berger Code will detect all of them. Assuming this average were to drop to <91% in very noisy environments, the question arises if the Berger code would now fail to detect 9% of errors. Considering that the bits of the code itself are also affected by the same errors, the issue is not trivial. A substitution in any but the lowest bit of the Berger code requires several opposing substitutions in the payload to go undetected. Because the distribution of 0-to-1 vs. 1-to-0 substitutions shows a large probability of an equal amount of substitutions (see Figure 4.4), those two facts together suggest that actually, even among non-unidirectional errors, we can expect many to be detected. Pictured in Figure 4.7, our previous experiment, now with added EDC, shows that the detection rate is indeed more than 98.6% at all times.

With the Berger code selected, we add an additional measure to mitigate noise caused by sender descheduling. Since there are only 2 ways to create 1-to-0 substitutions, namely failure to evict and sender descheduling, we make the assumption that a word consisting entirely of zeros was most likely caused by a sender descheduling. We can make this assumption because firstly, the Berger code does not allow all zero words and secondly, the probability of so many eviction failures in one word is very low. So when a zero word is read, it is dropped from the current window of considered reads. This allows the receiver to move past burst errors caused by descheduling more quickly when no additional 0-to-1 substitutions occur.

4.2.3 Sequence Numbers

Once a data word has passed the EDC check, its sequence number needs to be validated as well. As we have shown previously, in a small percentage of words noise will produce a valid code for a wrong data word. At this point, the next step is to determine a suitable bit-length for the sequence number, balancing bandwidth and noise resistance.

We consider this example for the minimum sequence number length of 1 bit, as illustrated in Figure 4.8. A 1 bit sequence number will, statistically, at some point be inverted while still producing a valid EDC. When this happens soon after a word was accepted, this erroneous sequence number

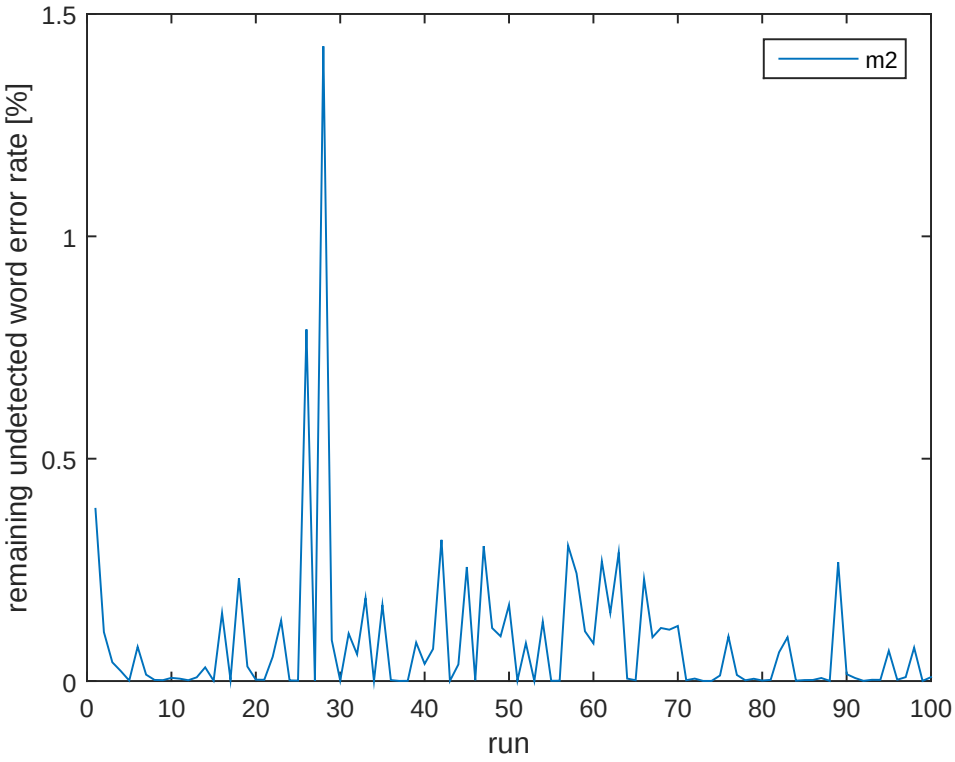


Figure 4.7: Percentage of non-unidirectional bit errors in words out of all words with errors, for each run and all 3 series.

will now match the very sequence number the receiver expected next, as there are only 2. This means the receiver will now change its back-channel message again, asking for the next sequence number, which also equals the previous one. Since the receiver has accepted the current word so quickly (as it was produced by an error), it is not unlikely that the sender has not yet successfully read the request at all. From the perspective of the sender, no word was transmitted at all by this point. The receiver, on the other hand, believes to have received 2 correct words and expects a third. At this point the sequence number has failed to do what it was designed for: sender and receiver are desynchronized.

Unfortunately, accepting a previous transmission as the current one because of substitution errors is a problem that persists even for longer sequence numbers. The consequences of this skipping, however, change. In sequence numbers of 2 bits or longer, a number's predecessor never equals its successor, so the result is a deadlock where the sender is delivering a word that is too old, and the receiver requests a word that is too far ahead. This deadlock can only be broken by more substitution errors, either allowing one party to catch up or the other to skip further ahead, increasing the problem.

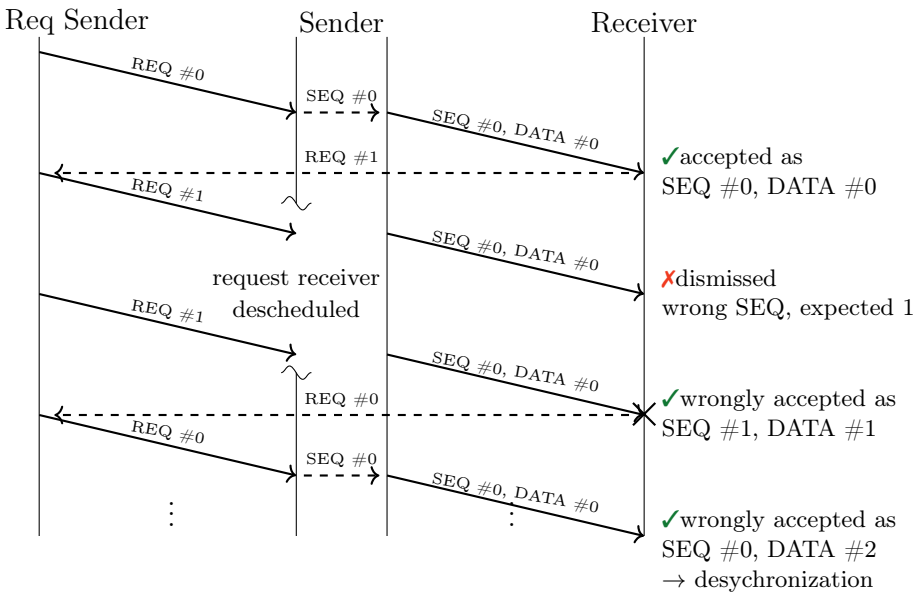


Figure 4.8: Desynchronization as a result of a failure to read in the sender's request receiver and substitution errors in the sequence number in the data packet side.

In the interest of achieving predictable outcomes, relying on substitution errors to rectify a situation caused by substitution is clearly not the answer. To solve this, we implement a mechanism by which the party expecting the lower sequence number can catch up and resolve the deadlock. However, simply accepting the expected sequence number $+1$ as valid extends the previous 1 bit scenario because it removes the requirement of a descheduling or failure to read in the other party. Any sequence number can now change to $+1$ and be accepted, which introduces more of these errors. While this simple implementation succeeds in reducing the number of desynchronizations, it also increases the bit error ratio, because every skip creates 2 words of broken data. We mitigate this with a strategy we call *n-strike-skip*. As the name suggests, a word is not immediately skipped as soon as an EDC-validated sequence number is read that is ahead by 1. Instead, this is recorded, and the read buffer is emptied. Only when the same sequence number is read n times is it accepted. This measure virtually eliminates noise-induced skips meant to catch up, because it requires the read buffer to randomly produce the same wrong sequence number n times, before the correct one.

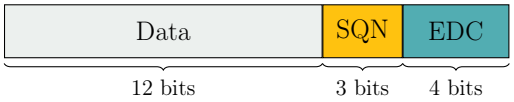
We will determine a suitable value for n and show that the n-strike-skip policy does indeed prevent unnecessary skips in Chapter 5, but in the search for an appropriate sequence number length in this section, it creates a new constraint. When we revisit the same scenario as above for 1 bit but now use

2-strike-skip and a 2 bit sequence number, we find a similar, albeit less likely outcome. Again, the receiver correctly reads a word and proceeds to request the next one, but again, the thread receiving the requests is descheduled at that moment. Because the receiver will not accept the current sequence number of the sender, it is essentially waiting for substitution errors. A single substitution toward the expected number is not a problem, as the sender can catch up once it is scheduled again. In the rare case that a second substitution error to the next expected sequence happens, the receiver then expects a sequence number of +3 relative to the one that the sender is still sending. Within limited space of 2 bits, unfortunately, +3 is the same as -1. The receiver will consider itself behind the sender and skip ahead, desynchronizing itself by 4 words.

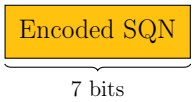
For 3 bits, this scenario becomes even more unlikely, as it would take 5 consecutive EDC-valid sequence errors to circle back to the beginning and cause desynchronization. Of course, 2 consecutive errors are still possible, but result in a deadlock. This problem affects sequence numbers of any length, so we devise a solution similar to skipping, *backtracking*. By allowing either party to move its sequence counter backward from 2 up to $\frac{seq\_length}{2}$ positions, deadlocks can be resolved. Because backtracking is only intended for a deadlock scenario, its conditions are much more stringent than skipping. In a deadlock, the expected value is not sent at all, thus it is harder to differentiate noise from the correct sequence number. For this reason, we implement a high, empirically chosen minimum number of reads for the same sequence number before a backtracking action is triggered. Because there are $\frac{seq\_length}{2} - 1$ possible backtracking positions, the number of reads for all of them is tracked. Once a position has been read more than the required minimum, we additionally check the read ratio of the highest to the other positions. This is intended to prevent a scenario where there is no real backtracking situation and several sequence numbers appear randomly.

Even under a lot of stress, the channel proves stable with theses measures in place, so we settle on a 3 bit sequence number. For 12 bits of data and 3 bits EDC, the required length of the Berger code is 4 bits. The resulting structure of an encoded word can be seen in Figure 4.9a.

Lastly, the request on the back-channel needs to be encoded as well, because the same types of errors apply to it. If we disregard packets that were already received, the data channel and the back-channel essentially operate sequentially. For this reason, high speeds on back-channel also increase the overall transmission speed. At the same time, this may not come at the cost of its primary function: synchronization. We could again use the Berger code for this, but a 3 bit word only encodes to a total of 5 bits. Because testing proves this to be too error-prone, we use the Hadamard code [45] instead. With this code, 3 bit sequence numbers are encoded to 7 bits (Figure 4.9b),



(a) Structure of an encoded word on the data channel.



(b) An encoded request on the back-channel.

