Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud

Clémentine Maurice*, Manuel Weber*, Michael Schwarz*, Lukas Giner*,
Daniel Gruss*†, Carlo Alberto Boano*, Kay Römer*, Stefan Mangard*
*Graz University of Technology, † Microsoft Research

***Abstract*—Covert channels evade isolation mechanisms between multiple parties in the cloud. Especially cache covert channels allow the transmission of several hundred kilobits per second between unprivileged user programs in separate virtual machines. However, caches are small and shared and thus cache-based communication is susceptible to noise from any system activity and interrupts. The feasibility of a reliable cache covert channel under a severe noise scenario has not been demonstrated yet. Instead, previous work relies on either of the two contradicting assumptions: the assumption of direct applicability of error-correcting codes, or the assumption that noise effectively prevents covert channels.**

In this paper, we show that both assumptions are wrong. First, error-correcting codes cannot be applied directly, due to the noise characteristics. Second, even with extraordinarily high system activity, we demonstrate an error-free and high-throughput covert channel. We provide the first comprehensive characterization of noise on cache covert channels due to cache activity and interrupts. We build the first *robust* covert channel based on established techniques from wireless transmission protocols, adapted for our use in microarchitectural attacks. Our error-correcting and error-handling high-throughput covert channel can sustain transmission rates of more than 45 KBps on Amazon EC2, which is 3 orders of magnitude higher than previous covert channels demonstrated on Amazon EC2. Our robust and error-free channel even allows us to build an SSH connection between two virtual machines, where all existing covert channels fail.

I. Introduction

With the advent of cloud computing and virtualization, CPU caches have been largely studied in terms of covert channels. Covert channels are unauthorized communication channels between two parties, a sender and a receiver. The basis for cache covert channels is the difference in latency for memory accesses, depending on whether data is cached or not. Caches are well-suited for covert channels in virtualized environments, as they are not virtualized, and are thus shared across virtual machines of a same physical machine. Moreover, caches are a fast type of memory, shared across the cores of a CPU, and coherent between CPUs of the same machine. Therefore, state-of-the-art attacks have moved from same-core to cross-core and even cross-CPU covert channels.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23294

However, caches are currently quite small due to the price of SRAM and its physical footprint on dies. They are a few megabytes at most, compared to gigabytes of main memory. Caches are also shared between processes and between tenants in virtualized environments. Cache-based communication is therefore susceptible to noise from any system activity and interrupts. Indeed, data gets unintentionally evicted by other programs, e.g., memory intensive programs, the operating system, and the hypervisor. In addition to this, operating system and hypervisor scheduling is also a source of noise, as sender and receiver are not necessarily scheduled at the same time. Noise causes substitution errors as well as synchronization errors such as inserted and deleted bits in cache covert channels. Noise is a well-known problem in cache attacks, but it has not been studied in more depth than the observation of an increased error rate. Yet, some applications of covert channels require error-free communication, e.g., an SSH connection between the two communicating parties.

Noisy communication channels are not a unique issue of CPU caches: there are many protocols and techniques to cope with noise in wireless protocols, such as error detection and error correction schemes. Previous work on cache covert channels relied on one of the two contradicting assumptions: the assumption that an error-correcting code can be directly applied, and the assumption that noise effectively eliminates covert channels [5], [32].

In this paper, we show that both assumptions are wrong. We show that caches offer some unique challenges in terms of errors and how to deal with them at the protocol level. In particular, the fact that sender and receiver may not be scheduled at the same time introduces synchronization errors. The receiver can completely miss bits when it is not scheduled, and wrongfully read additional bits when the sender is not scheduled. Sender and receiver have no trivial way to detect when the other process is not scheduled. In addition to this, virtualized environments do not offer a synchronized clock between sender and receiver.

We comprehensively characterize errors on cache covert channels, and use this knowledge to build a robust error-handling protocol. We thus show that even in the presence of heavy noise due to high system activity in cloud environments our protocol still achieves a throughput of up to 45.09 KBps between two instances of Amazon EC2. Using our robust covert channel, we demonstrate a reliable SSH connection between two virtual machines. This is not possible with any previous covert channel between virtual machines, as even a minuscule error rate will prevent a sustained connection.

To summarize, we make the following contributions:

1) We comprehensively characterize noise on cache covert channels, due to cache activity and interrupts.
2) We propose a protocol that handles the physical layer as well as the data-link layer of a robust cache covert channel.
3) We evaluate our covert channel between two virtual machines on Amazon EC2 and obtain a throughput as high as 45.25 KBps, with 0% error rate. The communication stays error-free in the presence of extraordinarily high system activity, where we obtain between 34.27 KBps and 45.09 KBps. This is 3 orders of magnitude faster than any previously demonstrated covert channel on Amazon EC2.
4) Using our robust and error-free channel, we demonstrate an SSH connection between two virtual machines on Amazon EC2, where existing covert channels fail.

Section II presents the background and state of the art. Section III describes the covert channel and the measurement method, and introduces our problem statement. Section IV characterizes noise on the cache, both due to cache activity and interrupts. Section V proposes our new protocol. Section VI evaluates the speed and robustness of our covert channel. Section VII gives a practical application: building an SSH connection over our covert channel.

II. Background and State of the Art

A. CPU Caches

CPU caches are a type of memory that is small and fast, that the CPU uses to store copies of data of main memory in order to hide the latency of memory accesses. Modern CPUs have different levels of cache, typically three, varying in size and latency: the L1 cache is the smallest and fastest, while the L3 cache, also called last-level cache, is bigger and slower.

Modern CPUs are set-associative, *i.e.*, a cache line is stored in a fixed set, as determined by either its virtual or physical address. Each line can then be stored in any of the ways of this cache set, as determined by the replacement policy. The L1 cache typically has 8 ways, and the last-level cache has 12 to 20 ways, depending on the size of the cache. The replacement policy is a crucial element for the performance of the cache. For example, for Intel CPUs until Sandy Bridge, the replacement policy has been pseudo least-recently used (LRU). Since Ivy Bridge, this has changed and is now undocumented.

The last-level cache is physically indexed and shared across cores of the same CPU. It is also inclusive of L1 and L2, which means that all data stored in L1 and L2 is also stored in the last-level cache. To maintain this property, every line evicted from the last-level cache is also evicted from L1 and L2 caches. The last-level cache, though shared across cores, is also divided into slices. The undocumented hash function that maps physical addresses to slices in Intel CPUs has been reverse-engineered [23], [44], [17].

B. Cache Attacks

Cache attacks are based on the difference of timing between cached and non-cached memory. They can be applied to build side-channel attacks and covert channels alike. Among cache attacks, access-driven attacks are the most powerful ones, where an attacker monitors its own activity to infer the activity of its victim, and in particular which cache lines or cache sets the victim has accessed.

Access-driven attacks can further be categorized into two types, depending on whether or not the attacker shares memory with its victim, e.g., using a shared library or memory deduplication. Flush+Reload [43], Evict+Reload [13] and Flush+Flush [12] all use shared memory, that is then shared in the cache, to infer whether the victim accessed a particular cache line. The attacker evicts data from the victim either by using the `clflush` instruction (Flush+Reload and Flush+Flush), or accessing congruent addresses, *i.e.*, cache lines that belong to the same cache set (Evict+Reload). These attacks have a very fine granularity (*i.e.*, a 64-byte cache line), but they are not applicable in any environment where shared memory is not available. This is the case for some cloud providers, who disable memory deduplication for security concerns.

The second type of access-driven attacks, called Prime+Probe [26], [21], [19], does not rely on shared memory and is, therefore, applicable in more restrictive environments. As the attacker has no shared cache line with the victim, it cannot use the `clflush` instruction but rather has to access congruent addresses to evict data from the victim. The granularity of the attack is coarser, *i.e.*, an attacker only obtains information about which cache set the victim accessed, and is also more prone to noise. Indeed, in addition to the noise caused by other processes, the replacement policy makes it hard to guarantee that data has actually been evicted from a cache set [11].

In the remainder of this article, we focus on Prime+Probe, as it has the lowest requirements concerning execution privileges and is more prone to noise than the other cache attacks.

C. Cache Covert Channels

Table I summarizes state-of-the-art cache covert channels. The first cache covert channel has been theoretically shown by Hu [16] in 1992. Percival [26] was the first to practically build a covert channel on the L1 cache, and estimated the capacity to 400 KBps, "using an appropriate error-correcting code". Ristenpart et al. [30] were the first to build a cache covert channel in a cloud environment, with 0.2 bps between two virtual machines on Amazon EC2. Xu et al. [41] studied the disparity between theoretical and practical results, in particular in virtualized and noisy environments. From a quiet to a noisy environment, they showed a decrease of the bit rate from 215.11 bps to 81.19 bps, and an increase of the error rate from 5.12% to 28%.

Covert channels on the last-level cache allow the sender and the receiver to run on different cores of the same CPU. Wu et al. [40] described the first cross-core covert channel using Prime+Probe on a Nehalem CPU and were followed by Maurice et al. [24] and Liu et al. [21] on more recent CPUs. Gruss et al. [12] later demonstrated cache covert channels using Flush+Reload and Flush+Flush. The authors built a framework with a protocol to be able to compare different cache covert channels. The protocol includes retransmission in case the receiver did not receive a packet in order to create a low-error covert channel. However, it does not include

TABLE I: State-of-the-art cache covert channels. Bit rates have been converted to bytes per second for comparison.

