C5: Cross-Cores Cache Covert Channel

Clémentine Maurice[1], Christoph Neumann[1],
Olivier Heen[1], and Aurélien Francillon[2]

[1] Technicolor, Rennes, France,
[2] Eurecom, Sophia-Antipolis, France
{clementine.maurice,christoph.neumann,olivier.heen}@technicolor.com
aurelien.francillon@eurecom.fr

Abstract. Cloud computing relies on hypervisors to isolate virtual machines running on shared hardware. Since perfect isolation is difficult to achieve, sharing hardware induces threats. Covert channels were demonstrated to violate isolation and, typically, allow data exfiltration. Several covert channels have been proposed that rely on the processor's cache. However, these covert channels are either slow or impractical due to the *addressing uncertainty.* This uncertainty exists in particular in virtualized environments and with recent L3 caches which are using complex addressing. Using shared memory would elude addressing uncertainty, but shared memory is not available in most practical setups.
We build C5, a covert channel that tackles addressing uncertainty without requiring any shared memory, making the covert channel fast and practical. We are able to transfer messages on modern hardware across any cores of the same processor. The covert channel targets the last level cache that is shared across all cores. It exploits the inclusive feature of caches, allowing a core to evict lines in the private first level cache of another core. We experimentally evaluate the covert channel in native and virtualized environments. In particular, we successfully establish a covert channel between virtual machines running on different cores. We measure a bitrate of 1291bps for a native setup, and 751bps for a virtualized setup. This is one order of magnitude above previous cache-based covert channels in the same setup.

Keywords: Covert channel, Cache, Cross-VM, Virtualization, Cloud computing.

1 Introduction

Cloud computing leverages shared hardware to reduce infrastructure costs. The hypervisor, at the virtualization layer, provides isolation between the virtual machines. However, the last years have shown a great number of information leakage attacks across virtual machines, namely covert and side channels [13, 14, 23, 26–28, 30]. These attacks evidently violate the isolation.

Covert channels involve the cooperation of two attackers' processes to actively exchange information. Side channels imply passive observation of a victim's process by an attacker's process. Covert and side channels have been built in a native environment between two processes, and in a virtualized environment between two virtual machines. These attacks leverage elements of the microarchitecture that are accessible remotely by an attacker, such as the memory bus [26], the data cache [4, 21, 23, 28], the instruction cache [1, 30] or the branch target buffer [2].

In this article, we focus on covert channels. They are used to exfiltrate sensitive information, and can also be used as a co-residency test [29]. There are many challenges for covert channels across virtual machines. Core migration drastically reduces the bitrate of channels that are not cross-core [27]. Simultaneous execution of the virtual machines over several cores prevents a strict round-robin scheduling between the sender and the receiver [26]. Functions of address translation, and functions that map an address to a cache set are not exposed to processes, and thus induce uncertainty over the location of a particular data in the cache. This *addressing uncertainty* (term coined in [26]) prevents the sender and the receiver to agree on a particular location to work on.

Covert channels that don't tackle the *addressing uncertainty* are limited to use private first level caches, thus to be on the same core for modern processors [26]. This dramatically reduces the bitrate in virtualized environment or in the cloud, with modern processors that have several cores with a shared and physically indexed last level cache. Ristenpart et al. [23] target the private cache of a core and obtain a bitrate of 0.2bps, with the limitation that the sender and receiver must be on the same core. Xu et al. [27] quantify the achievable bitrate of this covert channel: from 215bps in lab condition, they reach 3bps in the cloud. This drop is due to the scheduling of the virtual machines across cores. Yarom and Falkner [28] circumvent the issue of physical addressing by relying on deduplication offered by the hypervisor or the OS. With deduplication, common pages use the same caches lines. However, deduplication can be deactivated for all or some specific security relevant pages, e.g., OpenSSL pages, making the previous attacks impractical. Wu et al. [26] avoid the shortcomings of caches by building a covert channel across cores that is based on the memory bus, using the lock instruction. We refer the reader to Section 6 for more details on state of the art techniques.

We differentiate our covert channel by tackling the issue of the *addressing uncertainty* without relying on any shared memory. Our covert channel works between two virtual machines that run across any cores of the same processor. We revisit the method of Ristenpart et al. [23], and take advantage of the shared and inclusive feature of the Last Level Cache in modern processors. We obtain a high bitrate, arguing that the covert channel is practical.

Contributions

In this paper, we demonstrate the information leakage due to a new covert channel that uses the last level cache.

1. We build the C5 covert channel across virtual machines, on modern hardware. In particular, we tackle the *addressing uncertainty* that severely limits the previous covert channels.