Figure 4.9: Structures of data and request packets [10].

relative sequence	action taken
-3	backtrack
-2	backtrack
-1	reject, the other party will skip
0	accept
1	skip over 0 & accept
2	reject, the other party will backtrack
3	reject, the other party will backtrack

Table 4.1: Action taken by the receiver according to the received relative sequence number (= expected sequence number - received sequence number).

providing a Hamming distance of 4 between all of them. While the Hadamard code provides the ability to correct 1-bit errors, more testing shows that the error ratio is still too high and so we will not employ this feature. Further, because '0' is encoded to '0', we use only sequence numbers 1 to 6, as the descheduling of the sender often leads to zero words.

Table 4.1 lists the actions taken in response to the received relative sequence numbers.

4.2.4 Read Delay

In Prime+Probe, priming typically takes longer than probing due to the number of addresses that need to be accessed. This can lead the receiving party to probe a set with a higher frequency than that of the sender priming it, causing erroneous reads. Our solution is to introduce an artificial delay on the receiving side, implemented as a simple loop. The right delay will keep the receiver from probing too quickly and decrease its error ratio. However, the time needed to prime (and probe) depends on the number of addresses that need to be accessed, how many of those need to be loaded from memory

and the time it takes to do so. This changes with the data, the current stress on memory and the hardware. Though we can determine a constant delay that is high enough to accommodate most situations, this is far from optimal. A channel transmitting data on a system without any significant stress on the cache might achieve optimal speed with no delay at all, while a channel on the same system under heavy load will need a significant delay to continue operating without errors.

To approach optimal transmission speed, we implement a simple control loop to determine a dynamically changing delay. Because there is no direct measure to determine how high this delay should be, we look to the number of wrong reads per word and try to keep this measure at a set, empirically determined value by increasing or decreasing the delay with each update. As any calculations done within the read loop potentially slow the transmission, we only update every few hundred words and keep track of the current average of wrong reads over a larger sliding window.

4.3 Finding Cache-Set-Congruent Addresses

Now that the channel is ready to stay synchronized without `rdtsc`, it still needs a way to find addresses that belong to the same cache sets. This section solves challenge C3.

Looking at previous works, we can see that Liu et al. have already achieved this in 2015 [13], but they do not use the hash function that determines the slices. Because of this, they need to construct eviction sets for each set index individually, assuming the set index and slice hash function use overlapping bits of the physical address. They do this by starting with all addresses of the same index in a large block of memory (2 times the size of the LLC) and reducing it first to a set containing n addresses (for n-way caches) per slice and then separating this large set into a distinct eviction set for each slice. As this process takes about 0.2s per set index, for our requirement of 26 sets this would amount to 5.2 seconds. Since we can rely on the reverse-engineering work done by Maurice et al. [24] and use the now known hash function, we attempt to streamline this process for our channel.

Just like Liu et al., we require the hypervisor and operating systems to support 2MB huge pages. A continuous 2MB block of memory reveals the lower 21 bits of the memory's physical addresses, as these blocks are always aligned to 2MB boundaries. This allows us to immediately determine an address's set index from bits 6-16. The only thing left to ascertain is the slice of each address. Given the full physical address, we can determine what we will refer to as the *physical slice index* SIp. However, since the slice function uses bits beyond the 21 that are known to us, we can only

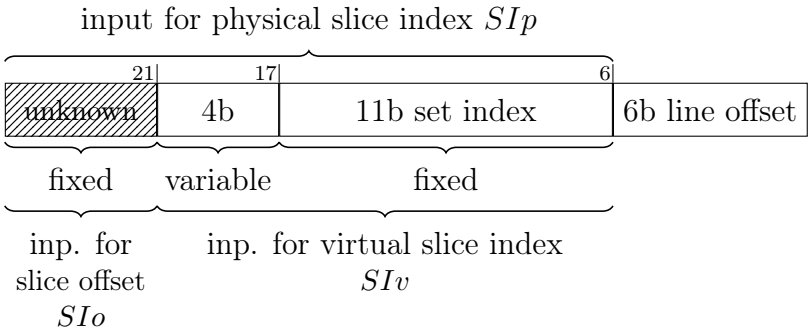


Figure 4.10: Inputs to the slice indexing hash function for 2MB pages.

calculate what we will call the *virtual slice index* SI_v for each address in a huge page. In a CPU with m slices, the slice function will be $o = ld(m)$ bits wide. Because the slice function consists merely of an XOR sum for each of its output bits, we can reduce all unknown address bits to an index of o unknown bits. As this unknown index is unchanging for the entire huge page, we can consider it a constant XOR offset SI_o which translates the relative virtual slice index of addresses in a page to their absolute physical slice indices: $SIp = SIv \oplus SIo$. The only problem with this equation is that we cannot find SIo without already knowing SIp. Luckily, we do not need to. After all, the only requirement is to know which addresses belong to the same cache set, not on which slice the set is located.

With this in mind, instead of trying to determine the absolute offset SIo for each 2MB page, we look only for a relative offset $SIo_{0,j}$ towards a reference page, such that for page j we can convert its virtual slice indices into indices of the reference page: $SIv_0 = SIv_j \oplus SIo_{0,j}$. Which page is chosen to be the reference page is of no consequence, as SIo isn't known for any page. How many 2MB pages we will need to align in this way depends on the number of addresses that share the same cache set index and slice index in each page as well as the number of addresses we need to evict a set. Figure 4.10 shows how bits are distributed with regards to the hash function. We can see that for each set index, there are only 4 bits in a 2MB page which we can vary. Assuming at least one of bits 17-20 is included in each part of the hash function (which is the case for Haswell), each page contains 2^{4-o} addresses in the same cache set. So, to construct an eviction set for an n-way cache, we need $p = \frac{n}{2^{4-o}}$ 2MB pages. According to this, for example, on our test system with a 16-way set associative cache using 4 slices, this method requires 4 2MB pages.

In practice, the relative offset can be found by testing for set eviction. This means an extra page is needed, as we need n addresses for eviction and at least 2 to probe. Listing 4.3 shows the entire process in pseudo code.

```
1  function setAddresses(page) begin
2   set_addresses ← {};
3   address ← start_address(page);
4   while (address < end_address(page)) do
5    set_addresses ← ∪ address;
6    address ← address + 4096;
7   end
8   return set_adresses;
9  end
10
11 function createInitialMapping(pages) begin
12  foreach page in pages do
13   foreach address in setAddresses(page) do
14    mapped_sets[page][slice(address)][set(address)] ← ∪ address;
15   end
16  end
17
18  return mapped_sets;
19 end
20
21 reference_page ← 0;
22 test_slice ← 0;
23 test_set ←  0;
24
25 mapped_sets ← createInitialMapping(pages);
26
27 // produces n*m addresses
28 test_eviction_set ← mapped_sets[pages\{reference_page}][*][test_set];
29
30 probe_set ← {2 of mapped_sets[reference_page][test_slice][test_set]};
31 eviction_set ← {};
32
33 slice_offset[number_of_pages] ← {0..};
34
35 foreach page do
36  foreach slice do
37   test_eviction_set  ← test_eviction_set\{addresses of page & slice};
38   count ← 0;
39   miss ← 0;
40   hit ← 0;
41   while count < number_of_tests do
42    if evicted(probe_set, test_eviction_set ∪ eviction_set)
43     miss ← miss + 1;
44    else
45     hit ← hit + 1;
46   end
47   if miss < hit then do
48    slice_offset[page] ← slice;
49    break;
```

```
50      end
51    end
52
53    correct_addresses ← mapped_sets[page][slice][test_set];
54    eviction_set ← eviction_set∪correct_addresses;
55  end
56
57  reorder mapped_sets according to slice_offset;
58
59  return mapped_sets;
```

Listing 4.3: Cache Set Finding Algorithm

First, the set addresses in the p pages are sorted into an array by their set- and virtual slice indices. The total number of cache set indices is here reduced by 6 bits, from 2048 to 32. This is done to avoid more than one set per 4kB page, as repeated accesses to addresses in the same 4kB page can cause the hardware prefetcher to load more addresses, thereby possibly interfering with the data transmission. The resulting array contains $32*p*m$ sets of $16/m$ addresses each.

Now cache set 0 of virtual slice 0 in page 0 is defined as the reference set, offsets of all other pages will be calculated in relation to it. Two of the addresses in this set form the *probe set*. Page, slice and set index 0 are chosen arbitrarily, but without loss of generality. Next, the *test eviction set* is formed from all addresses with cache set index 0 in the remaining pages. This set includes at least $m*n$ addresses, which enables it to evict cache set 0 on any slice, as it contains at least n addresses for each. The *eviction set* is initially empty; it will be used to store addresses that have been found to be on the same slice as those in the probe set.

The process for determining the slice offset for each page is quite simple. For every page, each slice is tested for equality to the reference slice. To this end, first, the slice's addresses are removed from the test eviction set. The remaining test eviction set is temporarily combined with the eviction set and together used to try to evict the probe set in the standard Prime+Probe fashion. As always, there is a chance of interference from noise in this operation, which is why it is repeated a large number of times. What makes this somewhat more challenging than the regular Prime+Probe scenario is that we try to decide not between non-eviction or complete eviction, but non-eviction and almost-eviction, as priming always uses at least $n - 16/m$ addresses. Additionally, the probing set will consist of a maximum of $16/m$ addresses. Both of these issues increase the chances for accidental eviction, depending on the number of slices m. So for $m = 16$, a wrong slice would try to evict 1 probe address with $n - 1$ prime addresses, which still has a high chance to succeed. While we have no hardware to test this on, we assume

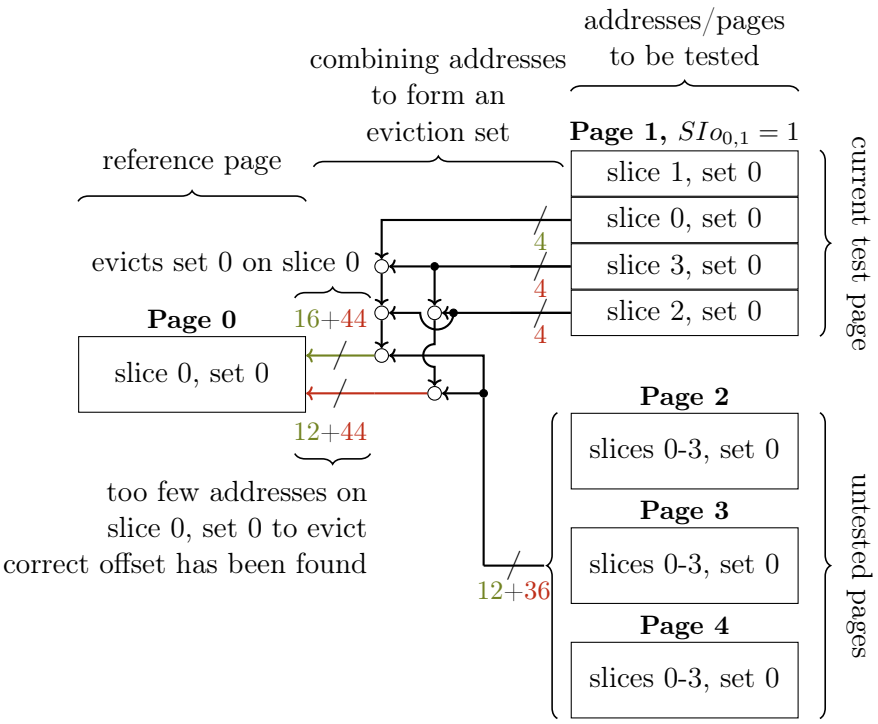


Figure 4.11: Testing a 2MB page of a quad core CPU such as the i7-4770 to find its relative offset. Depicted is the creation of 2 test eviction sets for the testing of slices 1 and 0 in page 1 and their subsequent eviction test against the reference set on page 0. The number of addresses on slice 0 is noted in green, other slices in red.

that for these reasons this method would not work reliably on a 16-core CPU and would have to be refined.