Article	Type	Setup	Bit rate	Error rate	Remarks
Hu [16]	–	–	–	–	Theoretical
Percival [26]	Prime+Probe	Native	400 KBps	–	Covert channel on L1, bit rate is an estimation with error-correcting codes
Ristenpart et al. [30]	Prime+Probe	Cloud	0.025 Bps	–	Amazon EC2
Xu et al. [41]	Prime+Probe	Virtualized	26.9 Bps	5.12%	Quiet setup
Xu et al. [41]	Prime+Probe	Virtualized	10.1 Bps	28%	Noisy environment: third VM added
Wu et al. [40]	Prime+Probe	Native	23.8 KBps	–	Error rate not measured, Nehalem CPU
Maurice et al. [24]	Prime+Probe	Native	129.1 Bps	1.6%	Quiet setup, no protocol, receiver decodes bits offline
Maurice et al. [24]	Prime+Probe	Virtualized	93.9 Bps	5.7%	Quiet setup, no protocol, receiver decodes bits offline
Liu et al. [21]	Prime+Probe	Virtualized	75 KBps	1%	RZ self-clocking encoding, busy wait between bits
Gruss et al. [12]	Prime+Probe	Native	67 KBps	0.36%	Quiet setup, 5 B packets, 1 B sequence number, CRC-16 checksum
Gruss et al. [12]	Flush+Reload	Native	298 KBps	0.00%	Quiet setup, 28 B packets, 1 B sequence number, CRC-16 checksum
Gruss et al. [12]	Flush+Flush	Native	496 KBps	0.84%	Quiet setup, 28 B packets, 1 B sequence number, CRC-16 checksum

error-correcting codes and has not been evaluated in a noisy environment. Yet, with noise, the retransmission rate will increase, and the bit rate will decrease.

The fastest Prime+Probe cross-core covert channel in the literature reports a bit rate of 75 KBps for a 1% error rate [21]. However, note that the error rate is calculated using an edit distance, *i.e.*, counting the minimum number of single-character edits, including insertion and deletion, to transform one string into another. Yet, insertion and deletion errors cause numerous substitution errors, which cannot be corrected by an upper layer in a protocol. Thus, the practical utility of such a covert channel is severely limited.

D. Countermeasures Against Cache Covert Channels

The techniques used to build cache attacks are the same for covert channels and side-channel attacks. Thus, both share the same countermeasures. The countermeasures can target timing sources by either removing them or making them more coarse-grained. They can also target the timing differences themselves by either removing them or introducing noise such that they are not exploitable anymore.

Countermeasures introducing noise include Düppel [46], a kernel component that cleanses private caches by priming all the sets until the entire cache has been evicted. This component has been created to cleanse shared-time caches, on CPUs not equipped with simultaneous multithreading (SMT). As it targets private caches, it defends against same-core cache attacks, and would need to be modified to defend against the newer cross-core cache attacks on the last-level cache, where the spy and the victim can run concurrently. Fuchs and Lee [9] propose to add a randomized prefetching policy that performs additional memory accesses and makes the behavior of the cache less predictable. Brumley [5] and Schmidt et al. [32] also cite noise as a countermeasure against covert channels.

Adding noise to caches has also been investigated on the attacker side to improve attacks. Indeed, Gruss et al. [13] and Allan et al. [1] have shown that by constantly flushing data, an attacker can slow down an encryption process and therefore amplify side-channel leaks to get more information.

E. Efficient Protocols For Noisy Channels

There are different possibilities to handle transmission errors in a communication channel. A first approach is to retransmit a data packet until its successful arrival, which is indicated either via an *acknowledgment (ACK)* or its counterpart negative ACK (NACK). To detect bit errors, parity bits are added to each packet by the sender. The receiver then checks if all the parity equations are fulfilled. If so the incoming transmission is valid. Another approach is to detect the presence of noise, e.g., an ongoing concurrent transmission, before starting to send. This avoids colliding with either ongoing transmission or channel activity strong enough to corrupt and destroy the message being sent. If so, the sender postpones its transmission for a certain time, which increases with every successive postpone, up to a maximum when the packet is dropped. This approach, typical to wireless communication, is known as carrier sense multiple access. A third approach is to avoid noise on the channel altogether by using a channel which is not in use yet or changing the communication channel periodically. This approach is known as channel hopping. Even though this method evades much temporal interference and is far more resilient than just retransmitting, it is not always possible.

If sender and receiver are not synchronized well, a different category of errors occurs: synchronization errors. If the receiver's sampling rate is too high, the receiver reads more symbols than the sender has sent. If the sampling rate is too low, *i.e.*, the sender transmits faster than the receiver is able to sample, some symbols are not seen by the receiver. It is necessary to deal with these errors first, as they often make other measures taken against noise unusable, causing too many substitution errors. Methods against synchronization errors include sampling more often to increase certainty about the measured signal or using a coding scheme which includes timing information (e.g., Manchester coding). One adopted countermeasure is to slow down communication sufficiently to achieve reasonable synchronization. It is also possible to add redundancy to messages to recover from single bit synchronization errors, but this is ineffective against burst errors and deletion or insertion of whole bytes [25]. Therefore, a basic form of synchronization must exist before these codes are applicable. Wireless sensor networks also suffer from synchronization problems: Due to their goal to save energy, the sensor nodes switch on their radios only for short periods of time to check for incoming transmissions. This is called radio duty-cycling. To make resynchronization easier, the sender starts transmissions with a specific pattern, called preamble, before sending the actual data [6].

Shannon's theorem [33] states that it is possible to compute the maximum capacity of a channel and to achieve error-free transmission through a noisy channel if the data is sufficiently

encoded and the noise is not too strong. Since the introduction of this theorem, the community has been developing codes to either detect and correct errors in data transmission. With *error detection* codes, the message is augmented with extra redundancy bits in order to detect errors, also called parity bits. There are also special codes that can detect all bit flip errors, such as the Berger codes [2], which have been used in ECC RAM. *Error-correcting* codes comprise block codes and convolutional codes. With block codes, fixed-sized blocks are encoded using methods such as the Reed-Solomon codes [29]. Although quite old, Reed-Solomon codes were the most widely used error-correcting codes [39] and are still used today, e.g., for correcting errors in Android verified boot [35]. Classical block codes are deterministic and perform well, even without dedicated hardware. Convolutional codes use additional information about received symbols in the form of probabilities, which are then used to find the word which was most likely transmitted. While being introduced rather early [10], [38], these codes require significant computational power and have started to be used only recently. These codes benefit greatly from dedicated hardware and are used in today's latest communication technologies. Examples are low-density parity-check codes and turbo codes [3].

Often, more than one code is used to enable better error correction capabilities. This technique is called concatenation. One error-correcting code is applied to the words which are transmitted, while another code is applied to the packet. In addition, the encoded data can be interleaved such that burst errors do not destroy blocks that are too big to correct.

F. Operating System and Hypervisor Scheduling

On modern computer systems, the number of processes exceeds the number of cores and processors by far. Thus, it is still necessary to run time-sharing algorithms that frequently interrupt the process running on a CPU core to hand over control to another waiting process. Besides hardware interrupts originating from I/O devices and software interrupts such as page faults, the most frequent interrupt source are timer interrupts mostly used for scheduling. These scheduling interrupts can be periodic or deadline driven (e.g., earliest-deadline first) [18]. If the hardware is configured to issue periodic timer interrupts, the timer interrupts will occur at a constant frequency that is fixed to real time and does not depend on the actual speed of the CPU. With deadline driven timer interrupts, the operating system sets a number of CPU cycles after which the interrupt will occur. On modern systems, the interrupt frequency is typically in the range of 1 Hz to 1000 Hz [7], [31]. Similarly to the operating system, hypervisors in cloud systems use timer interrupts to control the scheduling of the different virtual machines.

Scheduling has also been exploited for microarchitectural attacks and instrumented for countermeasures against microarchitectural attacks. Gullasch et al. [14] exploited a flaw in the deadline-driven "completely fair scheduler" of Linux to reduce the time the victim gets scheduled to a minimum. Thus, they were able to measure the cache footprint of the victim application at a much higher frequency. Varadarajan et al. [36] instrumented the scheduler of a hypervisor to prevent frequent context switches between an attacker virtual machine and a victim virtual machine.

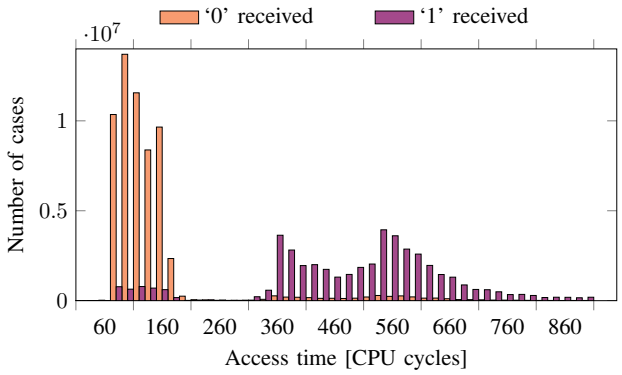


Fig. 1: Histogram for time taken to receive '0's and '1's.

III. Problem Statement

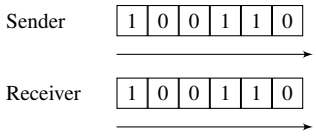
In this Section, we describe our Prime+Probe covert channel, the challenges we face in making it robust, and in particular the type of errors we encounter.