2. We analyze the interferences that are the root causes that enable this covert channel, *i.e.*, microarchitectural features such as the shared last level cache, and the inclusive feature of the cache hierarchy.
3. We evaluate the C5 covert channel in a native environment and achieve a bitrate of 1291bps (error rate: 3.1%). We evaluate the C5 covert channel in a virtualized environment and achieve a bitrate of 751bps (error rate: 5.7%). We explore the relation between the bitrate and the error rate.

The remainder of this paper is organized as follows. Section 2 covers background on the cache internals needed for the remainder of the article. Section 3 details our technique to build a cache-based covert channel. Section 4 exposes our experiments in lab-controlled native and virtualized setups. Section 5 discusses the factors that impact performance, as well as mitigations. Section 6 presents the related work and the differences with our work. Section 7 summarizes our results and their implications.

2 Background

In this section, we provide background notions on cache internals. We then review techniques that exploit cache interferences for the communication between two processes. Finally, we discuss the effect of virtualization and complex addressing on the cache addressing, and its impact on cache-based covert channels.

2.1 Cache fundamentals

The cache is faster than main memory and stores recently-used data. Intel processors[1] use a cache hierarchy similar to the one depicted in Figure 1 since the Nehalem microarchitecture (2008) and until the most recent Haswell microarchitecture [12]. There are usually three cache levels, called L1, L2 and L3. The levels L1 and L2 are private to each core, and store several kilobytes. The L3 cache is also called Last Level Cache (LLC in the rest of this paper). The LLC is divided into slices that are connected to the cores through a ring interconnect. The LLC is shared between the cores, *i.e.*, each core can address the entire cache. It is also the largest, usually several megabytes.

To read or write data in main memory, the CPU first checks the memory location in the L1 cache. If the address is found, it is a *cache hit* and the CPU immediately reads or writes data in the cache line. Otherwise, it is a *cache miss* and the CPU searches for the address in the next level, and so on, until reaching the main memory. A cache hit is significantly faster than a cache miss.

Data is transfered between the cache and the memory in 64 bytes blocks called *lines*. The location of a particular line depends on the cache structure. Today's caches are *n-way associative*, which means that a cache contains sets of

[1] In this article, we focus on Intel processors. Still, most of this discussion on caches applies also to other x86 processors.

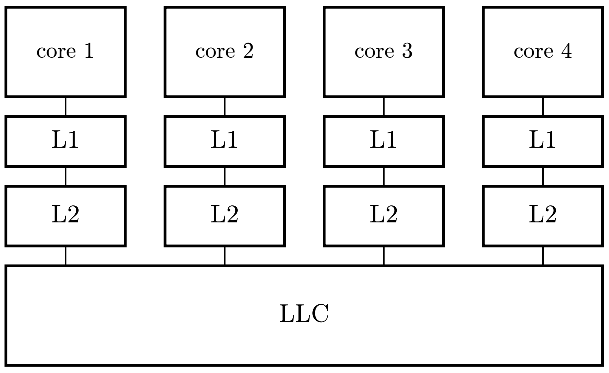
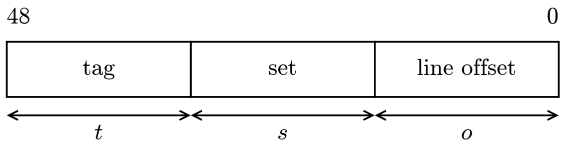


Fig. 1. Cache hierarchy of a quad-core Intel processor (since Nehalem microarchitecture). The LLC is inclusive, which means it is a superset of the L1 and L2 caches.

n lines. A line is loaded in a specific set depending on its address, and occupies any of the n lines.

With caches that implement a *direct addressing* scheme, memory addresses can be decomposed in three parts: the tag, the set and the offset in the line. The lowest o bits determine the offset in the line, with: $o = \log_2(\text{line size})$. The next s bits determine the set, with: $s = \log_2(\text{number of sets})$. And the remaining t bits form the tag. A 48 bit address can be represented as follows:



In contrast to direct addressing, some caches implement a *complex addressing* scheme, where potentially all address bits are used to index the cache. The function that maps an address to a set is not documented. This has important implications for covert channels.

The address used to compute the cache location can be either a physical or a virtual address. A *Virtually Indexed, Virtually Tagged* (VIVT) cache only uses virtual addresses to locate the data in the cache. Modern processors involve physical addressing, either *Virtually Indexed Physically Tagged* (VIPT), or *Physically Indexed Physically Tagged* (PIPT). The physical address is not known by the processes, *i.e.*, a process cannot know the location of a specific line for physically addressed caches. This too has important implications for covert channels.