If, after all tests were run, the slice is determined not to be a match to the reference slice, the algorithm moves on to the next. When a match is found, the addresses of that slice are added to the eviction set, the slice number is stored as the offset for this page and the process starts again for the next page. After all pages have been processed, each has been assigned its relative offset $SIo_{0,j}$. With this, the initial sorting of the addresses can be adjusted to remove the pages and account for their offsets, such that what remains is an array of $32 * m$ cache sets of at least n addresses each. In our transmission scheme, each available cache set equals 1 bit of bandwidth. This means that we will have the required 26 cache sets (see Figure 4.9) even on dual-core CPUs, for which only 64 cache sets are produced. Figure 4.11 illustrates the testing of 2 slice offsets for a quad-core CPU. Some parameters of this algorithm for the hardware used in this thesis can be found in Table 4.2.

CPU	cache size	assoc	slices	addr. per set&page	pages req.
i7-4770k	12MB	16	4	4	4+1
i7-7770k	12MB	16	8	2	8+1
e5-2670	20MB	20	8	2	10+1

Table 4.2: A comparison of hardware and algorithm parameters for CPUs used in the evaluation of this channel.

4.4 Jamming Agreement

The last piece required to establish a channel across virtual machines is the ability to communicate some parameters from one machine to the other before starting regular transmission. This section will thus deal with overcoming challenge C4. Specifically, the initiating party needs to convey to the other which cache sets to use. As the relation of the slice mapping obtained in Section 4.3 is not fixed with respect to the hardware mapping, using predetermined sets is not an option.

Given the noisy nature of the cache, we were inspired by Boano et al. [46] to devise a protocol to transmit the used cache sets from a server to its client based on the jamming of cache sets. Where jamming in a wireless communications context refers to transmitting a carrier signal for a certain time, for our application it means evicting a set over and over. Unlike the regular cache channel transmission, where many sets are evicted in parallel, here we limit ourselves to one at a time. This allows us to communicate not only which sets to use, but also in which order.

For both server and client, this protocol is very simple. Both use the same 2 basic operations, *jamming* and *detection*. As Figure 4.12 illustrates, the server merely alternates between jamming and detection. The client meanwhile cycles through all possible sets in the detection phase, checking each for jamming from the server. When jamming is detected, the client saves the current set's index and proceeds to respond by jamming the same set. If the server detects this response, it moves on to the next set, otherwise it keeps jamming and detecting on the current set. To prevent the client and server detection phases from interfering with each other, the addresses for detection on the server and jamming on the client are offset by 64 bytes, which creates a parallel channel for acknowledgments.

Detection is done by counting the number of evictions in a sliding window that is smaller than the jamming period and the detection period. This allows even a partial overlap between the two to still be identified. If more than a certain threshold of the probes in the sliding window were evicted, it is considered a positive detection and the algorithm moves on. This threshold needs to be chosen such that it is extremely unlikely to be reached by random,

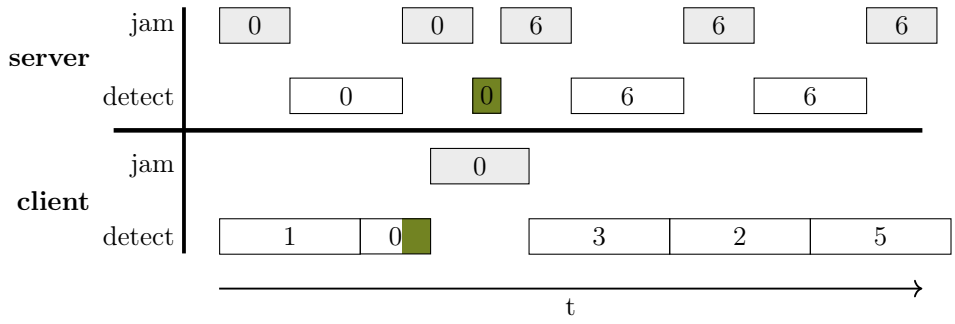


Figure 4.12: The first few jam/detect cycles of the jamming agreement. Sets are numbered in relation to the server's virtual slice index, hence the client begins reading at set 1 and continues with 0,3,2,5,4,7 etc. The green marker indicates a successful detection.

possibly heavy, noise, but will reliably be exceeded when the set is being jammed. Sometimes detection will still fail; since client and server both endlessly perform the same actions until all sets have been transmitted, this only means the process will take longer. Special care need only be taken for the very last set, as the client will end the jamming agreement on its side once the expected number of sets has been received. To make sure the server knows the set has been received, the last response-jamming by the client is significantly longer.

The speed and reliability of the process are highly dependent on the choice of timing for the basic operations, jamming/detection for server and client each $(t_{sj}, t_{sd}, t_{cj}, t_{cd})$. Several restrictions apply to these parameters:

- The jamming times t_{sj} and t_{cj} need to be at least long enough to reach the eviction threshold.
- The jamming time t_{cj} should be at least t_{sj} plus the time it takes to reach to reach the eviction threshold, so that the client ideally only has to confirm each set once.
- The detection time t_{cd} should be at least t_{sd} plus the time it takes to reach to reach the eviction threshold, to minimize the chance that the client misses a set transmission.

We will explore the practical realization of these rules in Section 5.4.

This section has also been expanded upon and presented at Blackhat Asia 2017 by Schwarz et al. [47].

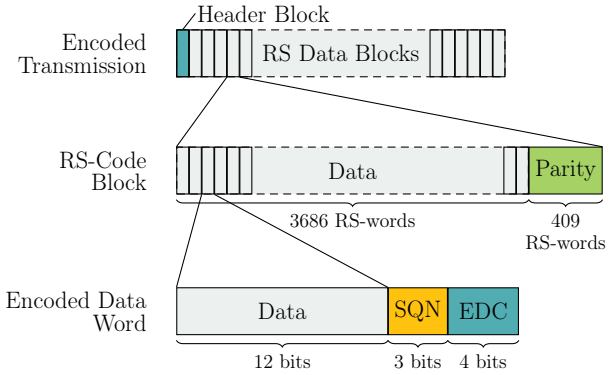


Figure 4.13: The structure of a transmission for 5% error correction capability [10].

4.5 Error Correction

To fulfill the premise of this thesis and show that a fast and robust cache covert channel is indeed practical, we add error correction to the channel.

In the previous sections, we have shown how to overcome all the challenges presented in Chapter 3. With the implementation of Section 4.4, our channel can now be established and communicate across virtual machines, producing a steady stream of data with substitution errors. We will use RS Codes for this task (Section 2.3.1) because of their good tolerance of burst errors and their easily available implementation.

In our implementation, the server starts by sending a header-block to the client, which includes the uncoded and encoded length of the data that will follow as well as a "magic number" that serves as a simple verification. The server encodes all the data, including the header, into blocks before the transmission. The client decodes the first block (the header) on the fly, but delays decoding of the data until all the blocks have been received, to avoid introducing additional points of desynchronization.

Chapter 5

Performance Evaluation

Now that we have implemented a functional, cross-core, cross-VM cache covert channel, we want to evaluate how well the channel as a whole, as well as its components separately, perform in different scenarios. For this we will use the testing setup for artificial cache noise as described in Section 4.1.1 and extend it up to `stress "-m 4"`. Together with the channel, this option simulates 8 threads of a CPU at full capacity. These tests represent extreme conditions for our channel and serve to test its limits. We run these tests on the hardware shown in Table 4.2, where the E5-2670 CPU is the one used in our Amazon EC2 instances. EC2 tests are performed on a dedicated host, both to reliably achieve co-location of our instances as well as to prevent contamination of our results by third party use of shared resources.

5.1 Transfer Speed

To begin this evaluation, we look at the transfer speed. We examine our channel under 5 artificial noise levels in a native environment on our hardware and in the cloud. We contrast this with a Flush+Reload channel using the same back-channel design, which will allow an estimate of the performance impacts of the back-channel and using Prime+Probe over Flush+Reload. As in Section 4.1.1, one series consists of 100 runs, each transmitting 512kB of (the same) randomly generated data without error correction. For this section, we do not consider channel failures before the start of data transmission.

A note on hyper-threading and CPU clock speed: All CPUs we test are hyper-threaded, and we expect most that are used by cloud computing providers are as well, so we need to test with this in mind. As 2 hyper-threads running on the same physical core will contend for some of its resources but not others,

the total performance is neither simply double that of 1 exclusive thread, nor is it the same. Instead, the performance gain will vary depending on the specific load. In the case of our channel, we find that when the 2 threads of the receiver or the sender respectively share a physical core, we see a drop in performance, but not usually to the point of failure. When some of the channel's threads share a physical core with a `stress` thread, however, we see a sharp increase in channel failures and error ratios. Based on what we have observed in our testing on the EC2, we make the assumption that cloud providers always assign whole physical cores to instances, when those instance feature more than one virtual core. Therefore, we do not consider the scenario where other applications share the same physical core as our channel. Instead, we separate our tests into two categories, hyper-threaded (sender/receiver share 1 physical core each) and non-hyper-threaded (each channel thread has a dedicated physical core). `Stress` is always executed on different physical cores.

Another factor to consider is Intel Turbo Boost. While the CPU is not used at full capacity, transmissions will usually benefit from a higher sustained clock speed. We can observe in our tests that without `stress`, the CPU often reaches its maximum turbo potential, while 1 or more threads of `stress` reduce this to a mostly stable, lower clock speed. The frequency noted in Table 5.1 is thus a lower bound and subject to fluctuations, though on our cloud instances, the frequency seems to be unchanging.