A. Description of the Covert Channel and Measurements

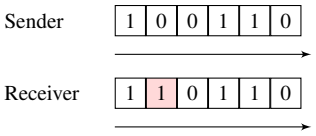
At a high level, the sender transmits bits by evicting cache lines from the receiver. The receiver constantly probes a set in his L1 cache. These cache lines are also present in the last-level cache due to the inclusive property. To transmit a '0', the sender does nothing. The lines thus stay in the L1 cache of the receiver, which thus observes a short timing to probe its lines. To transmit a '1', the sender accesses cache lines that are mapped to the same set in the last-level cache as the receiver's. The lines of the receiver are thus evicted from the last-level cache, and consequently from the L1 cache due to the inclusive property. The receiver thus observes a long timing to probe its lines, as they have to be retrieved from the DRAM.

Prime+Probe cache covert channels are based on timing information, *i.e.*, whether a received bit is a '0' or a '1' depends on the timing difference measured over a defined set of memory accesses. The measurement itself thus takes a different amount on time depending on the bit that is transmitted. Moreover, cache hits and cache misses themselves have varying timings, influenced by a multitude of factors such as unrelated system activity. Thus, the timing difference measured also varies independently of the actual bit that is transmitted, due to the memory accesses that are performed. Figure 1 shows a histogram of the measured time for the set of memory accesses when the sending party transmits a '0' or a '1' on an Intel Core i7-4790. While the two cases are clearly distinguishable, the exact time measured varies widely. If a '0' is transmitted, the access time is between 120 and 240 cycles in 93.59% of the cases. In 5.78% of the cases, the access time was above 300 cycles due to noise. If a '1' is transmitted, the access time is between 380 and 800 cycles in 86.51% of the cases. In 8.25% of the cases, the access time was below 300 cycles due to the cache eviction performed by the sender process being unsuccessful or sender and receiver evicting simultaneously.

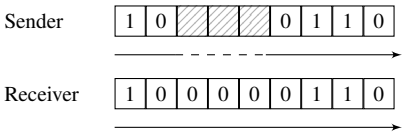
The sender and receiver processes run independently and have no way of communicating apart from the covert channel.



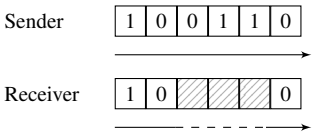
(a) Transmission without errors. Sender sends `100110`, receiver receives `100110`.



(b) Substitution error due to noise on the cache. Sender sends `100110`, receiver receives `110110`.



(c) Insertion errors due to the sender descheduled during transmission. Sender sends `100110`, receiver receives `100000110`.



(d) Deletion errors due to the receiver descheduled during transmission. Sender sends `100110`, receiver receives `100`.

Fig. 2: Illustration of a normal transmission and the three different types of errors. Dashed lines represent the time when either the sender or the receiver is not scheduled.

Thus, the sender cannot determine how many cycles it took the receiver process to measure the transmitted bit. Hence, the operations of the two processes are inherently not synchronized.

B. Measurement Errors

There are mainly two sources of errors in cache covert channels. First, the cache is a resource shared with other programs. It is a rather small memory, thus data gets evicted by other concurrent programs. As eviction is also used by the sender to transmit a bit, a heavy use of the cache creates interferences with the covert channel in the form of false positives for the receiver. It can also introduce false negatives if the eviction is not successful. These kinds of errors are called substitution errors. Second, sender and receiver are not always running simultaneously, depending on the scheduler. As they have no way of signaling that they are either not transmitting or not receiving any bits, this results in insertion or deletion errors on the receiver's end.

These issues result in three distinct type of errors. Figure 2 illustrates a normal transmission and the effect of errors.

Substitution errors: Another program causes eviction of the receiver's lines, resulting in '0' $\rightarrow$ '1' bit flips. Substitution from '1' $\rightarrow$ '0' happens when the sender unsuccessfully evicted the set.

Insertion errors: The sender is interrupted or not scheduled, while the receiver still continues its measurements. The receiver will read consecutive '0's while the sender is being descheduled. If not handled correctly, the receiver will continue to put the incoming transmission after the block of '0's, resulting in corrupted data after the first appearance of this kind of error. Insertions of '1's are possible in the case where there is a lot of noise, but this happens less than insertions of '0's.

Deletion errors: The receiver is interrupted during measurement, which leads to a sequence of bits never being read and lost without being noticed. If not handled correctly, the sender will continue as if nothing happened; the result would be faulty data beginning from the first occurrence of this error.

Insertion errors and deletion errors are synchronization errors that are due to the changing sampling rate of both the sender and the receiver. These errors are caused by scheduling and are therefore burst errors. Synchronization errors further complicate the problem, causing a dramatic increase in substitution errors until the next synchronization—if any—, as the receiver has the wrong symbol boundaries. It is, therefore, necessary to first correct synchronization errors before substitution errors.

C. On the Absence of a Common Clock

In a native environment, covert channels between processes can be synchronized based on a common clock, such as the time stamp counter. The time stamp counter provides a high-resolution timestamp which is accessible through the unprivileged `rdtsc` instruction. This cycle count can be used for highly accurate measurements even on a sub-nanosecond scale. While it is sometimes assumed that such a common clock exists in cloud environments as well, this is in fact not the case for security and compatibility reasons. For instance a virtualized `rdtsc` can make sandbox detection more difficult. At the same time virtualizing `rdtsc` incurs wide deviations when compared inside two independent virtual machines. This is due to the fact that the `rdtsc` base offset is frequently adapted by the hypervisor to account for interrupts [22]. Hypervisor interrupts are sometimes completely hidden from the virtual machine, *i.e.*, the timestamp counter is not updated. In fact, there is typically no common clock apart from very coarse-grained clocks in the range of milliseconds. Even here clock deviation can occur, depending on the hypervisor.

In order to reliably transmit and receive data, the communicating parties do not require a synchronized clock. The clock does not deviate as long as the current party is running, but the clock can deviate as soon as the communicating party is descheduled. Wireless sensor networks have to deal with similar problems, as the sending party often does not know if its communication peer is awake. The communicating parties using the cache covert channel have the problem that they are switched off seemingly randomly, in contrast to sensor nodes, which decide themselves when they switch off their radios. Therefore, due to the emulated `rdtsc` and scheduling, we

TABLE II: Experimental setup.

Environment	CPU model	Cores	LLC associativity
Lab	Core i5-5200U	2	12
Lab	Core i7-4790	4	16
Lab	Xeon E3-1220v3	2	16
Amazon EC2	Xeon E5-2670	8	20

need to resynchronize during communication, while sensor nodes only need to synchronize once and note the wake-up time of its peer.

Intuitively, a common clock might solve these problems. Generally the transmitted signal can be used to derive a common clock or also transmit a clock signal. However, the transmission of a common clock would suffer from the same issues as the transmission of the data itself. The only possibility would be to transmit a self-clocked signal, which also needs to be resilient against noise. Thus, the transmission of a clock signal is possible but causes a huge overhead. As described in Section III-B, sender and receiver can run simultaneously, but due to interrupts and scheduling, there are phases where one process is active but not the other. If the sender transmits a clock signal and the receiver misses multiple clocks, the receiver cannot detect this error. Similarly, the sender has no way to detect that the receiver was sleeping. Thus, additional bits would need to be transmitted to reduce the probability of errors in the clock signal.

D. Problem Statement

While many of the errors that are encountered on our channel have already been solved in communication engineering and information theory, there is no readily usable design to solve the particular issues of our channel. Moreover, we are aiming at a high capacity channel, while minimizing the complexity of the protocol. It is, therefore, impossible to *just apply error-correcting codes* as was hinted in previous work.

As we cannot influence noise or synchronization issues on our channel, we apply existing methodology for wireless communication. We start by characterizing noise on the channel, both due to other applications and the scheduler. We then design a protocol that is capable of correcting these errors and evaluate our covert channel in high-noise environments.

E. Experimental Setups

In the remainder of the article, we use the experimental setups described in Table II.

For our experiments in a cloud environment, we used Amazon EC2 g2.2xlarge instances. Recent works have already shown how to acquire co-located instances with a victim [37], [42], [17]. As the co-location step is not in the scope of our paper, we obtain co-located instances by performing our experiments on a dedicated host. The g2.2xlarge instances are backed by 8 vCPUs from an Intel Xeon E5-2670, which corresponds to 8 hardware hyperthreads according to Amazon. The larger instance of this family, g2.8xlarge, has 32 vCPUs. As a Xeon E5-2670 has 8 cores, and it is possible to fit 32 hardware hyperthreads on a single machine, we speculate that one machine is composed of 2 CPUs, and we run 3 instances on the same host.