When a cache set is full, a cache line needs to be evicted before storing a new cache line. When a line is evicted from L1 it is stored back to L2, which can lead to the eviction of a new line to LLC, etc. The replacement policy decides the victim line to be evicted. Good replacement policies choose the line that is the least likely to be reused. Such policies include Least Recently Used (LRU), Least Frequently Used, Pseudo Random, and Adaptive.

Depending on the cache design, data stored on one level may also be stored on other levels. A level is *inclusive* if it is a superset of the lower levels. To guarantee the inclusion property, when a line is evicted from the LLC, the line is also removed (invalidated) in the lower caches L1 and L2. A level is *exclusive* if a data is present at most once between this level and the lower levels. Intel CPUs from Nehalem to Haswell microarchitecture have exclusive L2 caches, and an inclusive LLC.

2.2 Playing with caches for fun and profit

Isolation prevents processes from directly reading or writing in the cache memory of another process. Cache-based covert and side channels use indirect means and side effects to transfer information from one process to another. One side effect is the variation of cache access delays.

Cache hits are faster than cache misses. This property allows monitoring access patterns, and subsequently leaking information. In *access-driven attacks*, a process monitors the time taken by its own activity to determine the cache sets accessed by other processes.

Two general strategies exist: *prime+probe* [21, 19, 24, 18] and *flush+reload* [8, 28]. With *prime+probe*, a receiver process fills the cache, then waits for a sender process to evict some cache sets. The receiver process reads data again and determines which sets were evicted. The access to those sets will be slower for the receiver because they need to be reloaded in the cache. With *flush+reload*, a receiver process flushes the cache, then waits for a sender process to reload some cache sets. The receiver process reads data again and determines which sets were reloaded. The access to those sets will be faster for the receiver because they don't need to be reloaded in the cache. The *flush+reload* attack assumes shared lines of cache between the sender and the receiver – and thus shared memory – otherwise the sets reloaded by the sender will not be faster to reload by the receiver than the evicted ones. Indeed, the receiver cannot access sets reloaded by the sender if they don't share memory.

2.3 The problem of *addressing uncertainty*

The previous attacks rely on the fact that it is possible to target a specific set. However, two conditions individually create uncertainty on the addressing, making it difficult to target a specific set: virtualization and complex addressing.

Processors implement virtual memory using a Memory Management Unit (MMU) that maps virtual addresses to physical addresses. With virtual machines, hypervisors introduce an additional layer of translation, known as Extended Page Tables on Intel processors. The guest virtual pages are translated to the guest physical pages, and further to the actual machine pages. The hypervisor is responsible for mapping the guest physical memory to the actual machine memory. A process knowing a virtual address in its virtual machine has no way of learning the corresponding physical address of the guest, nor the actual machine address. In a native environment, the layer of translation from virtual to physical

Table 1. Characteristics of the CPUs found on Amazon EC2 [3, 7, 20].

Model	**Microarch**	**Year**	**Cores**	**LLC**	**Potential for C5**
Opteron 270	K8	2005	2	private L2 exclusive	not cross-core
Opteron 2218 HE	K8	2007	2	private L2 exclusive	not cross-core
Xeon E5430	Core	2007	4	2×6MB L2 non-inclusive	not cross-core
Xeon E5507	Nehalem	2010	4	**shared** 4MB L3 **inclusive**	✓
Xeon E5645	Nehalem	2011	6	**shared** 12MB L3 **inclusive**	✓
Xeon E5-2670	Sandy Bridge	2012	8	**shared** 20MB L3 **inclusive**	✓
Xeon E5-2670 v2	Ivy Bridge	2013	10	**shared** 25MB L3 **inclusive**	✓
Xeon E5-2666 v3	Haswell	2014	9	**shared** 25MB L3 **inclusive**	✓

addresses does not create uncertainty on the set if both processes allocate large portions of aligned and contiguous memory[2]. In a virtualized environment, the additional layer of translation does create uncertainty, as the alignment is not guaranteed.

In addition to this, the complex addressing scheme maps an address to a set with a function that potentially uses all address bits. As the function is undocumented, a process cannot determine the set in which it is reading or writing. Even aligned memory does not guarantee that two processes will target the same set.

This has implications for the design of covert channels. Indeed, with the *addressing uncertainty*, two processes without any shared memory cannot directly agree on a set to work on.

3 C5 Covert Channel

Our covert channel relies on the fact that the LLC is shared and inclusive. Those two characteristics are present in all CPUs from Nehalem to Haswell microarchitectures, *i.e.*, all modern Intel CPUs, including most CPUs that are found in, e.g., Amazon EC2 (Table 1).