The resulting means for all series are shown in Table 5.1 and some are visualized as boxplots in Figure 5.1. A detailed view of the data that makes up the boxplot is given in Figure 5.2. From these tests we can gain several insights:

- Our Prime+Probe channel performs very well on the Amazon cloud, even under high stress. Both tests show a lower performance impact of noise when compared to the other CPUs. This might be attributable in part to more consistent clock speed, and in part to the larger cache associativity of the e5-2670.
- The speed on the EC2 hardware is reduced by approximately 30%-35% when using hyper-threading and the error ratio is increased.
- With our back-channel design, the error ratio does not increase markedly as the cache noise increases; instead, we see a drop in transfer speed.
- On our test system, Flush+Reload achieves significantly higher transfer speeds compared to Prime+Probe, especially under heavier noise. This is due to its much faster basic operations, which are not appreciably slowed down by higher cache noise. The error ratio, however, is much higher than Prime+Probe. We attribute this to the fact that our basic Flush+Reload implementation relies on a single address for each trans-

hw/test	stress	speed [kB/s]	bER [%]	BER [%]	WER [%]	fail [%]
i7-7700k, native F+R, w/ back-channel, hyper-threaded, 4.4GHz						
	0	237.2 ± 1.7	1.773	6.640	7.170	0
	1	224.6 ± 1.7	1.871	6.957	7.366	1
	2	212.5 ± 2.6	1.946	7.212	7.554	2
	3	203.1 ± 1.9	1.947	7.303	7.587	4
	4	193.9 ± 3.4	1.918	7.307	7.552	6
i7-7700k, native P+P, w/ back-channel, hyper-threaded, 4.4GHz						
	0	177.4 ± 5.2	0.302	1.730	1.790	3
	1	89.5 ± 6.9	0.425	2.457	2.559	1
	2	61.9 ± 8.5	0.461	2.659	2.772	0
	3	55.3 ± 8.9	0.445	2.558	2.664	0
	4	51.2 ± 6.7	0.441	2.528	2.633	5
i7-7700k, native P+P, w/ back-channel, not hyper-threaded, 4.4GHz						
	0	200.5 ± 4.5	0.155	0.823	0.800	4
i7-4770k, native P+P, w/ back-channel, hyper-threaded, 3.7GHz						
	0	107.8 ± 29.2	0.274	1.430	1.423	7
	1	52.0 ± 16.9	0.057	0.307	0.312	1
	2	57.1 ± 17.0	0.042	0.228	0.233	0
	3	48.2 ± 7.1	0.016	0.087	0.087	1
	4	42.8 ± 7.0	0.022	0.118	0.119	2
e5-2670, cross-vm P+P, w/ back-channel, not hyper-threaded, 2.6GHz						
	0	89.2 ± 0.8	0.045	0.216	0.212	0
	1	83.0 ± 1.3	0.044	0.223	0.220	0
	2	76.4 ± 1.7	0.046	0.239	0.238	0
	3	69.2 ± 2.2	0.050	0.266	0.266	0
	4	60.1 ± 5.4	0.065	0.345	0.347	0
e5-2670, cross-vm P+P, w/ back-channel, hyper-threaded, 2.6GHz						
	0	59.1 ± 1.8	0.088	0.397	0.397	6
	1	51.8 ± 1.9	0.134	0.595	0.586	12
	2	46.5 ± 1.0	0.167	0.742	0.729	25
	3	45.0 ± 1.1	0.170	0.763	0.750	26
	4	43.0 ± 3.9	0.162	0.729	0.718	12

Table 5.1: Average Speed and bit/byte/word error ratios for different channels and configurations. The failure rate represents transmissions that stopped before completion.

mission, which is much more likely to be randomly evicted by noise than all probe addresses of Prime+Probe. We conclude that overall, the back-channel design impacts the transfer speed more than using Prime+Probe over Flush+Reload, as the differences are much smaller than they are in a purely time-synced implementation (e.g., Section 4.1.1).

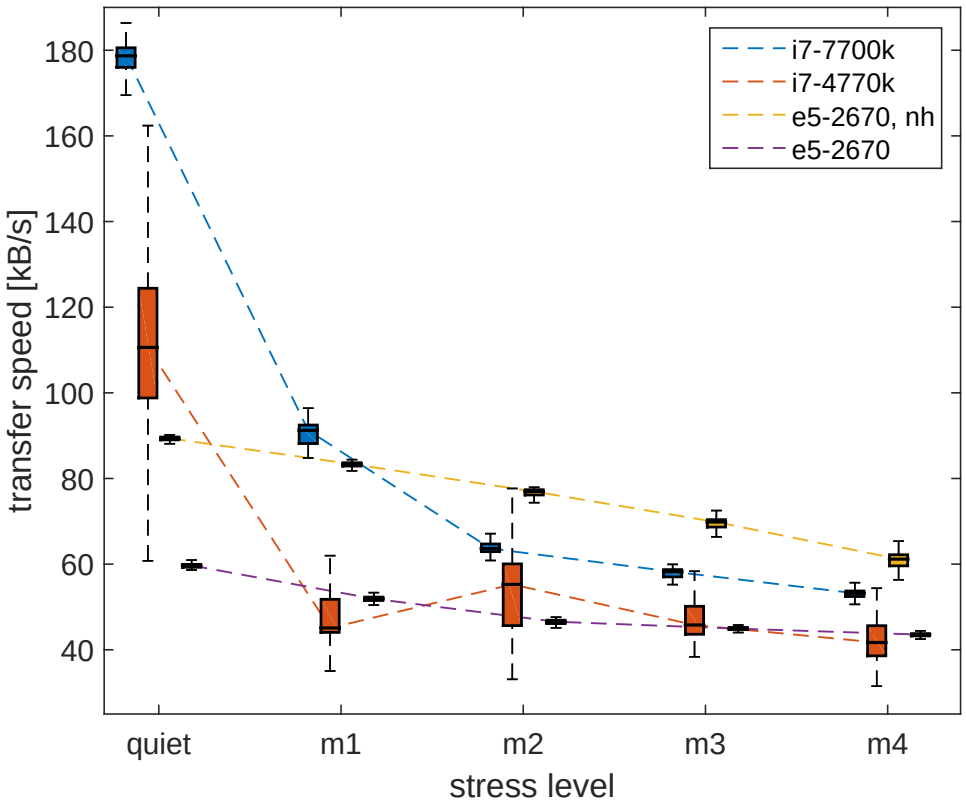


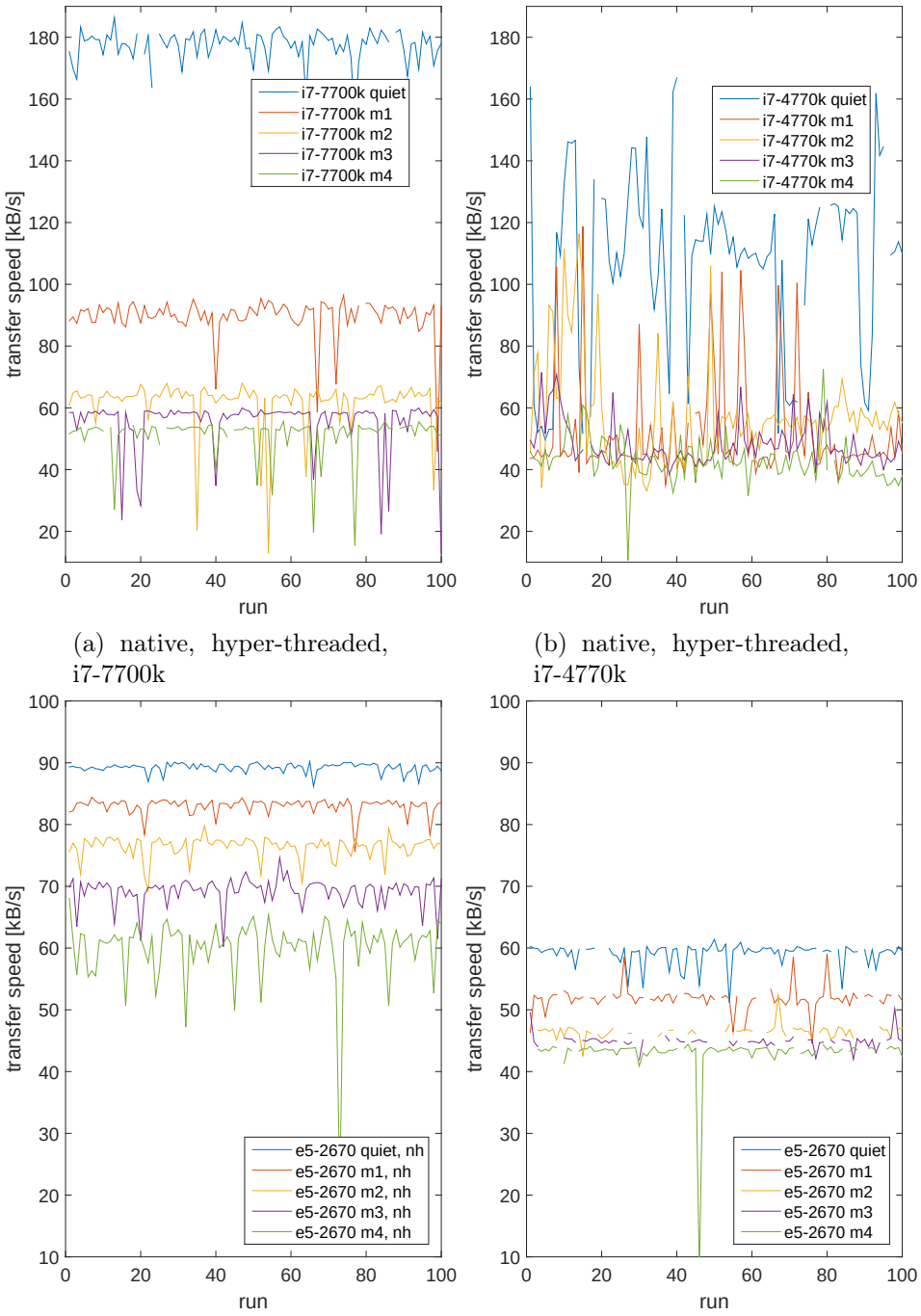
Figure 5.1: Boxplot of transfer speeds for 4 Prime+Probe scenarios as per Table 5.1. The boxes indicate 25th-75th percentile, inside of them the median is marked in black. The extending whiskers encompass data within $\pm 2.7\sigma$.

5.1.1 Dynamic Delay

We can again look to Figure 5.3 to see how the delay described in Section 4.2.4 influences transfer speeds. The delay starts at the default value of 1000 and immediately begins to adjust with the current read errors per second. The delay stays mostly stable until something changes; in this case, it might be a temporary change in CPU speed on some of the cores, a change in the stress program or some other process entirely.

5.2 Transmission Error Analysis

So far, we have seen the mean error ratio for bits, bytes, and words, as well as the failure rate. In this section, we examine these values more closely and try to understand the consequences with regards to a robust and reliable channel.



(a) native, hyper-threaded, i7-7700k

(b) native, hyper-threaded, i7-4770k

(c) cross-vm EC2, not hyper-threaded, e5-2670

(d) cross-vm EC2, hyper-threaded, e5-2670

Figure 5.2: Transfer speeds of the covert channel with back-channel for different hardware and scenarios. Missing datapoints show failed runs.