Algorithm 1: Creating a heatmap

input : nb_sets: int, nb_ways: int, $threshold$: int,
$nb\_measure$: int,
$pointers$: uint64_t[nb_sets][nb_ways]
output: $expected$: double[nb_sets]

```
cases ← new int[nb_sets][nb_ways + 1];
expected ← new double[nb_sets];
for 1 .. nb_measure do
    for s ← 0 to nb_sets do
        misses ← 0;
        for l ← 0 to nb_ways do
            time ← probe_line(s,l);
            if time > threshold then misses++ ;
        cases[s][misses]++;
for s ← 0 to nb_sets do
    for l ← 0 to nb_ways + 1 do
        expected[s] += l × cases[s][l]/nb_measure;
```

IV. Characterizing Noise on the Cache

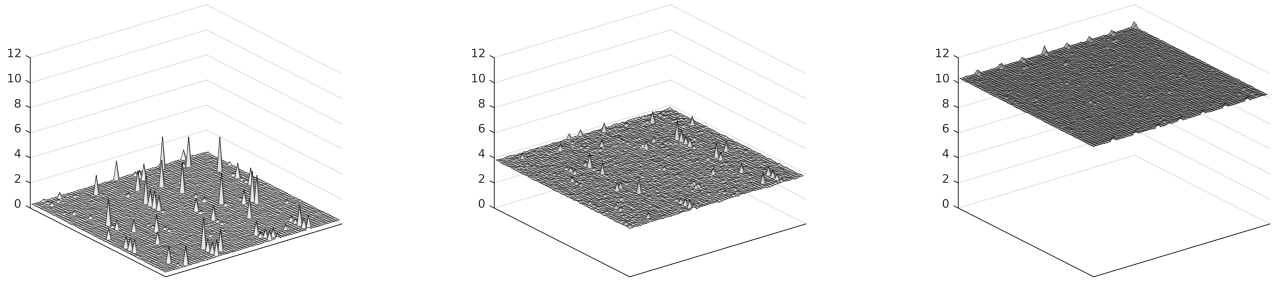
In this Section, we characterize the noise on the channel due to cache activity of other programs as well as interrupts.

A. Characterizing Eviction by Other Programs

We first characterize unwanted cache line evictions due to cache activity of other programs. Heatmaps have already been used to find cache usage patterns on the L1 cache [46], [36], [9]. We propose a new heatmap to help to determine the parameters of the covert channel, by computing the expected value of the number of evicted cache lines for every set of the last-level cache.

To do so, we run a Prime+Probe attack on all sets, as described in Algorithm 1. We start with an array of addresses mapping to every set, with nb_ways addresses per set. We probe all addresses mapping to one set, and measure the probing time of each line to determine if it is a hit or a miss. We thus count the number of evicted lines for each set. The probing function `probe_line` first accesses a line, then waits for some time, so that other programs have the time to evict it, and finally times the access of this line. The waiting time is performed using the `sched_yield()` Linux function. We repeat this procedure for every set, and repeat the measurements over all sets to compute probabilities of the number of cache lines evicted, and subsequently the expected value for each set.

The heatmap of three scenarios can be seen in Figure 3, for an i5-5200U, with 10240 measurements. The expected value is on average 0.38 on a quiet system, 3.89 while watching a 1080p video, and 10.41 while running `stress -m 2`. We observe that noise is equally distributed over all sets (*i.e.*, the variance is very low), and that as expected, the more memory-intensive the application, the higher the expected number of evicted cache lines. This metric can be used both to measure the noise caused by other applications on the cache and to choose parameters for the covert channel.



(a) Quiet system: mean = 0.38, variance = 0.016.

(b) Watching a 1080p video: mean = 3.89, variance = 0.0043.

(c) Running `stress -m 2`: mean = 10.41, variance = 0.0010.

Fig. 3: Heat maps of the expected value for the number of lines evicted by other programs in 3 scenarios, on an i5-5200U (12-way cache). Small square represent last-level cache sets. The height of a square represents the number of evicted lines.

The heatmap gives us a good visual indication of the kind of noise. However, to choose optimal error-correcting codes, we are interested in a mathematical model of the noise. The overall noise in a cache set is a summation of different noise sources. Each program running on the CPU accesses the cache in an unpredictable manner, *i.e.*, the access pattern appears to be random for an observer. According to the central limit theorem, the summation of a large number of random effects will be approximately normal [8]. Assuming that the noise is Gaussian is, therefore, a valid assumption in this scenario.

The exact location of data inside a cache is determined by its address and consequently also the cache slice function. As programs have no control over these parameters, the data is uniformly distributed over the cache. The heatmap visualization shows that this is indeed the case for the empirical experiments. Therefore, we assume that the cache sets are independent and identically distributed random variables. Under this assumption, it is sufficient to analyze only one cache set to characterize the noise.

Similar to the heatmap, we analyze one set over time. At discrete time intervals, we determine the number of lines that have been evicted from the cache set. If this number is higher than our threshold for sending a bit, the set is "destroyed" by noise, represented as -1 in the time series. In contrast, if fewer lines are evicted, the set is usable for transmission, represented as 1 in the time series. Applying the autocorrelation function $\rho(\tau)$ on this data reveals underlying noise patterns. We repeated this analysis for multiple cache sets over varying time spans. As a result, $\rho(\tau)$ was always zero for non-zero τ. Given these empirical results, we can assume that the observed signal is white noise.

The Gaussian white noise generated by processes running on the same CPU is always present in our channel. We can model the background noise as *additive white Gaussian noise* (AWGN). An AWGN channel does not account for different noise sources present in wireless communication such as fading, dispersion or interference. However, if these effects play no role, the model is sufficient to describe the noise [15]. The absence of such physical effects makes the AWGN an accurate model. Real-world applications can be found in deep-space communication [20]. The model applies to cache attacks as well, as there are no physical effects generating noise.

One important property of Gaussian noise is that the noise is independent of the data. As the sender and receiver access the cache as well, it seems that we cannot guarantee this property. Inevitably, there will be sets containing data-dependent values. However, as they form only a subset of all available cache sets, we will always be able to use sets that are independent of the data.

Modeling the background noise as AWGN is a common principle in wireless sensor networks. Most of the research in the area of wireless communication view AWGN as a standard model to characterize wireless transmission. AWGN is the worst-case additive noise in general wireless networks if the noise is independent of the transmitted data [34]. Having a robust communication over this channel guarantees that the communication will work in all other noise scenarios at least as well, as long as there are no burst errors. However, to handle burst errors, encoded packets can be interleaved.

B. Characterizing Interrupts

As already discussed, scheduling introduces either insertion or deletion errors. These errors are in all cases burst errors.

Hardware interrupts are always handled in kernel space. Thus, control is handed to a kernel thread for some time, just as regular scheduling interruptions. Therefore, we do not distinguish between hardware and scheduling interrupts.

The frequency of the Linux *process scheduler* influences the running time after which a process is interrupted. This frequency depends on several kernel parameters, such as `CONFIG_NO_HZ` or `CONFIG_HIGH_RES_TIMERS`. We derive the maximum timer interrupt frequency by requesting such interrupts and measuring their time difference. This gives the highest frequency we have to expect for scheduling interrupts. Depending on the CPU model, we observe frequencies of up to 1 MHz. In order to calculate the average length of a descheduled phase, we require the average scheduling frequency. We read the number of schedules s from `/proc/self/status`

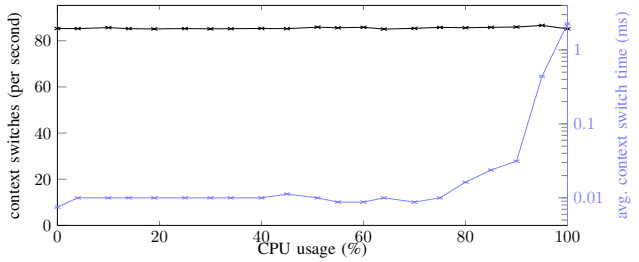


Fig. 4: Number of schedules and average scheduling times at different CPU loads.

after a fixed time t. We observe an average scheduling frequency of 85 Hz. During the time t we measure the CPU time t_{CPU} of the process, *i.e.*, the time the process actually runs on the CPU. The average length of the descheduled phase is then $t_{schedule} = \frac{t - t_{CPU}}{s}$.

The length of the descheduled phase is heavily affected by the current CPU usage. The time a process is not running on the CPU is directly proportional to the number of active processes. Figure 4 shows the results of measurements conducted on an Intel Core i7-4790. For low CPU utilization, the length of an average descheduled phase is 10 µs. The time our program is not scheduled is below 1 ms each second. For high CPU utilization, the length of a descheduled phase increases to several milliseconds. At a scheduling rate of 85 schedules per second, the process is not running for several hundred milliseconds per second.

In a quiet environment, one solution to prevent errors caused by not being scheduled is to increase the transmission time of one bit. If the transmission time exceeds the time it takes to get re-scheduled, no bit is lost. The average length of a descheduled phase requires a minimum transmission time of 10 µs for one bit, resulting in a maximum transmission speed of 100,000 bits per second. For higher CPU utilization, *i.e.*, under noise, this would again introduce synchronization errors and therefore require even slower transmission speeds. Hence, we will instead ensure that a word is sufficiently small to be transmitted during one scheduling.

V. Robust Cache Covert Channels

This Section contains the description of the communication protocol we developed for the covert channel, first on the physical and then on the data-link layer. The data-link layer receives a buffer with data which is to be transmitted. Before transmission starts, the data is divided into chunks of the same size, which are afterwards enhanced with error correction, resulting in a *packet*. Then transmission is started, where each packet is divided into *words* small enough to be transmittable easily in between scheduling. These words are then again encoded to guard them against synchronization errors. The whole encoding process can be seen in Figure 5 including the requests sent on the physical layer. We will explain in detail how every layer works in the remainder of the section.

A. Physical Layer

The physical layer is concerned with transmitting words as a sequence of '0's and '1's. It has to make sure that the receiver can reliably detect incoming transmissions. Additionally, it has to take decisions about the encoding on the medium itself as well as dealing with synchronization errors. Words consists of multiple symbols (single bits) that are transmitted simultaneously.