The sender process sends bits to the receiver by varying the access delays that the receiver observes when accessing a set in the cache. At a high level view, the covert channel encodes a '0' as a fast access for the receiver and a '1' as a slow access. In this sense, our covert channel strategy is close to *prime+probe*.

Figure 2 illustrates our covert channel. The receiver process repeatedly probes one set. If the sender is idle (a '0' is being transmitted), the access is fast because the data stays in the private L1 cache of the receiver, see Figure 2-1. The data is also present in the LLC because of its *inclusive* property.

To send a '1', the sender process writes data to occupy the whole LLC, see Figure 2-2; in particular this evicts the set of the receiver from the LLC. Because of the *inclusive* property, the data also gets evicted from the private L1 cache of the receiver. The receiver now observes that the access to its set is slow; the data must be retrieved from RAM, see Figure 2-3.

We now provide a detailed description of the sender and the receiver.

[2] It cannot be guaranteed using `malloc` on 4kB pages, but it is possible to use huge pages of 2MB or 1GB if the CPU supports it.

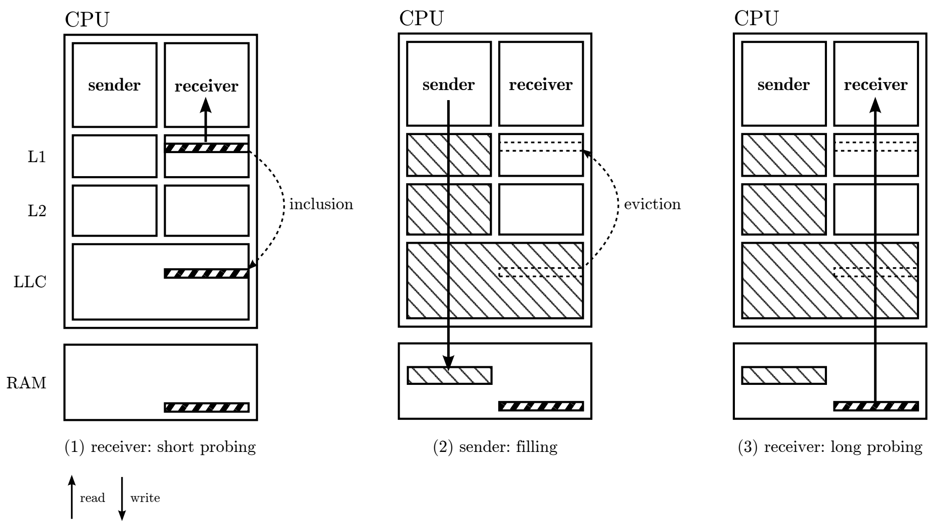


Fig. 2. Cross-core covert channel illustration of sender and receiver behavior. Step (1): the receiver probes one set repeatedly; the access is fast because the data is in its L1 (and LLC by inclusive feature). Step (2): the sender fills the LLC, thus evicting the set of the receiver from LLC and its private L1 cache. Step (3): the receiver probes the same set; the access is slow because the data must be retrieved from RAM.

3.1 Sender

The sender needs a way to interfere with the private cache of the other cores. In our covert channel, the sender leverages the inclusive feature of the LLC (see Section 2.1). As the LLC is shared amongst the cores of the same processor, the sender may evict lines that are owned by other processes, and in particular processes running on other cores.

A straightforward idea is that the sender writes in a set, and the receiver probes the same set. However, due to virtualization and complex addressing, the sender and the receiver cannot agree on the cache set they are working on (see Section 2.3). Our technique consists of a scheme where the sender flushes the whole LLC, and the receiver probes a single set. That way, the sender is guaranteed to affect the set that the receiver reads, thus resolving the *addressing uncertainty*.

In order to flush the whole LLC, the sender must evict cache lines and therefore writes data into a buffer. In fact, either writing or reading data would provoke a cache miss. We choose to write because a read miss following a write induces a higher penalty for the receiver than a read miss following a read. This leads to a stronger signal. We further discuss the influence of this choice in Section 5.