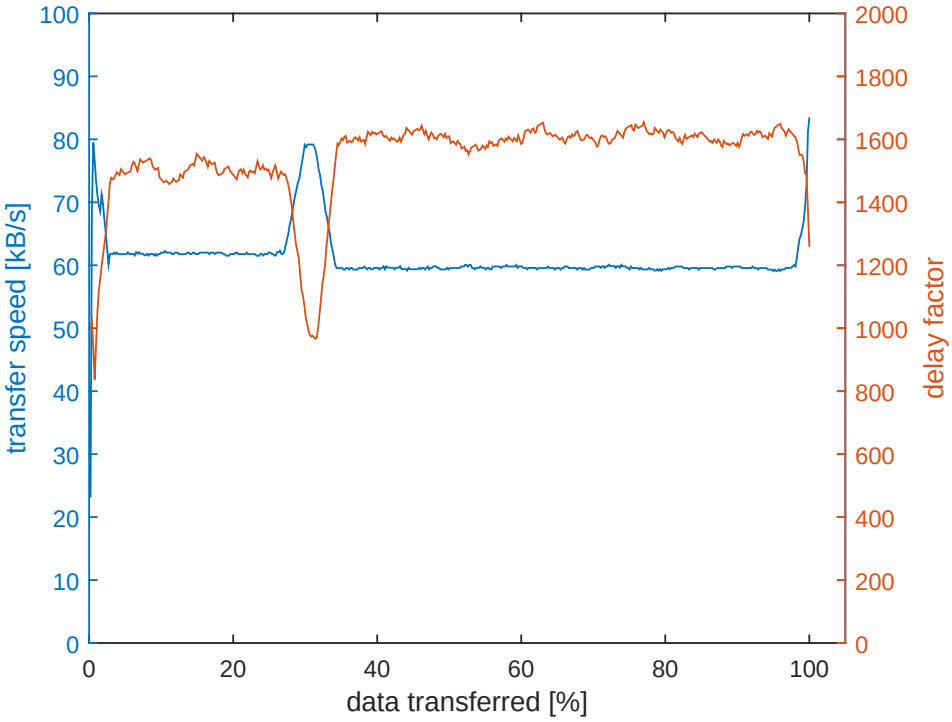


Figure 5.3: A run from the m2 series on the i7-7700k, showing the interaction of transfer speed and delay factor over time.

5.2.1 Transmission Failures

In our dataset, we classify a transmission as a failure (see Table 5.1) when detailed analysis shows a large number of errors in close proximity. Where in the transmission this happens is, in this case, irrelevant, because, for the purposes of analyzing errors, these runs are tainted. Transmissions with any kind of error during the setup of the channel were not recorded at all, as these are detected almost immediately and discussed in Sections 5.3 and 5.4. With the notable exception of hyper-threaded operation on the EC2, we see a fairly low failure rate that does not always appear to be correlated with noise.

Either receiver or sender should, in theory, be able to detect an aborted or desynchronized transmission. We can distinguish between 3 categories of errors: failure directly at the start of, during, or at the end of the transmission. Failures at the beginning can occur when the jamming agreement produced an error that neither party detected, and consequently, no data can be correctly transmitted. Failures at some point during the transmission occur when either sender or receiver cannot decode a word for a large number of reads. These are rare because the synchronization scheme should allow a

hw/test	quiet		m1		m2		m3		m4	
	fn	fp	fn	fp	fn	fp	fn	fp	fn	fp
i7-7700k	0	1/4	0	3/4	0	2/2	0	6/6	1	5/9
i7-4700k	0	0/7	0	0/1	0	0/0	0	0/1	0	0/2
e5-2670, nh	0	0	0	0/0	0	0/0	0	0/0	0	0/1
e5-2670	0	0/6	1	1/12	7	0/18	6	0/20	7	0/5

Table 5.2: Number of false negatives (fn) and false positives/detected (fp) for reported transmission failures.

continuation of the transfer with any offset, even if the correct synchronization is lost after the action. Finally, most failures are detected at the end of a transmission. When synchronization is lost, neither side can recognize this until the very end, when one of the two will stop receiving data or acknowledgments prematurely. This last case can sometimes produce false positives when the receiver has read the data in its entirety, but the sender, for some reason, detects a failure just before the end of a transmission. There can also be false negatives when synchronization is lost at some point during the transmission, but through some error at the end, both parties still accept the transfer as complete. Table 5.2 shows false negatives as well as false positives vs. the number of failures as reported by the sender or receiver. As both errors in failure reporting are similar in nature, we can conclude that there is a weakness in our implementation regarding the end of transmissions.

5.2.2 Word Error Ratio

As with the transfer speed, the mean error ratios in Table 5.1 are a good overview, but of course, they cannot represent the nature of the distribution. The boxplot in Figure 5.4 and the detailed views in Figure 5.5 show that, for some of the series, the distribution is almost bimodal. Nonetheless, for most series, the word error ratios are quite stable from run to run. For the most part, we still see a trend towards higher error ratios with higher noise, but it is not as clear as for the transfer speed. This can be explained by the self-slowing nature of our channel. The Berger code in conjunction with the delay parameter works to keep errors low, at the cost of speed.

5.2.3 Synchronization

The high failure- but low error ratio in some of the hyper-threaded EC2 series suggests that hyper-threading can manifest issues similar to descheduling and that synchronization for this is not perfectly solved, so we will evaluate our related measures here.

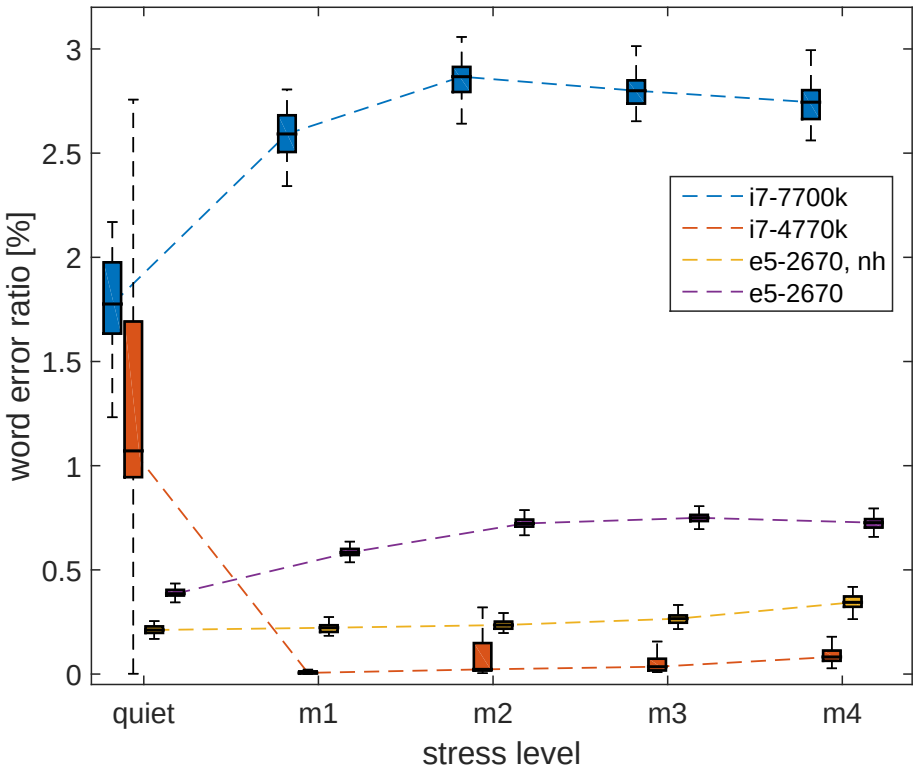
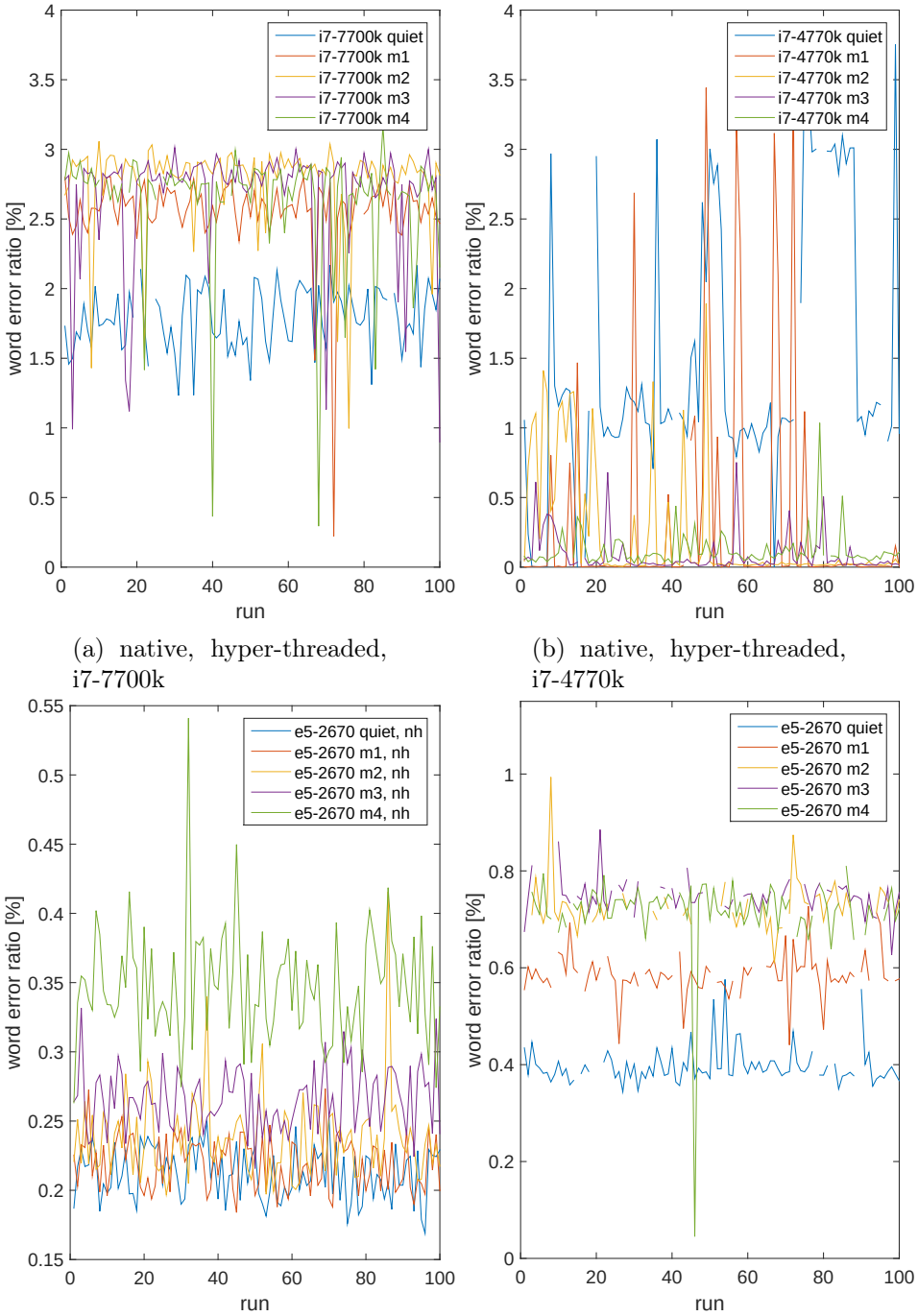


Figure 5.4: Boxplot for word error ratios of 4 Prime+Probe scenarios as per Table 5.1. The boxes indicate 25th-75th percentile, inside of them the median is marked in black. The extending whiskers encompass data within $\pm 2.7\sigma$.