Algorithm 2: Sender transmitting a '1'

```
input: nb_ways: int, addrs: int[nb_ways]
for i ← 0 to nb_ways − 1 do
    *addrs[i];
    *addrs[i+1];
    *addrs[i];
    *addrs[i+1];
```

Algorithm 3: Receiver accessing a set.

```
input: nb_probes: int,
       addrs: int[nb_probes]
for i ← 0 to nb_probes − 2 do
    *addrs[i];
    *addrs[i+1];
    *addrs[i+2];
    *addrs[i];
    *addrs[i+1];
    *addrs[i+2];
```

1) Transmitting a symbol: Transmitting and receiving bits relies on priming and probing lines of set associative caches. In general in wireless communication, not only one but several bits per symbol are transmitted using different modulation techniques, such as amplitude modulation, phase-shift keying or frequency modulation. These methods are not available due to the nature of the signal and the unreliable clock.

To transmit any information, the sender primes lines in one last-level cache set. It has to prime enough lines in order to actually evict the receiver's lines inside the same last-level cache set, leading to an eviction from the receiver's L1 cache set. Depending on the number of lines evicted from the L1, the receiver could then infer which symbol was transmitted. The minimum of information to transmit is either achieving eviction ('1') or not ('0'). The sender does not need to evict all lines of this last-level cache set, as the L1 has fewer ways than the last-level cache. However, there is no way for it to know which lines of the last-level cache are present in the L1 cache set of the receiver. In practice, we found that the optimal number of lines primed by the sender is the number of ways in the last-level cache. Since Ivy Bridge, the replacement policy is not pseudo-LRU anymore. To increase the probability of evicting the set, we access cache lines with the same patterns as described by Gruss et al. [11]. Algorithm 2 describes set eviction.

To receive any information, the receiver probes nb_probes lines in one L1 cache set. At the minimum, the receiver has to probe one line. Using more lines opens up the possibility to transmit more data through one set. Intuitively, probing all L1 lines and counting the number of evictions seems to be the best way to achieve the maximum of bits per symbol, but it incurs the risk of having interferences with lines being evicted

to L2 and last-level cache. This can happen either because of the sender itself probing more than 8 lines—as the L1 is 8-way associative—or because of other programs even in the case the receiver probes less than 8 lines. However, even at a lower number of lines probed, counting the number of evicted lines proved to be too unreliable to infer more than 1 bit of information as self-eviction and interference by other programs introduced too much uncertainty. Therefore, only one bit per cache set or symbol is transmitted. In practice, we fixed $nb\_probes = 5$. The decision whether a value of 0 or a value of 1 was transmitted is based on the number of CPU cycles (cf. Section III-C) the memory accesses take. The memory accesses are done in the same fashion as the receiver, by accessing memory lines in a certain pattern, as shown in Algorithm 3.

2) Transmitting words: In order to increase the capacity of the covert channel, we can use spatial multiplexing to transmit multiple symbols (bits) of a word concurrently. We do so by exploiting the different cache sets, which do not interfere with each other. In this setup, each channel is used to transmit one bit, either a '0' or a '1'. The total number of symbols per word depends on the error detection codes that we apply. The order of the bit fields is the order in which the sets are probed. Spatial multiplexing is a difference with the covert channel of Liu et al. [21], where only two sets are used, one to transmit '1's exclusively, and the other '0's exclusively. In order to introduce further reliability, the receiver reads every word several, odd number of times and does a majority vote on each symbol.

To avoid triggering the prefetcher and unintentionally prefetching a line in another set, both the sender and the receiver must access lines that are in different 4 KB pages, as the hardware stride prefetcher does not prefetch lines across page boundaries. The sender thus avoids to unintentionally prime a set that should not be primed, which could cause the receiver to receive a '1' instead of a '0'. The receiver also avoids prefetching a line that it could probe afterwards, risking receiving a '0' instead of a '1'.

3) Set agreement: Sender and receiver must agree on which sets to use to transmit words. The main challenge to this part is that an unprivileged process — or a process in a virtual machine — cannot translate virtual addresses to physical addresses, and therefore cannot target a specific set index in a specific slice. As noted by previous work, it is still possible for an unprivileged process to target a set index, irrespective of the slice, using 2 MB pages [19], [21]. Indeed, a 2 MB page gives the 21 least significant bits of a physical address, as they are then the same for virtual and physical addresses. The set agreement is done in two steps: (1) both the sender and the receiver perform the same procedure to build eviction sets, (2) they perform an agreement to ensure that they both target the same set in the same slice.

In the first step of the set agreement, both the sender and the receiver select candidate addresses that form *eviction sets*, *i.e.*, addresses that belong to the same set in the same slice, without knowing which exact slice. We describe a new method that uses the information given by the reverse-engineered function from Maurice et al. [23] without privileges, *i.e.*, without using physical addresses[1]. We call a *slice physical index* the slice index computed with the addressing function on all bits of the physical address. Knowing only a virtual address, we compute the *slice virtual index* by truncating the 21 least significant bits of the address and applying the reverse-engineered addressing function. Inside a 2 MB page, all addresses that have the same slice virtual index belong to the same slice. To each pair (page, slice virtual index) corresponds a slice physical index. The goal is to find the virtual index for each page, such that it corresponds to the same physical index for all pages, without knowing this physical index.

The following notations hold for a CPU with c cores and a w-way last-level cache:

- p is a 2 MB page.
- i is a set index, irrespective of the slice.
- j is a slice virtual index.
- $S_{p,i,j}$ is the set of addresses in a page p, that belong to the same cache set, *i.e.*, that have the same cache set index i and slice virtual index j. $|S_{p,i,j}| = n_{\text{a}}$.
- $Sr_{i,j}$ is the probe set in the reference page p_{r}, composed of addresses with the same set index i and slice virtual index j. It is sufficient that this set is composed of two addresses. The addresses of the probe set must be evicted by the eviction set, ensuring that we obtain addresses with the same set index in the same physical slice.
- $Se_{i,j}$ is the eviction set for i and j, *i.e.*, the set of addresses evicting $Sr_{i,j}$. Given a set $Sr_{i,j}$, we thus seek to build the set $Se_{i,j}$.

P is the set of allocated 2 MB pages, *i.e.*, $p \in P$ with $|P| = n_{\text{p}}$. We exclude the reference page p_{r} from P. I is the set of set indices, irrespective of the slice, *i.e.*, $i \in I$ with $|I| = 32$. Indeed, to ensure that each address belongs to a different 4 KB page, we only target sets that have addresses aligned to 4 KB. There are 2048 set indices per slice, *i.e.*, 32 different sets aligned with 4 KB pages—independently of the number of cores. There are 16 addresses having the same set index i per 2 MB page, *i.e.*, $n_{\text{a}} = 16/c$ addresses belonging to the set index i in the same slice. We therefore need to allocate $n_{\text{p}} = \lceil w/n_{\text{a}} \rceil$ pages to obtain an eviction set of w addresses.

J is the set of slice virtual indices, *i.e.*, $j \in J$ with $|J| = c$. As we do not have the knowledge of all bits of the physical address, the slice virtual indices are different for every 2 MB page. For example, if an address a_{r} in the reference page p_{r}, an address a_1 in a page p_1 and an address a_2 in a page p_2 all belong to the same slice physical index 1, it is possible that the virtual index for a_{r} is 0, the one for a_1 is 2 and the one for a_2 is 3. As the indices in themselves are irrelevant, we take as a reference the virtual indices of addresses in the reference page p_{r}. We thus seek to find the corrective offset o_p to apply for each page p, so that the slice virtual index j on page p and the slice virtual index j_{r} on the reference page p_{r} correspond to the same slice physical index. In our previous example, $o_{p_1} = 2$ and $o_{p_2} = 3$.

We build an eviction set $Se_{i,j_{\text{r}}}$ for a fixed set index i and the slice virtual index $j_{\text{r}} = 0$. We start by constructing the set of candidate addresses Sc_i, which is our test eviction set. We

[1]If the last-level cache addressing function is unknown, it is possible to revert to the method of Liu et al. [21] which assumes no prior knowledge.

have $Sc_i = \{ S_{p,i,j} \mid p \in P, j \in J \}$, with $|Sc_i| = n_\text{p} \times c$. By construction, this set contains the addresses of the eviction set that we are searching for, as well as more addresses that belong to the same set index, but a different slice. We seek to remove these additional addresses to build an optimal set. Thus, for one page p and set index i, we remove one set $S_{p,i,j}$ of n_a addresses from Sc_i. We then test the eviction of the probe set with this smaller test eviction set. If eviction succeeds, then the set $S_{p,i,j}$ is not part of the final eviction set, and we remove it from the test eviction set. For one page, we remove sets $S_{p,i,j}$ testing all possible values for j until the probe set is not evicted anymore. We have then found one set $S_{p,i,j} \in Se_{i,j_\text{r}}$, *i.e.*, the set of addresses for the page p belonging to the final eviction set for the set index i. We repeat this procedure for all pages $p \in P$, until we have the final eviction set Se_{i,j_r}. We can compute the corrective offset o_p for page p, $o_p = j \oplus j_\text{r}$, with $S_{p,i,j} \in Se_{i,j_\text{r}}$. As we have fixed $j_\text{r} = 0$, we have $o_p = j$, and we can now compute the eviction sets for other values of j_r and i.