We leverage the replacement policy within a set to evict lines from the LLC. The replacement policy and the associativity influence the buffer size b of the

Algorithm 1 Sender: $f(n, o, s, c, w)$

```
message ← {0,1}*
$n$ ← LLC associativity
$o$ ← $\log_2$(line size)
$s$ ← $\log_2$(number of sets in LLC)
$b$ ← $n \times 2^{\mathrm{o+s}} \times c$
buffer[$b$]
**for each** bit in message **do**
    wait($w$)
    **if** bit == 1 **then**
        **for** $i = 0$ to number of sets **do**
            **for** $j = 0$ to $n \times c$ **do**
                buffer[$2^o i + 2^{\mathrm{o+s}} j$] = constant
            **end for**
        **end for**
    **end if**
**end for**
```

sender. Considering a pure LRU policy, writing n lines in each set is enough to flush all the lines of the LLC, n being the associativity. The replacement policies on modern CPUs drastically affect the performance of caches; therefore they are well guarded secrets. Pseudo-LRU policies are known to be inefficient for memory intensive workloads of working sets greater than the cache size. Adaptive policies [22] are more likely to be used in actual processors. Since the actual replacement policy is unknown, we determine experimentally the size b of the buffer to which the sender needs to write.

The order of writes into the buffer is highly dependent on the cache microarchitecture. Ideally, to iterate over the buffer we would take into account the function that maps an address to a set. However this function is undocumented, thus we assume a direct addressing; other types of iterations are possible. The sender writes with the following memory pattern $2^o i + 2^{o+s} j$ as described in Algorithm 1. 2^s is the number of sets of the LLC and 2^o the line size; j and i are line and set indices respectively.

Algorithm 1 summarizes the steps performed by the sender. The parameters are the LLC associativity n, the number of sets 2^s, the line size 2^o, and a constant c to adapt the buffer size. To send a '1', the sender flushes the entire LLC by writing in each line j ($n \times c$ times) of each set i, with the described memory pattern. To send a '0', the sender does nothing. The sender waits for a determined time w before sending a bit to allow the receiver to distinguish between two consecutive bits.

3.2 Receiver

The receiver repeatedly probes all the lines of the same cache set in its L1 cache. Algorithm 2 summarizes the steps performed by the receiver. The iteration is dependent on the cache microarchitecture. To access each line i (n

Algorithm 2 Receiver: $f(n, o, s)$

```
$n \leftarrow$ L1 associativity
$o \leftarrow \log_2$(line size)
$s \leftarrow \log_2$(number of sets in L1)
buffer[$n \times 2^{o+s}$]
**loop**
    read $\leftarrow 0$
    begin measurement
    **for** $i = 0$ to $n$ **do**
        read $+ =$ buffer[$2^{o+s}i$]
    **end for**
    end measurement, record ($localTime, accessDelay$)
**end loop**
```

times) of the same set, the receiver reads a buffer – and measures the time taken – with the following memory pattern: $2^{o+s}i$. The cumulative variable `read` prevents optimizations from the compiler, by introducing a dependency between the consecutive loads so that they happen in sequence and not in parallel. In the actual code, we also unroll the inner `for` loop to reduce unnecessary branches and memory accesses.

The receiver is able to probe a set in its L1 cache because the L1 is virtually indexed, and does not use complex addressing. We do not seek to probe the L2 or L3, because all read and write accesses reach the L1 first and they might evict each other, creating differences in timing that are not caused by the sender.

The receiver probes a single set when the sender writes to the entire cache, thus one iteration of the receiver is faster than one iteration of the sender. The receiver runs continuously and concurrently with the sender, while the sender only sends one bit every w microseconds. As a consequence, the receiver performs several measurements for each bit transmitted by the sender.

One measurement of the receiver has the form ($localTime$, $accessDelay$), where $localTime$ is the time of the end of one measurement according to the local clock of the receiver and $accessDelay$ is the time taken for the receiver to read the set. Figure 3 illustrates the measurements performed by the receiver.

Having these measurements, the receiver decodes the transmitted bit-sequence. First, the receiver extracts all the '1's. The receiver removes all points that have an $accessDelay$ below (or equal to) typical L2 access time. Then the receiver only keeps the $localTime$ information and applies a clustering algorithm to separate the bits. We choose DBSCAN [6], a density-based clustering algorithm, over the popular k-means algorithm. A drawback of the k-means algorithm is that it takes the number k of clusters as an input parameter. In our case, it would mean knowing in advance the number of '1's, which is not realistic. The DBSCAN algorithm takes two input parameters, $minPts$ and ϵ:

1. $minPts$: the minimum number of points in each cluster. If $minPts$ is too low, we could observe false positives, reading a '1' when there is none; if