Besides adjusting our existing measures, it would be possible to implement periodic synchronization checks on top of our channel to recover in the event of an error. One option for this would be to use the RS-code's ability to detect unrepairable blocks and add a retransmission protocol, which would also serve to resynchronize sender and receiver. This, however, is out of scope for this thesis.

n-Strike-Skip

All the series in Table 5.1 were recorded for 3-strike-skip. To see if n-strike-skip is even necessary, we recorded the number of skips and the number of skips that were prevented. As Table 5.3 shows, in most series the receiver only performs a few skips per transmission and prevents 100 or 1000 times more with the 3-strike-skip, while the sender shows low figures for both. This pattern is inverted on the sender side for the problematic hyper-threaded EC series. There are a few possibilities for this: If these skips do what they are



(a) native, hyper-threaded, i7-7700k

(b) native, hyper-threaded, i7-4770k

(c) cross-vm EC2, not hyper-threaded, e5-2670

(d) cross-vm EC2, hyper-threaded, e5-2670

Figure 5.5: Word error ratios of the covert channel with back-channel for different hardware and scenarios. Missing datapoints show failed runs.

hw/test	receiver		sender	
	skips	**skips prev.**	**skips**	**skips prev.**
i7-7700k	1.26	1829.52	2.80	7.84
i7-4700k	2.27	211.10	1.21	12.94
e5-2670, nh	0.12	109.97	3.06	0.90
e5-2670	2.62	653.12	196.67	9.44

Table 5.3: Mean number of words per transmission that were skipped and skips that were prevented by n-strike-skip.

#strikes:	**1 [%]**	**2 [%]**	**3 [%]**	**4 [%]**
sender	98.8713	0.5141	0.3129	0.3017
receiver	99.8527	0.1467	0.0003	0.0003

Table 5.4: Distribution of number of strikes to prevent skips in all hyper-threaded EC2 series.

supposed to, the fault lies with the receiver for incorrectly accepting a packet a second time as the following packet. The skips on the sender might also be reactions to skips on the receiver, produced in error, similar to the first case. If this were true, we would see skips occurring in pairs on the sender and receiver side, but we only see a total of 5 such pairs for all 5 series on the EC2. Finally, the sender could make the skips in error. Because of the asymmetry of sender and receiver, an accidental skip for the sender would mean a deadlock that can only be resolved by a backtrack (or more errors). Since we see very few backtracks, the vast majority of these skips must be legitimate.

We now run a new series for $n = 5$ on the EC2 to measure how many strikes it takes before an attempt to skip is stopped. In total, 98.87% of prevented skips occurred after only 1 strike on the sender and 99.85% on the receiver (see Table 5.4). Given these numbers, we feel confident that n-strike-skip does what it is intended to do and $n = 3$ is indeed an appropriate choice for the receiver. For the sender, the fact that 0.3% of skip preventions occurred at 4 strikes indicates that there might have been a few unnecessary skips that were not prevented. Since this would lead to a temporary deadlock, as discussed above, for the server $n = 6$ might be appropriate in this hardware scenario.

Backtracking

Table 5.5 contains the average number of backtracks per series. Overall, we can see that the number of backtracks is very low, even compared to skips. This is somewhat expected because backtracks are corrective measures that

hw/test:	i7-7700k	i7-4700k	e5-2670, nh	e5-2670
sender	0.216	0.054	0.002	0.162
receiver	0.018	0.008	0.000	0.378

Table 5.5: Mean number of backtracks per transmission.

are only employed after several errors have occurred in a row. We can confirm that backtracking is a useful tool in general by determining the number of backtracks that are directly involved in a desynchronization. We do this by looking at the word error ratio in the data before and after the backtrack. Over all series in the 4 major hardware setups we have recorded, out of 84 backtracks on the sender side, 11 were not involved in desynchronization, 58 occurred exactly at the point of desynchronization and 15 are inconclusive, because they happened after desynchronization. On the receiver side, out of 195 backtracks, 98 were unrelated to desynchronizations, 75 were related and 22 inconclusive. From this, we can say that at least 109 deadlocks were successfully resolved by backtracking, but at first glance we might also think that up to 133 backtracks caused desynchronization. However, we also recorded the result of the "voting" system described in Section 4.2.3. We chose a high barrier of 200 votes as a minimum for backtracking, and we see that in every case the result was close to 200:0, which means that it is *extremely* unlikely any of the backtracks were directly caused by chance, and thus were the cause of desynchronization as opposed to being only a symptom.

As with the skips, the hyper-threaded EC2 series stands out for having by far the most backtracks on the receiver and the second most on the sender. Of the backtracks mentioned above, which coincide with desynchronizations, all but 4 occurred during these 5 series. Given the successful backtracks, we believe the feature to be working in principle, albeit in need of some tuning for certain environments. For example, the minimum of 200 reads could be too high, keeping the channel in a deadlock for longer than necessary and increasing the chances of a different desynchronizing error appearing in the meantime.

Regarding hyper-threading, however, we want to mention that in a scenario where an attacker is trying to exfiltrate data from a virtual machine, the receiver VM is completely controlled by the attacker. This means that on instances with more than 2 virtual cores, the receiver threads need not be confined to only one physical core and would perform much closer to the non-hyper-threaded setup.

hw/test	6 bit	7 bit	8 bit	9 bit	10 bit	11 bit	12 bit
i7-7700k	40.0	31.0	22.5	18.5	16.5	13.5	8.5
i7-4700k	49.5	29.5	24.0	17.5	15.0	14.0	9.5
e5-2670, nh	30.5	20.0	17.0	14.0	9.0	6.0	3.0
e5-2670	30.5	20.0	12.5	8.5	7.0	5.5	3.0

Table 5.6: Minimum RS-Code parity percentage for completely error-free transmission for different symbol sizes.

5.2.4 The RS-Code and Bit Errors

We initially chose a symbol size of 12 bits for the RS-Code because this matches the size of our decoded data words. Because an error in a word breaks the symbol it is contained in, symbols that are larger than data words unduly increase the symbol error ratio in a block. However, a data word of 12 bits can have any number of bit substitutions, including only 1. To examine the actual distribution of the number of bit errors in erroneous words, we analyze data words from all Prime+Probe series, excluding only those transmissions where desynchronization caused high error ratios. The results (see Figure 5.6) show that 89% of words contain only 1 or 2 wrong bits. Given these results, it is worth examining if symbol sizes of less than 12 bits could lead to a lower symbol error ratio. An obvious candidate would be 6 bits, as the worst case is the same symbol error ratio as before, but the best case promises a halving of the error ratio. At lower symbol sizes, the RS-Code produces much smaller blocks, and so the risk of burst errors compromising entire blocks is increased. We can first calculate the overall symbol error ratios for symbol sizes from 6 to 12 bits to see if any might be preferable to 12 bits. As a dataset we use all 5 hyper-threaded series on the EC2 and filter out any transmissions that failed. Figure 5.7 confirms that a 6 bit symbol size provides the lowest symbol error ratio over the entire dataset. But since we are interested in how many errors we can correct, we also need to look at the number of blocks the RS-Code cannot correct for a given symbol size and parity percentage. Illustrated in Figure 5.8, this analysis reveals that the effect of multiple errors in a row is indeed too much for symbol sizes other than 12 bits. Naturally, the ultimate goal for our channel is a completely error-free transmission. Table 5.6 shows the minimum amount of error correction necessary to achieve 0 errors on the given channel/hardware setup, had the data been transmitted with a certain symbol size. The results are clear: any symbol size other than 12 results in a large performance penalty.

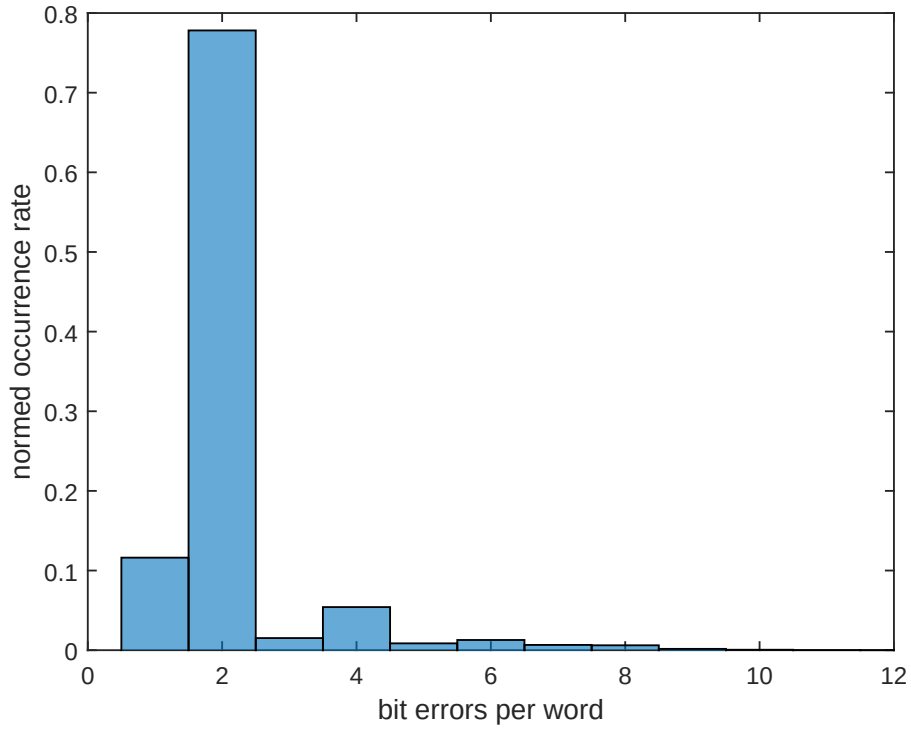


Figure 5.6: Distribution of number of bit errors in erroneous words over all Prime+Probe test series.

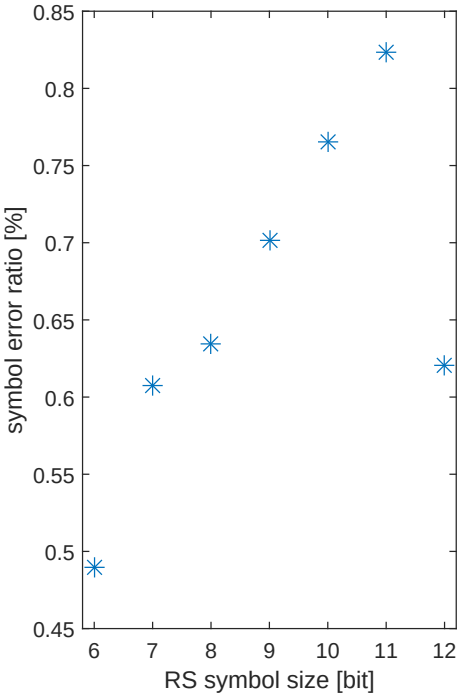


Figure 5.7: Symbol error ratio for different symbol sizes over all hyper-threaded series on the e5-2670 CPU (after removing failed transmissions).

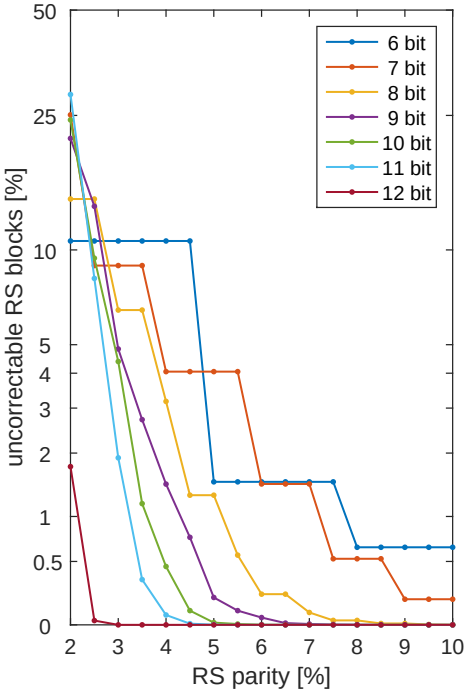


Figure 5.8: Rate of uncorrectable blocks for different symbol sizes and parity percentages over all hyper-threaded series on the e5-2670 CPU (after removing failed transmissions).