In the second step of the set agreement, the sender and the receiver ensure that they both target the same sets using jamming agreement, inspired by the method of Boano et al. [4] in wireless communication. The sender and receiver both have a set of eviction sets. As every slice virtual indices are computed given their own reference page, they do not match between the sender and the receiver. The sender and the receiver perform *jamming* and *listening* operations that are the same as sending and receiving bits during the transmission. The sender starts by jamming its first set, *i.e.*, accessing its own set in a loop that we fixed to $5,000$ iterations. The sender then listens if the receiver jams back to agree on this set, *i.e.*, it measures the time it takes to access this set in a loop that we fixed to $10,000$ iterations. The sender repeatedly jams and listens to one set until it receives a signal back from the receiver. During this time, the receiver listens to one of its own eviction sets in a loop that we fixed to $10,000$ iterations. If the number of misses is greater than a threshold, the receiver jams back for $15,000$ iterations. If not, the receiver repeats its procedure for the next set. When the sender gets a signal for one set, it repeats this procedure until they agree on enough cache sets to proceed to the communication.

Table III shows the evaluation of the speed of the set agreement, comprising of the set creation and jamming agreement, in native environment and between two instances on Amazon EC2, on 100 measurements, with varying degree of noise. In a native environment on a 4-core i7-4790, without noise, the agreement takes on average $0.30\,\text{s}$, with a standard deviation of 0.03. When running `stress -m 1`, the agreement takes on average $0.40\,\text{s}$, with a standard deviation of 0.02. On Amazon EC2, the jamming agreement between two instances takes on average $2.62\,\text{s}$ with a standard deviation of 0.15 without any noise. When running high synthetic noise such as `stress` whether on the sender, receiver or a third virtual machine, the jamming agreement still works and takes between 2.74 and $3.47\,\text{s}$. The jamming agreement is unlikely to succeed in case of an extremely high noise such as `stress -m 8` on the third virtual machine. The reason is the high amount of noise on many cache sets that is caused by the high system load. As described in Section V-A1, the lack of a reliable amplitude on our measurement makes it impossible to distinguish different noise levels on a particular cache set. Thus, the two parties cannot distinguish intensive noise on a cache set from cache misses induced by the other party.

TABLE III: Speed of set agreement in native environment (i7-4790) and between 2 instances of Amazon EC2, on 100 runs.

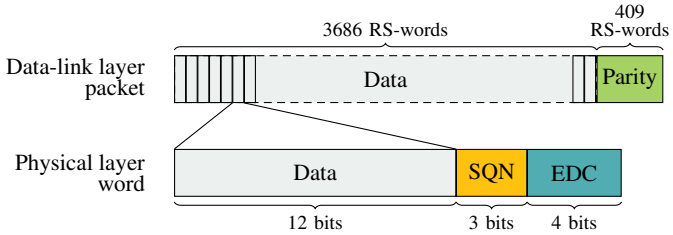
Environment	Average speed (s)	Standard deviation	Noise
Native	0.30	0.03	–
Native	0.40	0.02	`stress -m 1`
Amazon EC2	2.62	0.15	–
Amazon EC2	2.74	0.22	`stress -m 1` on the sender VM
Amazon EC2	3.47	0.38	`stress -m 1` on the receiver VM
Amazon EC2	2.95	0.27	`stress -m 4` on the third VM
Amazon EC2	–	–	`stress -m 8` on third VM

4) Handling Synchronization Errors: The covert channel suffers from the three distinct errors described in Section III-B of which two are due to synchronization issues described in Section IV-B.

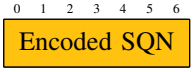
Deletion errors: Deletion can only appear if the receiver is descheduled while the sender continues to transmit. Therefore we apply a simple request-to-send scheme that also serves as acknowledgment. Every word contains a 3-bit sequence number, which is used to avoid desynchronization at the low level. The receiver repeatedly transmits a request with a sequence number. Upon reception, the sender begins to send the requested word and transmits this word until the receiver transmits the next request – which serves as acknowledgment of the current request. The sequence number is encoded with Hadamard codes to make the scheme robust against substitution errors, resulting in a 7-bit request. We can always recover 2-bit or detect errors as long as less than half the bits are flipped. We can also use the nature of the substitution errors to calculate the most probable candidates and increase the error correcting capability. We chose to use the Hadamard codes only for their error detecting capabilities as using them otherwise introduced too many wrongly decoded requests.

'0'-Insertion errors: Inserted '0's happen if the sender is descheduled while the receiver continues its measurements. To reliably detect these errors we use Berger codes, which consist in appending the number of '0's contained in the word to itself. The Berger code needs $\lceil \log_2 (len(sqn + data) + 1) \rceil$ bits, in our case 4 bits. While Berger codes are typically used to correct unidirectional substitution errors, we use their properties to detect insertion errors. In addition to detecting errors, Berger codes guarantee the property that a word cannot consist solely of '0's, since even if a data-word only consists of '0's, the transmitted word will contain '1's. The Hadamard codes used for the request do not guarantee non-zero words. Therefore, we do not allow the sequence number '0'. The seven different sequence numbers introduce a sliding window of size 6, *i.e.*, the receiver can detect a sequence number mismatch in the range ± 3. If there is a mismatch between sender and receiver, e.g., the sender skips a word due to substitution errors, which is accepted by the receiver. The position of the skipped word is then noted and used in the error correction in the data-link layer (see Section V-B).

By not allowing any '0' words to be transmitted, '0' readings are dismissed as errors and are not saved for the majority vote. In addition, one reading before the first '0'



(a) Relation between words on the physical layer and packets on the data-link layer.



(b) A request on the physical layer.

Fig. 5: The structure of a word and an acknowledgment.

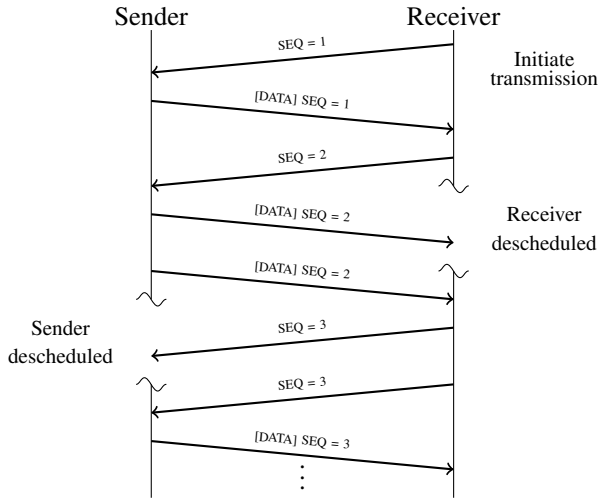


Fig. 6: Retransmission on the physical layer due to scheduling interrupts.

reading and after the last one are dismissed as well, due to the chance of being affected by insertion errors as well.

Figure 5 shows the structure of the 19-bit word and the 7-bit acknowledgment that are transmitted on the physical layer. A 4-bit Berger code is used as Error Detection Code (EDC) for the 12-bit data and the 3-bit sequence number (SQN) of the word. The acknowledgment consists only of the Hadamard encoded sequence number. Figure 6 shows the transmission in the absence of noise. Interrupts caused by scheduling trigger implicit retransmission of packets. In noisy environments, the number of retransmissions will increase due to corrupt packets.

5) Channel Model: Channels in communication engineering are commonly modeled after a *binary symmetric channel* (BSC). In this model, a signal arrives at the receiver with probability $1-p$ correctly or with probability p flipped. This is the case for our covert channel, see Figure 7. The higher the system load the higher the probability of a bit flip.

Our channel consists of several parallel BSCs which all have the same error probability distribution, see Section V-A4.

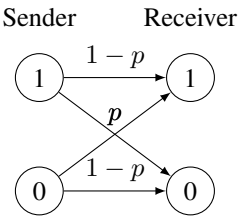


Fig. 7: The covert channel without synchronization errors: A binary symmetric channel

Using that we can calculate the probability of receiving a word without errors: $p_{cor,word} = (1-p)^n$ where n is the number of bits transmitted per word. Therefore $p_{err,word} = 1-p_{cor,word}$. Most of the erroneous words will be found by the Berger code and will be corrected by retransmissions, while the rest will have to be corrected by the data-link layer.

B. Data-Link Layer

The physical layer transmits symbols with a fairly low error rate, but it is still not error free, a property which we need for further applications. We showed in Section IV that the noise on the channel is AWGN and will therefore uniformly destroy bits and words with the error probability shown in Section V-A5. Thus we chose a commonly used error-correcting code with high decoding speed for small packets: Reed-Solomon codes (RS codes). In order to achieve fast encoding and decoding speed we use an RS word size of 12 bits for encoding, therefore our packet size is $2^{12}-1 = 4095$ RS words with $4095 - len(data)$ parity. We use the word size of the physical layer as RS word size, as an erroneous symbol does not affect more than one RS word. For small packets of data (*i.e.*, less than 100 bytes), it is not advisable to use a RS word size of 12, as it would introduce a huge overhead in transmitted parity. Therefore we choose to use 8 bits for RS word size in these cases, resulting in 255 bytes per RS encoded packet.