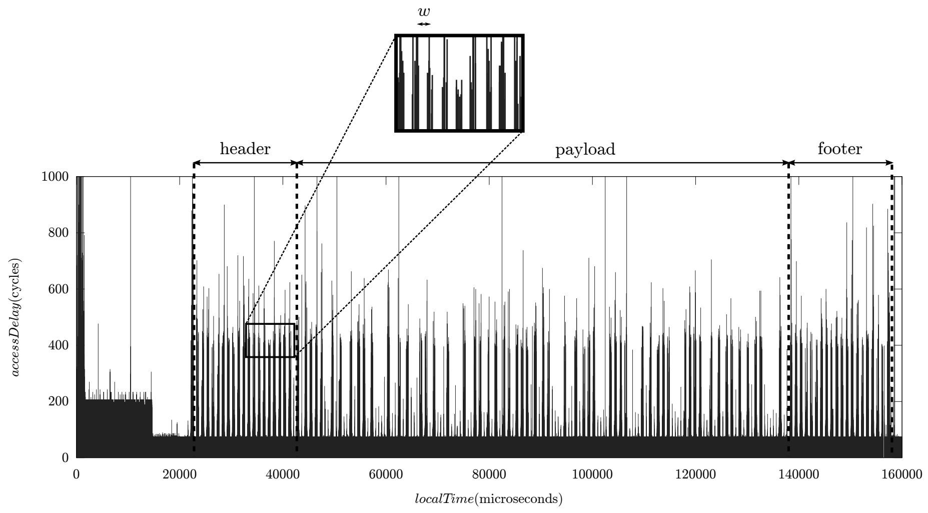


Fig. 3. Reception of a 128-bit transmission. *Laptop* setup in native environment, with $w = 500\mu s$, $b = 3MB$.

minPts is too high, we could observe false negatives, not reading a '1' when there is one. In practice, we use *minPts* between 5 and 20.

2. ϵ: if a point belongs to a cluster, every point in its ϵ-neighborhood is also part of the cluster. In practice, we choose ϵ below w.

Once all the '1's of the transmitted bit-sequence have been extracted, the receiver reconstructs the remaining '0's. This step is straightforward as the receiver knows the time taken to transmit a '0' which is w.

4 Experiments

In this section, we evaluate the C5 covert channel on native and virtualized setups. We then illustrate the effect of the complex addressing regarding *addressing uncertainty*.

4.1 Testbed

Table 2 summarizes the characteristics of the *laptop* and *workstation* setups. Some parameters of the architecture are constant for the considered processors. The line size in all cache hierarchy is 64 bytes, and the L1 is 8-associative and has 64 sets. We conduct our experiments in lab-controlled native and virtualized environments.

We adjust two parameters: the size b of the buffer that evicts the LLC, and the delay w between the transmission of two consecutive bits. The size b and the delay w impact the bitrate and the error rate of the clustering algorithm, as

Table 2. Experimental setups, LLC characteristics.

Name	**Model**	**Microarch**	**Cores**	**Size**	**Sets**	**Asso.**	**Complex addressing**
laptop	i5-3340M	Ivy Bridge	2	3MB	4096	12	yes
workstation	Xeon E5-2609v2	Ivy Bridge	4	10MB	8192	20	yes

depicted in Figures 4 and 5. The precision of the clustering algorithm increases with the size b, however the bitrate is proportionally reduced. The size b is controlled by the multiplicative parameter c and must be at least the size of the LLC. The bitrate increases with lower values of w, but the precision of the clustering algorithm decreases.

To evaluate the covert channel, the sender transmits a random 4096-bit message to the receiver. We transmit series of 20 consecutive '1's as a header and a footer framing the payload to be able to extract it automatically. The receiver then reconstructs the message from its measurements. We run 10 experiments for each set of parameters, and calculate the bitrate and the error rate. We derive the error rate from the Levenshtein distance between the sent payload and the received payload. The Levenshtein distance is the minimum number of characters edits and accounts for insertions, deletions and bit flips. We provide the evaluation results for each environment: native in Section 4.2 and virtualized in Section 4.3.

Establishing cache-based channels demands fine grained measurements. Processors provide a timestamp counter for the number of cycles since reset. This counter can be accessed by the `rdtsc` and `rdtscp` instructions. However, reading the counter is not sufficient as modern processors use out-of-order execution. The actual execution may not respect the sequence order of instructions as written in the executable. In particular, a reordering of the `rdtsc` instruction can lead to the measurement of more, or less, than the wanted sequence. We prevent reordering by using serializing instructions, such as `cpuid`. We follow Intel recommendations for fine-grained timing analysis in [11].

4.2 Native environment

We evaluate C5 in the *laptop* and *workstation* setups, in a native (non-virtualized) environment. We run the sender and the receiver as unprivileged processes, in Ubuntu 14.04. To demonstrate the cross-core property of our covert channel, we pin the sender and the receiver to different cores[3]. Figure 3 illustrates a transmission of 128 bits in the *laptop* setup, for $w = 500\mu s$ and $b = 3MB$.