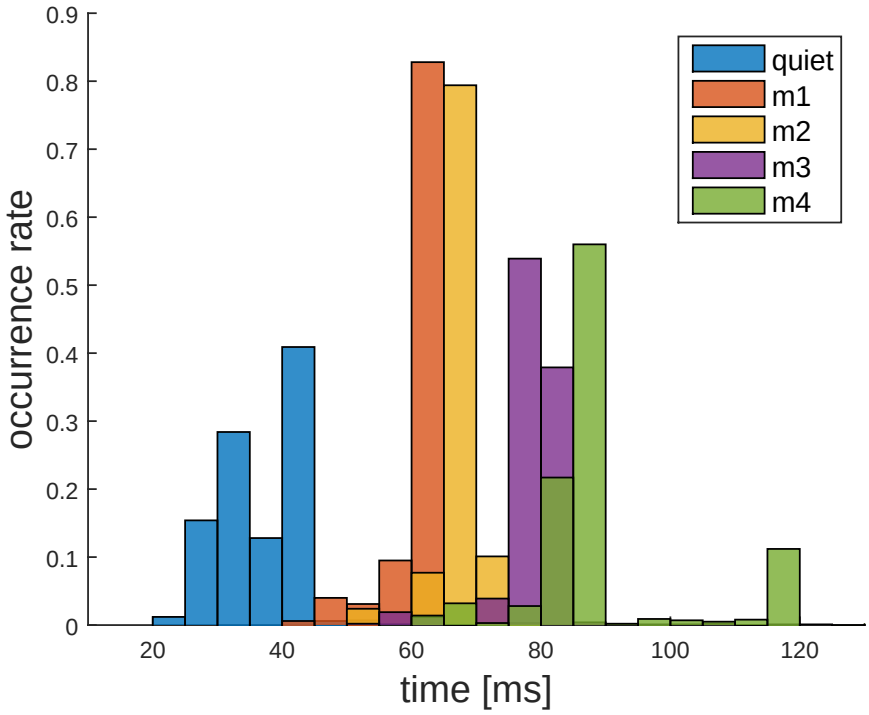


Figure 5.9: Distribution of run times for the cache-set-finding algorithm with different levels of stress for 1000 runs each. Different color tones show overlapping ranges.

stress level	**quiet**	**m1**	**m2**	**m3**	**m4**
mean [ms]	37.0	60.3	66.8	78.3	88.2
std. dev. [ms]	6.3	3.8	3.3	4.8	12.1

Table 5.7: Run times for the cache-set-finding algorithm with different levels of stress.

5.3 Cache Set Finding

To measure the speed of the cache-set-finding algorithm for finding 26 sets, as described in Section 4.3, we time 1000 runs for 5 different `stress` levels, from none to 4 threads, on our i7-4700k CPU. We can see in Figure 5.9 that the distribution of run times correlates with the increase in noise on the cache produced by the `stress` tool. This increase is caused by a higher rate of eviction of the probing addresses and thus larger read times. From the average run times in Table 5.7 we can conclude that incorporating the reverse-engineered hash function indeed produces a much faster method for finding addresses in the same cache set than previous works [13], which makes it practical for our channel. Testing the created sets for correctness, we see

no errors during these test series. It should, however, be mentioned that errors can still happen, and when they do the channel will experience failure or a wrong result during the Jamming Agreement.

5.4 Jamming Agreement

We empirically find parameters that satisfy the rules mentioned in Section 4.4 to evaluate the Jamming Agreement. The challenge, in practice, is that the parameters $t_{sj}, t_{sd}, t_{cj}, t_{cd}$ are not actually implemented as times, but rather number of reads and evictions $n_{sj}, n_{sd}, n_{cj}, n_{cd}$. Consequently, the timings vary heavily with all activity in the relevant cache sets, including noise *and* jamming/detection from the other party. Jamming is also affected much more by high noise on the cache than detection, since there are more addresses involved in eviction. As a result, the ideal parameters change not only with the hardware, but also the current level of noise. The goal of this section is to find parameters that perform well under most circumstances.

To measure how well the selected parameters are performing, we can look at the loop count and repeat count (which in turn determine the run time) as well as failure rate. The loop count measures how many times the client started from set 0 again, after testing all available sets. The repeat count shows how often a set was received by the client, but the server did not detect its acknowledgment. The loop count indicates how well the detection count n_{cd} was chosen in relation to n_{sd} and n_{sj}, a high loop count indicates that n_{cd} (or possibly also n_{sj}) is too low. The repeat count gives feedback on the the relation between n_{cj} and n_{sj} and n_{sd}.

During first testing, it becomes immediately obvious that the total run time is dominated by n_{cd} and the number of loops. Additionally, we can see that the number of loops heavily depends on the virtual slice offset between server and client in a given transmission. Figure 5.12a shows the clear difference for a given test series of 100 runs. This is a result of the client's loop structure described in Section 4.4 and the way we order the sets first by slice, then index — when the server transmits the same set index on several slices, a virtual offset other than 0 can force loops, because the sets are in a different order on the client. We introduce a simple optimization by resetting the set counter on the client to the most recently read set on slice 0 after a positive detection. E.g., on a CPU with 4 slices, detecting jamming on set number 6 would reset the counter to set 4. This has the additional benefit that failure to detect the acknowledgment on the server-side does also not force another loop, as each successfully detected set is read again within the next m detection phases. Figure 5.12b shows the clear improvements of this optimization.

series	n_{sj}	n_{sd}	n_{cj}	n_{cd}
1, 1o	300	500	1500	1200
2, 2o	300	500	3000	1200

Table 5.8: Parameters for all 4 test series.

In total, we measure 2000 set transmissions, composed of 4 different test series, each of which is run 100 times at 5 `stress` levels (quiet to m4). In each of these tests, we transfer 26 sets, the amount we need for the channel. For this we use 2 sets of parameters $n_{sj}, n_{sd}, n_{cj}, n_{cd}$ and run each series with and without the previously described optimization (series 1/2 vs. 1o/2o). As we can see in Table 5.8, we only varied n_{cj} between series 1/1o and 2/2o, as we found the other parameters to be in a good balance to each other. n_{cj} is to some extent independently variable, as a value that is too low does not influence the rate of detection directly and only causes a higher number of repeats.

Comparing the mean[1] run time and loop count plots in Figure 5.13 illustrates how tightly the two variables are linked. We can also observe how drastically the run times drop from series 1/2 to their optimized counterparts. The difference in the parameter n_{cj} between series 1/1o and 2/2o is less pronounced in the run time, but can be seen quite well in Figure 5.11. The higher parameter n_{cj} leads to fewer repeated readings of the same set in the client, which in turn leads to a smaller chance of missing the transmission and causing loops, which significantly affects the run time. Regarding the count of repeated sets, it is also noteworthy that the `stress` level m3 appears to cause significantly more problems than any other, including m4, for all 4 test series. Finally, the amount of failures, seen in Figure 5.10, shows us that the parameters were well chosen, as only 9 transmission failures in 2000 runs.

The result for series 2o is an average runtime of 45.2ms±37.4ms over all noise levels, and 18.2ms±6.9ms for the minimum-noise scenario. The entire series combined is shown in Figure 5.14.

[1] means were calculated taking into account the slice offset, as the distribution was not always uniform in our tests, but can, in general, be expected to be unbiased.

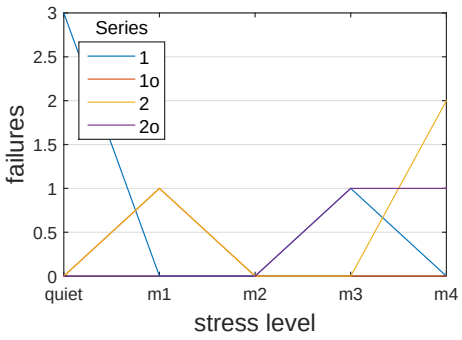


Figure 5.10: Number of failures per series and stress level.

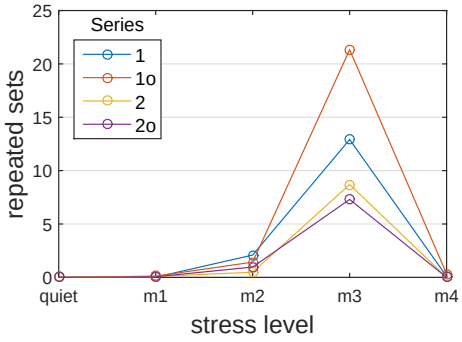
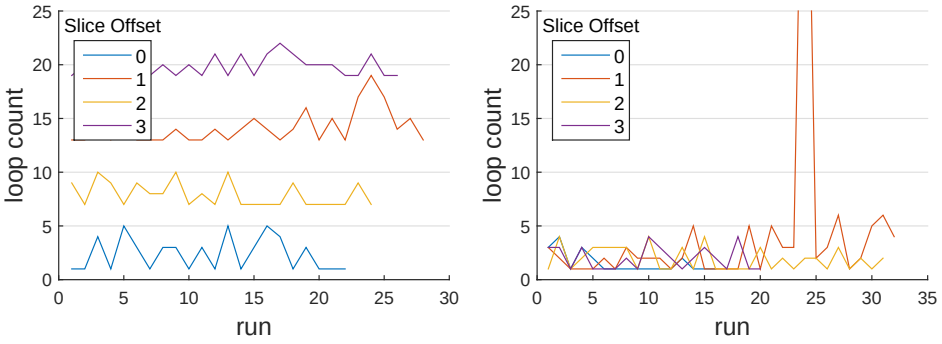


Figure 5.11: Mean number of repeated sets per series and stress level.



(a) without optimization

(b) with optimization

Figure 5.12: Mean number of loops by virtual slice offset before (a) and after (b) optimization. Series 1/1o, stress m2.

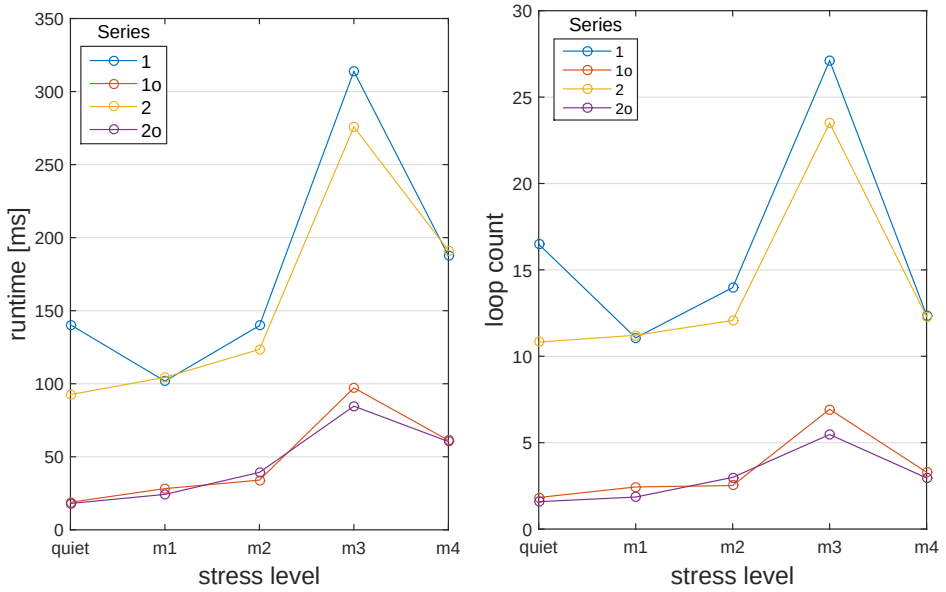


Figure 5.13: Mean runtime and mean loop count for each series and stress level.