We adjust the number of parity RS words to be able to correct more than $packet\,size \cdot p_{err,word}$ symbols. RS codes can correct up to $\lfloor \frac{parity}{2} \rfloor$ RS words. Therefore, we have twice the number of expected errors as parity RS words. If the positions of the erroneous RS words are known, one can correct up to a maximum of $parity$ errors, which is exploited if a symbol has been skipped by the physical layer. Using these numbers, we decided to use 10% error-correcting code 409 parity and 3686 data RS words for packet size 4096 and 25 parity and 230 data RS words for packet size 255. With these settings, we achieve error-free communication while either a YouTube video is being watched or `stress -m 1` is running.

VI. Evaluation

In this section, we evaluate our covert channel on different environments, including between two instances of the Amazon EC2 cloud, and in the presence of high system activity. For our tests we transfer 9 MB of image data between the sender and the receiver. To compute the error rate, we compare the transmitted data bit-by-bit with the original. Our experimental setups are described in Table II. The results are summarized in Table IV.

TABLE IV: Experimental results, with 10% error-correcting codes.

Environment	Bit rate	Error rate	Corrected errors	Noise
Native	75.10 KBps	0.00%	6333	–
Native	36.03 KBps	0.00%	13166	`stress -m 1`
Amazon EC2	45.25 KBps	0.00%	12996	–
Amazon EC2	45.09 KBps	0.00%	11574	web server serving files on sender VM
Amazon EC2	44.64 KBps	0.00%	11258	`stress -m 1` on sender VM
Amazon EC2	44.25 KBps	0.00%	11819	web server serving files on third VM (10 concurrent users)
Amazon EC2	42.96 KBps	0.00%	8462	`stress -m 2` on sender VM
Amazon EC2	42.26 KBps	0.00%	6974	`stress -m 1` on receiver VM
Amazon EC2	37.42 KBps	0.00%	6093	web server serving files on all three VMs, `stress -m 4` on third VM, `stress -m 1` on both sender and receiver VMs
Amazon EC2	34.27 KBps	0.00%	7147	`stress -m 8` on third VM, started after the jamming agreement

TABLE V: Channel capacity and observed errors with different sizes for the error-correcting code (in relation to the total packet size).

ECC	Bit rate	Bit errors	Byte errors	Corrected errors
10%	43.70 KBps	0	0	6922
4%	47.34 KBps	0	0	6728
2%	48.76 KBps	105	59	6570
1%	46.69 KBps	9448	6120	2093
0%	48.29 KBps	14841	9509	0

A. Lab-controlled Environment

We start by evaluating our covert channel in a native environment, on a Core i7-4790 CPU. The sender and the receiver are two unprivileged processes that do not share memory. For our measurement, we transmitted 9 MB of data over 2 minutes. Without any additional noise, our covert channel achieves a bit rate of 75.10s KBps and an error rate of 0.00%. We then increased the noise on the machine by running `stress -m 1`. Our covert channel stays error-free and achieves a bit rate of 36.03 KBps.

Figure 8 illustrates the need for error correction as well as its different stages. Figure 8a illustrates the result of the transmission, prior to solving scheduling issues: bit insertions and deletions render the image unreadable. Even a few bit insertions or deletions that would yield a low edit distance can create an entirely illegible image. Figure 8b is obtained after dealing with synchronization issues, but prior to applying RS codes: with a bit error rate of 0.72%, the image is readable but noisy. Finally, Figure 8c shows the result after applying error correction: we obtain a crystal clear, error-free image.

We also analyzed how the size of the error-correcting code influences the raw channel capacity and the number of errors. We performed the experiments on a Xeon E3-1220v3, in a virtualized environment with the KVM hypervisor. Table V shows that increasing the percentage of error-correcting code slows down the connection if too much error correction is applied and increases the number of corrected errors due to the slightly increased length. We can see that at 2% error correcting a few errors are introduced. During all the experiments the average amount of errors per packet stayed the same: 22 RS-words per RS-message were erroneous, which is correctable by 2% error-correcting code. However, due to errors of up to above 40 in one message the 2% are not enough.

B. Cloud Environment

For our experiments in a cloud environment, we used three instances on Amazon EC2 as described in Section III-E. As our covert channel operates across cores but not across CPUs, we use the jamming agreement to find the two instances that reside on the same CPU. In the remainder, the sender runs on one instance, and the receiver on the other instance, and communication is performed across virtual machines. Table IV contains the results for the communication in terms of speed, error rate and number of errors corrected.

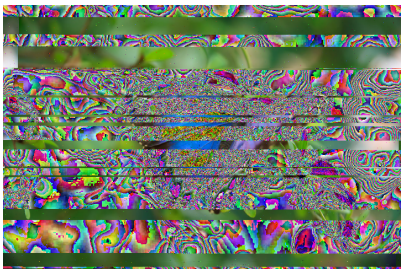
Without any additional noise, our covert channel achieves a bit rate of 45.25 KBps with an error rate of 0%. We then varied the noise on the sender and receiver virtual machines, as well as on the third instance. Unless stated otherwise, noise is started before the jamming agreement. The communication stays error-free with 45.09 KBps when running a web server on the sender virtual machine. It still stays error-free with extreme synthetic noise generated by the `stress` tool in a single virtual machine, with a bit rate going from 42.26 KBps to 44.64 KBps. Likewise, with a mix of synthetic noise and web server running on all three virtual machines, we obtain a bit rate of 37.42 KBps with 0% error rate. Our most extreme case, running `stress -m 8` on the third virtual machine, breaks the jamming agreement, but not the transmission itself that is 34.27 KBps with 0% error rate when the noise starts after the jamming agreement.

Table IV also shows the relation between the errors corrected by the RS codes and the bit rate. We see that the number of corrected errors decreases with the bit rate, and when noise increases. With more noise, the Berger codes at the physical layer are able to detect more errors, which causes more retransmissions, leading to fewer errors which need to be corrected by RS codes.

C. Comparison with State of the Art

The fastest cache covert channels in the literature are the ones relying on shared memory such as Flush+Reload with 298 KBps and 0.00% error rate and Flush+Flush with 496 KBps and 0.84% error rate [12]. These implementations, while yielding a low error rate thanks to a retransmission protocol, have not been tested in a noisy environment, and require shared memory that is not available in most cloud environments, such as Amazon EC2.

Prime+Probe covert channels achieve between 67 KBps with 0.36% error rate [12] and 75 KBps with 1% error



(a) Image distortion caused by insertion and deletion errors due to scheduling.

(b) Noisy image after handling insertion and deletion errors.

(c) The image after applying error correction. It is equivalent to the original image.

Fig. 8: The stages of error correction.

rate [21]. Similarly, Pessl et al. [27] demonstrated a covert channel based on DRAM addressing that is up to 307 KBps with 1.8% error rate in native environment and 51 KBps with 4.1% error rate in virtualized environment. Compared to cache-based covert channels, this one has the advantage of working across CPUs, but its lack of synchronization mechanism at the protocol level causes a dramatic loss in performance in virtualized environments. However, these covert channels have neither been tested in a noisy environment nor in a real-world scenario such as Amazon EC2. By first optimizing our covert channel for speed, and then applying a protocol to deal with synchronization issues and cache activity, we demonstrate that we can obtain the same order of magnitude in speed as state-of-the-art covert channels. Moreover, in contrast to previous work, our covert channel is able to stay error-free and fast in the presence of extraordinary high system activity and has been evaluated between two virtual machines on Amazon EC2.

Previous work by Wu et al. [40] demonstrated a covert channel exploiting the memory bus on Amazon EC2 between two instances of 100 bps, *i.e.*, 0.01 KBps, with 0.75% error rate. In addition to being error-free, our covert channel thus outperforms previous covert channels demonstrated on Amazon EC2 by 3 orders of magnitude.

VII. SSH Over Cache Covert Channel

We demonstrated that our covert channel is robust against noise while still yielding good performance. This section gives a practical application of the covert channel that exploits both the speed and the error-free transmission: SSH and Telnet connections between two instances in the cloud.

A. Implementing TCP over Covert Channel

Current state-of-the-art covert channels cannot be used as an underlying layer for high-level protocols. Indeed, high-level protocols expect the physical layer to transmit the data without errors. Depending on the high-level protocol, corrupted bits can be repaired using error correction or retransmission. However, inserted or deleted bits corrupt the data stream and cannot be repaired by high-level protocols.

In the OSI model, which is typically used to model network layers, the data-link layer provides a mostly error-free data transmission for the higher communication layers. As our covert channel is error-free, *i.e.*, sent packets are always correct, we can implement higher level protocols using our covert channel as data-link layer. We decided on the ubiquitous *Transmission Control Protocol* (TCP) as the transport protocol. Being a core protocol of the Internet protocol suite, many applications rely on it. Supporting this protocol over a covert channel opens the field for practical applications, such as remote connections with SSH or Telnet.

So far, the communication over our covert channel is unidirectional. TCP requires a bidirectional data transmission on the data-link layer. A straightforward solution would be to run one covert channel per direction. However, this is not a viable option as an additional instance increases the noise drastically. Instead, we implement a half-duplex system. In this setting, one channel is used to transmit data alternately in both directions, *i.e.*, data cannot be sent in both directions simultaneously. These transmission details are transparent to TCP and have to be handled in the lower layers to be compatible with existing applications. To ensure practical use, arbitrary applications must be able to communicate via TCP over the covert channel without adaption.

Our solution is to provide local TCP sockets as covert channel endpoints. The client application initiates the communication by connecting to the client endpoint socket, just as it would connect to the target application. The client endpoint communicates with the server endpoint through the covert channel. The server endpoint connects itself to the server application. The endpoints buffer TCP packets for transmission while receiving data from the covert channel. The received data is directly sent to the connected application through the local TCP sockets. When the covert channel has sent all buffered data, it switches the transmission direction. Additional switches can be introduced at the cost of transmission rates to allow a more responsive connection.