Figure 4 presents the results in the *laptop* setup, for two values of b, and three values for waiting time w. For $b = 3MB$ (the size of the LLC), varying w we obtain a bitrate between 232bps and 1291bps. The error rate is comprised between 0.3% (with a standard deviation $\sigma = 3.0 \times 10^{-3}$) and 3.1% ($\sigma = 0.013$). When we increase b to 4.5MB, the bitrate slightly decreases but stays in the

[3] Using the `sched_setaffinity(2)` Linux system call.

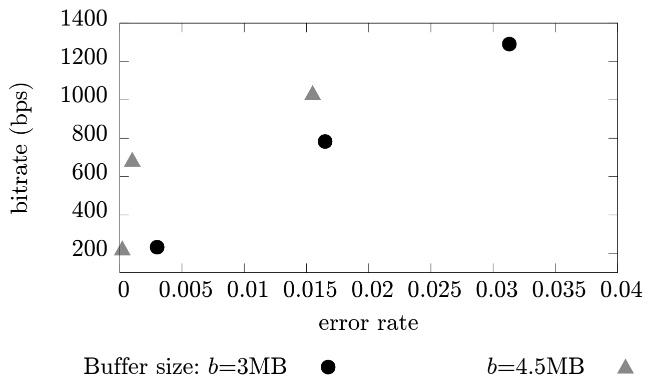


Fig. 4. Bitrate as a function of the error rate, for two sizes of buffer ***b***. *Laptop* setup (3MB LLC) in native environment.

same order of magnitude, between 223bps and 1033bps. The error rate decreases between 0.02% ($\sigma = 8.5 \times 10^{-5}$) and 1.6% ($\sigma = 1.1 \times 10^{-4}$). The standard deviation of the error rate also decreases, leading to more reliable transmission. We conclude that it is sufficient to write n lines per set, but that the transmission is more reliable if we write more than n lines. This is a tradeoff between the bitrate and the error rate.

In the *workstation* setup, we obtain a bitrate of 163bps for $b = 15$MB ($1.5 \times$ LLC), for an error rate of 1.9% ($\sigma = 7.2 \times 10^{-3}$). As expected, we observe that when the size of the LLC increases the bitrate decreases, since it takes longer to send a '1'. Compared to the *laptop* setup, the error rate and the standard deviation have also increased. There are two factors that can explain these results. First, the ratio of the associativity over the number of cores is smaller in the *workstation* setup, which means that lines have a greater probability of being evicted by processes running in other cores, leading to a higher error rate. Second, the LLC is bigger in the *workstation* setup, which means that the allocation of a buffer might not cover all the sets of the LLC, leading to a difference in the error rate between runs, and thus a higher standard deviation.

4.3 Virtualized environment

We evaluate C5 in the *laptop* setup, using Xen 4.4 as hypervisor. We run the sender as an unprivileged process in a guest virtual machine, and the receiver as an unprivileged process in another guest virtual machine. The guests and dom0 run Ubuntu 14.04. Each guest has one vCPU, and dom0 uses the default algorithm to schedule guest virtual machines.

Figure 5 presents the results for two values of b ($b = 3$MB and $b = 4.5$MB), and two waiting time w ($w = 4000$µs and $w = 1000$µs). For $b = 3$MB (the size of the LLC), varying w we obtain a bitrate between 229bps and 751bps. When we increase b to 4.5MB, the bitrate goes from 219bps to 661bps. There is a

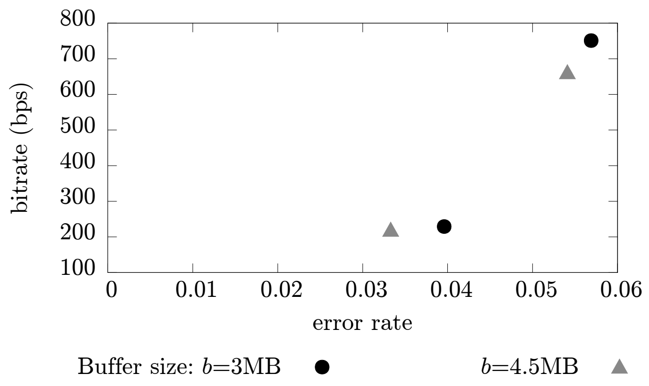


Fig. 5. Bitrate as a function of the error rate, for two sizes of buffer ***b***. *Laptop* setup (3MB LLC) in virtualized environment.

performance degradation compared to the native setup, but the bitrate stays in the same order of magnitude. The error rate is slightly higher than in the native setup, between 3.3% ($\sigma = 0.019$) and 5.7% ($\sigma = 8.8 \times 10^{-3}$), and is comparable for the two values of b. The standard deviation of the error rate is also higher than in the native setup, and is higher for a low value of b.