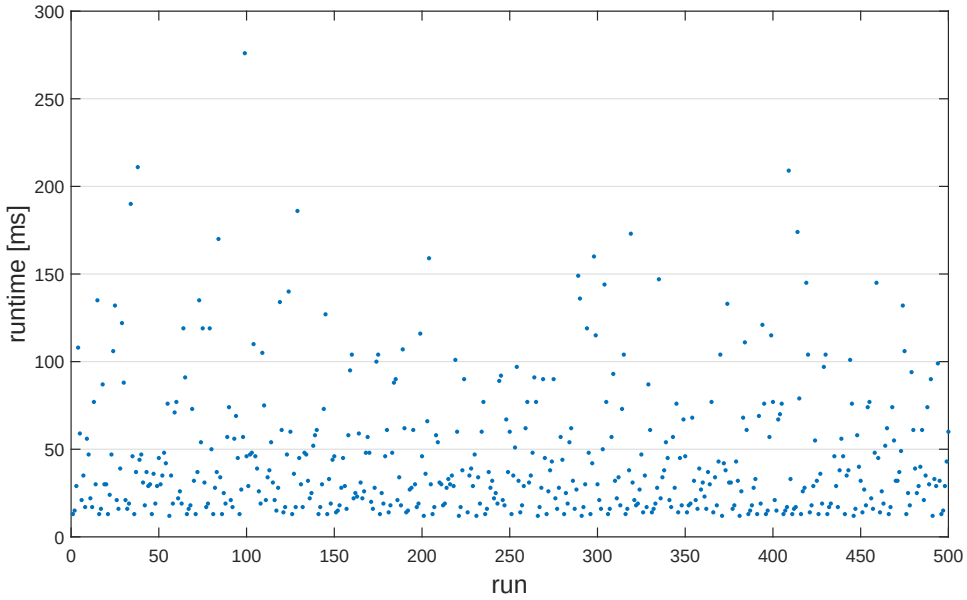


Figure 5.14: Combined run times for series 2o.

Chapter 6

Conclusion

When we started this thesis, there was already some literature declaring cache covert channels practical (as well as impractical). Upon review, we concluded that previous works were either not general enough, not fast enough or, in fact, not practical enough because they stopped short of achieving error-free transmissions. Thus, we set for ourselves the goals of achieving a robust, high-speed channel that works on a wide variety of CPUs with very few practical constraints.

In this thesis, we identified 4 principal challenges toward our goal and solved them by building and improving upon previous works as well as employing concepts novel to cache covert channels. With our jamming agreement, we created a way to share the most fundamental parameters of the channel ad hoc, while serving as a mechanism of initial synchronization at the same time. Our back-channel scheme allows us to be completely ignorant of large-scale errors and discontinuities in the `rdtsc` instruction, as long as small stretches of fewer than 300 instructions are mostly accurate. At the same time, the design around the Berger code lets our channel automatically scale its transfer speed to the level of cache noise and keep errors to a minimum. With this, we have created a building block that can easily be used as the basis for higher-level protocols such as TCP [10], which expect error-free transmission on the physical layer.

We have engineered a channel that can transmit between 41kB/s and 86kB/s with a byte error-rate of zero (including only 3% error correction) on the oldest generation of CPUs in the Amazon Elastic Compute Cloud at a clock speed of 2.6GHz and up to 183kB/s on newer hardware. Beyond the typical example of stealing credit card information, this is enough bandwidth to theoretically transfer gigabytes of data per day.

While there are many possible improvements to our channel, we believe to have shown conclusively that cache covert channels in the cloud can be fast as well as reliable, and are practical *now*.

Bibliography

[1] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *CCS*. 2009 (cit. on pp. 3, 4, 12).

[2] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. "Homealone: Co-residency detection in the cloud via side-channel analysis". In: *S&P*. 2011 (cit. on p. 3).

[3] Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., and Butler, K. "Detecting co-residency with active traffic analysis techniques". In: *CCSW*. 2012 (cit. on p. 3).

[4] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. "Cross-tenant side-channel attacks in PaaS clouds". In: *CCS*. 2014 (cit. on p. 3).

[5] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. "A Placement Vulnerability Study in Multi-Tenant Public Clouds". In: *USENIX Security Symposium*. 2015 (cit. on p. 3).

[6] Xu, Z., Wang, H., and Wu, Z. "A Measurement Study on Co-residence Threat inside the Cloud". In: *USENIX Security Symposium*. 2015 (cit. on p. 3).

[7] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. "Cache attacks enable bulk key recovery on the cloud". In: *CHES*. 2016 (cit. on pp. 3, 8, 9).

[8] Gruss, D., Spreitzer, R., and Mangard, S. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *USENIX Security Symposium*. 2015 (cit. on pp. 3, 11).

[9] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *DIMVA*. 2016 (cit. on pp. 3, 9, 11, 12).

[10] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C. A., Römer, K., and Mangard, S. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *NDSS*. 2017 (cit. on pp. 4, 28, 34, 42, 62).

[11] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. "An exploration of L2 cache covert channels in virtualized environments". In: *CCSW*. 2011 (cit. on pp. 4, 12).

[12] Maurice, C., Neumann, C., Heen, O., and Francillon, A. "C5: Cross-Cores Cache Covert Channel". In: *DIMVA*. 2015 (cit. on pp. 4, 12).

[13] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. "Last-Level Cache Side-Channel Attacks are Practical". In: *S&P*. 2015 (cit. on pp. 4, 12, 35, 57).

[14] Martineau, E. *The art of cache timing covert channel on x86 multi core*. `https://www.youtube.com/watch?v=7X772EBdvnM`. 2015 (cit. on pp. 4, 12).

[15] *Amazon EC2 Instance Types*. `https://aws.amazon.com/de/ec2/instance-types`. Retrieved on April 05, 2017. 2017 (cit. on p. 6).

[16] Levinthal, D. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`. Retrieved on March 21, 2017. 2009 (cit. on p. 7).

[17] *Intel Ivy Bridge Cache Replacement Policy*. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`. Retrieved on January 29, 2020. 2013 (cit. on p. 7).

[18] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. "Adaptive insertion policies for high performance caching". In: *ACM SIGARCH Computer Architecture News* 35.2 (2007), pp. 381–391 (cit. on p. 7).

[19] Jaleel, A., Theobald, K. B., Steely Jr, S. C., and Emer, J. "High performance cache replacement using re-reference interval prediction (RRIP)". In: *ACM SIGARCH Computer Architecture News* 38.3 (2010), pp. 60–71 (cit. on p. 7).

[20] Hennessy, J. and Patterson, D. *Computer Architecture, 5th Edition*. Morgan Kaufmann, 2011. ISBN: 9780123838728 (cit. on p. 8).

[21] Shilov, A. *More AMD 'Zen' CPU details emerge*. `http://www.kitguru.net/components/cpu/anton-shilov/more-amd-zen-cpu-details-emerge-quad-core-units-inclusive-cache-high-speed-interconnects/`. Retrieved on March 22, 2017. 2015 (cit. on p. 8).

[22] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2016 (cit. on p. 8).

[23] Hund, R., Willems, C., and Holz, T. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *S&P*. 2013 (cit. on pp. 8, 9).

[24] Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. “Reverse Engineering Intel Complex Addressing Using Performance Counters”. In: *RAID*. 2015 (cit. on pp. 8, 9, 20, 35).

[25] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud”. In: *Cryptology ePrint Archive* 2018.898 (2015) (cit. on p. 8).

[26] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. “Mapping the Intel Last-Level Cache”. In: *Cryptology ePrint Archive* 2015.905 (2015) (cit. on p. 8).

[27] Hu, W.-M. “Lattice scheduling and covert channels”. In: *S&P*. 1992 (cit. on p. 9).

[28] Page, D. “Theoretical use of cache memory as a cryptanalytic side-channel”. In: *Cryptology ePrint Archive* 2002.169 (2002) (cit. on p. 9).

[29] Bernstein, D. J. “Cache-timing attacks on AES”. In: (2005) (cit. on p. 9).

[30] Percival, C. “Cache missing for fun and profit”. In: *BSDCan*. 2005 (cit. on pp. 9, 10, 12).

[31] Osvik, D. A., Shamir, A., and Tromer, E. “Cache attacks and countermeasures: the case of AES”. In: *CT-RSA*. 2006 (cit. on pp. 9, 10).

[32] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. “Cross-VM side channels and their use to extract private keys”. In: *CCS*. 2012 (cit. on p. 9).

[33] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. “Wait a minute! A fast, Cross-VM attack on AES”. In: *RAID*. 2014 (cit. on p. 9).

[34] Benger, N., Van de Pol, J., Smart, N. P., and Yarom, Y. ““Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way”. In: *CHES*. 2014 (cit. on p. 9).

[35] Yarom, Y. and Falkner, K. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014 (cit. on pp. 9, 10).

[36] Irazoqui, G., Eisenbarth, T., and Sunar, B. “S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES”. In: *S&P*. 2015 (cit. on p. 9).

[37] Gulmezoglu, B., Eisenbarth, T., and Sunar, B. “Cache-based application detection in the cloud using machine learning”. In: *AsiaCCS*. 2017 (cit. on p. 9).

[38] Gruss, D., Maurice, C., and Mangard, S. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA*. 2016 (cit. on pp. 10, 22).

[39] Gullasch, D., Bangerter, E., and Krenn, S. “Cache games–bringing access-based cache attacks on AES to practice”. In: *S&P*. 2011 (cit. on p. 10).

[40] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. “ARMageddon: Last-Level Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016 (cit. on p. 11).

[41] Berger, J. M. “A note on error detection codes for asymmetric channels”. In: *Information and Control* 4.1 (1961), pp. 68–73 (cit. on p. 14).

[42] Reed, I. and Solomon, G. “Polynomial codes over certain finite fields”. In: *MIT Lincoln Laboratory Group Report* 47 (1958), pp. 22–31 (cit. on p. 15).

[43] Roman, S. *Coding and information theory*. Vol. 134. Springer Science & Business Media, 1992 (cit. on p. 15).

[44] *stress, a workload generator for POSIX systems*. `https://people.seas.harvard.edu/~apw/stress/`. Retrieved on April 30, 2017. 2017 (cit. on p. 23).

[45] Hadamard, J. “Resolution d’une question relative aux determinants”. In: *Bull. des sciences math.* 2 (1893), pp. 240–246 (cit. on p. 33).

[46] Boano, C. A., Zuniga, M. A., Römer, K., and Voigt, T. “Jag: Reliable and predictable wireless agreement under external radio interference”. In: *RTSS*. 2012 (cit. on p. 40).

[47] Schwarz, M. and Weber, M. “CJAG: Cache-based Jamming Agreement”. In: *Blackhat Asia*. 2017 (cit. on p. 41).