Figure 9 illustrates the communication between the client and the server application. Due to the transparent TCP socket endpoints, the applications do not see a difference to being connected directly through a socket. One limitation of this method is that we cannot have multiple sockets simultaneously as the simple communication endpoint design does not support packet switching. However, for most applications it is sufficient to have one socket for communication. Especially remote shells, which are the most likely use case, do not require more than one socket connection.

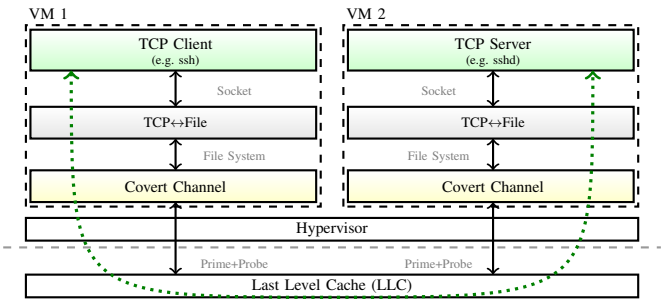


Fig. 9: TCP tunneling over the covert channel.

B. Evaluation of SSH Connections

As a practical proof-of-concept, we used an SSH client and server as application. We chose SSH as it is a well-known protocol for secure network services and remote login [45]. The protocol is also a good evaluation criterion for the covert channel's robustness and freedom from errors. Due to the cryptographic nature of the protocol, the connection is terminated as soon as there is a corrupted package, therefore our covert channel succeeds where previous work would fail.

We established an SSH connection over the covert channel in a native environment first. To evaluate the connection's stability, we executed common tasks, such as viewing files using `cat` or running more sophisticated programs such as `top`. For a more realistic scenario, we increased the level of noise on the computer. As a source of noise, we used Chrome to watch a music video on YouTube. Even in the presence of noise the SSH connection was stable and remained usable.

To evaluate a more realistic use case, we established the SSH connection between two co-located Amazon EC2 instances. To evaluate the practicability of the connection, we tried four common tasks: listing the files in the current directory (`ls`), viewing a text file (`cat /etc/passwd`), displaying the CPU utilization (`top`) and creating a file using `nano`. Without the presence of noise, the SSH connection remained stable over the whole time. We steadily increased the noise level by running an Apache2 web server on a third virtual machine, on the SSH server side and finally on all three virtual machines. Even the simulation of heavy noise on the third virtual machine using `stress -m 8` did not interrupt the SSH connection. Table VI summarizes the experiments. The delay between user input on the client side and the processing of the input on the server side is influenced on how frequent sender and receiver switch roles and can be configured to be in the range of milliseconds. The influence of noise on the input latency is barely perceivable.

C. Evaluation of Telnet Connections

Only after we started to generate artificial noise on the SSH server's virtual machine by running `stress -m 1`, we observed occasional disconnects due to corrupt packets or timeouts. In this scenario, we could still achieve a stable connection using Telnet instead of SSH. As Telnet is an unencrypted text-oriented protocol without authentication [28], there are fewer packets transmitted and the packets are not error-checked. Telnet allows stable communication even while running `stress -m 1` with occasional corrupted bytes. Table VII summarizes the experiments. Note that system modifications should not be executed over this connection as a random corruption of bytes might have devastating effects.

TABLE VI: Stability of SSH connection over the covert channel in the presence of noise, between two instances on Amazon EC2.

Noise	Connection
No noise	✓
`stress -m 8` on third VM	✓
Web server on third VM	✓
Web server on SSH server VM	✓
Web server on all VMs	✓
`stress -m 1` on server side	unstable

TABLE VII: Stability of Telnet connection over the covert channel in the presence of noise, between two instances on Amazon EC2.

Noise	Connection
No noise	✓
`stress -m 8` on third VM	✓
Web server on third VM	✓
Web server on SSH server VM	✓
`stress -m 1` on server side	✓

VIII. Conclusion

Caches are an ideal shared resource to establish covert channels between multiple virtual machines to exfiltrate sensitive data, as they are fast and shared by all tenants in public clouds. However, cache covert channels are prone to noise due to cache activity and scheduling, impairing their direct applicability in a real-world public cloud scenario.

In this paper we characterized the sources of noise and errors in cache covert channels comprehensively. We consequently designed a protocol that handles the physical layer and the data-link layer of a cache covert channel. We evaluated the performance of our covert channel in different scenarios in native and virtualized environments. Even in the presence of extraordinarily high system activity, we can maintain a transmission rate between 34.27 KBps and 45.09 KBps with an error rate of 0% on Amazon EC2 virtual machines, which is three orders of magnitude higher than previous covert channels on Amazon EC2. Based on this error-free covert channel, we built the first implementation of TCP through a cache covert channel. We verified the practical applicability of our error-free TCP connection by tunneling SSH and telnet connections reliably between two colocated Amazon EC2 virtual machines.

The possibility to establish reliable SSH connections and telnet sessions through the cache, moves cache covert channels beyond typical academic use cases, emphasizing the crucial importance of finding effective countermeasures.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement

No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

[1] T. Allan, B. B. Brumley, K. Falkner, J. V. D. Pol, and Y. Yarom, "Amplifying Side Channels Through Performance Degradation," *Cryptology ePrint Archive: Report 2015/1141*, 2015.

[2] J. M. Berger, "A note on error detection codes for asymmetric channels," *Information and Control*, vol. 4, no. 1, pp. 68–73, 1961.

[3] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Transactions on communications*, vol. 44, no. 10, pp. 1261–1271, 1996.

[4] C. A. Boano, M. A. Zuniga, K. Römer, and T. Voigt, "Jag: Reliable and predictable wireless agreement under external radio interference," in *IEEE 33rd Real-Time Systems Symposium (RTSS)*, 2012.

[5] B. B. Brumley, "Covert timing channels, caching, and cryptography," Ph.D. dissertation, 2011.

[6] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 307–320.

[7] J. Corbet, "(Nearly) full tickless operation in 3.10," https://lwn.net/Articles/549580/, May 2013.

[8] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[9] A. Fuchs and R. B. Lee, "Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs," in *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*, 2015.

[10] R. Gallager, "Low-density parity-check codes," *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA'16*, 2016.

[12] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA'16*, 2016.

[13] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[14] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P'11*, 2011.

[15] H. Hemmati, *Deep Space Optical Communications*, ser. JPL Deep-Space Communications and Navigation Series. Wiley, 2006. [Online]. Available: https://books.google.at/books?id=Cj52X7jK5OgC

[16] W.-M. Hu, "Lattice Scheduling and Covert Channels," in *S&P'92*, 1992.

[17] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES'16*, 2016.

[18] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," vol. 253665, 2014.

[19] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES," in *S&P'15*, 2015.

[20] M. Jeganathan and S. Mecherle, "A technical manual for focas 2.0—freespace optical communications analysis software," 1998.

[21] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P'15*, 2015.

[22] D. Magenheimer, "Tsc mode how-to," retrieved on August 13, 2016. [Online]. Available: http://xenbits.xen.org/docs/4.3-testing/misc/tscmode.txt

[23] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[24] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA'15*, 2015.

[25] H. Mercier, V. K. Bhargava, and V. Tarokh, "A survey of error-correcting codes for channels with symbol synchronization errors," *IEEE Communications Surveys & Tutorials*, vol. 1, no. 12, pp. 87–96, 2010.

[26] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005.

[27] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[28] J. Postel and J. Reynolds, "Telnet protocol specification," Internet Requests for Comments, RFC Editor, RFC 854, May 1983. [Online]. Available: https://tools.ietf.org/html/rfc854

[29] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[30] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS'09*, 2009.

[31] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows internals*. Pearson Education, 2012.

[32] W. Schmidt, M. Hanspach, and J. Keller, "A case study on covert channel establishment via software caches in high-assurance computing systems," *arXiv:1508.05228*, 2015.

[33] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[34] I. Shomorony and A. S. Avestimehr, "Is gaussian noise the worst-case additive noise in wireless networks?" in *International Symposium on Information Theory (ISIT'12)*, July 2012, pp. 214–218.

[35] S. Tolvanen, "Strictly Enforced Verified Boot with Error Correction," Jul. 2016, retrieved on August 9, 2016. [Online]. Available: http://android-developers.blogspot.co.at/2016/07/strictly-enforced-verified-boot-with.html

[36] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based Defenses against Cross-VM Side-channels," in *USENIX Security Symposium*, 2014.

[37] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A Placement Vulnerability Study in Multi-Tenant Public Clouds," in *USENIX Security Symposium*, 2015.

[38] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[39] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[40] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud," *IEEE/ACM Transactions on Networking*, 2014.

[41] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, 2011.

[42] Z. Xu, H. Wang, and Z. Wu, "A Measurement Study on Co-residence Threat inside the Cloud," in *USENIX Security Symposium*, 2015.

[43] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[44] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel Last-Level Cache," *Cryptology ePrint Archive, Report 2015/905*, 2015.

[45] T. Ylonen and C. Lonvick, "The secure shell (ssh) protocol architecture," Internet Requests for Comments, RFC Editor, RFC 4251, January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4251

[46] Y. Zhang and M. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *CCS'13*, 2013.