4.4 Complex addressing matters

We illustrate the impact of complex addressing on the cache-based covert channel proposed in the first part of [26]. The sender accesses a sequence of lines that map the memory pattern $M + X \cdot 256\text{k}$. The pattern is crafted such as to change the bits that correspond to the tag of the memory address. All accesses following this pattern target the same set. The receiver measures the access latency of another sequence of lines that follow the same pattern. The receiver then accesses the same set than the sender, and observes a longer access when the sender has flushed the lines. The authors of [26] evaluate the covert channel with a Core 2 Q8400 processor, which does not use a complex addressing scheme.

We reproduce this covert channel in the *laptop* setup in a native environment. The microarchitecture of the *laptop* CPU is more recent than the Q8400 processor, and the LLC uses complex addressing. We align the memory allocated by the sender and the receiver, and access the same memory pattern. As expected, we observe that the receiver is unable to retrieve the message. This is due to the complex addressing, causing the sender and the receiver to access different sets, removing the interference needed to establish a covert channel.

5 Discussion

In this section we expose the factors that impact – positively or negatively – the performances of the C5 covert channel. We also discuss the effects of a number of existing mitigation methods.

5.1 Performance

Evicting cache lines by reading or writing memory modifies the quality of the signal. A read is generally less costly than a write, so the sender could prefer to perform reads instead of writes in our covert channel. However, reads do not have the same impact in terms of timing for the receiver. When the receiver loads a new cache line, there are two cases. In the first case, the line had previously only been read by the sender, and not modified. The receiver thus requires a single memory access to load the line. In the second case, the line had previously been modified by the sender. At the time of eviction, it needs to be written back in memory. This requires an additional memory access, and thus takes longer. We choose to evict caches lines using memory writes because of the higher latency, which improves the signal quality.

Whenever the sender and the receiver run on the same core, the C5 covert channel may benefit from optimizations. In this case, there is no need to flush the entire LLC: flushing the L1 cache that is 32kB is sufficient and faster. However, the sender and the receiver are scheduled by the OS or the hypervisor, and frequently run on different cores [27]. We would need a method to detect when both run on the same core, and adapt the sender algorithm accordingly. Our method is simpler as it is agnostic to the scheduler.

The detection algorithm of the receiver impacts the overall bitrate. We put the priority on having a good detector at the receiver end, to minimize the error rate. In our implementation, the sender is waiting between the transmission of consecutive bits. The receiver uses a density-based clustering algorithm to separate the bits. Further work can be dedicated to reduce or eliminate the waiting time on the sender side, by using a different detection algorithm on the receiver side.

The C5 channel depends on the cache design. In particular, it depends on the shared and inclusive LLC. We believe this is a reasonable assumption, as this is the case since several generations of microarchitectures at Intel. The caches of AMD processors have historically been exclusive, and our covert channel is likely not to work with these processors. However, inclusive caches seem to be a recent trend at AMD. Indeed, they were recently introduced in the new low-power microarchitecture named Jaguar. As Wu et al. [26] note, cache-based covert channels also need to be on the same processor. On a machine with two processors, two virtual machines are on average half the time on the same processor. Such a setting would introduce transmission errors. These errors may be handled by implementing error correcting codes or some synchronization between the receiver and the sender. In any case the bitrate of our covert channel would be reduced.

In a public cloud setup such as on Amazon EC2, several factors may impact the performance and applicability of the C5 covert channel. First, the sender and receiver must be co-resident, *i.e.*, run on the same hardware and hypervisor despite the virtual machine placement policy of the cloud provider. Ristenpart et al. [23] showed that it is possible to achieve co-residency on Amazon EC2. A test is also required to determine co-residency; the assigned IP addresses, the round-trip times [23] and clock-skews [16, 17] help establish if two machines are co-resident or not. Second, the hypervisor, and in particular the associated vCPU scheduler, may have an impact on performances. Amazon EC2 relies on a customized version of Xen. The exact scheduler used is unknown. As a consequence, the results obtained in our lab experiments using Xen cannot be translated as is to a cloud setting. We expect the performance to be degraded in a cloud environment.

Similarly, we expect the error rate to increase in presence of a high non-participating workload, as it is the case with other cache covert channels [27]. The resulting error rate depends on the memory footprint of the workload, the core on which it executes, and on its granularity of execution compared to the transmission delay of the message.

5.2 Mitigation

Several papers focus on mitigating cache-based side channels, at the software and hardware levels. We review some of these solutions to determine if they also apply to the mitigation of covert channels.

Domnister et al. [5] propose to modify the replacement policy in the cache controller. Their cache design prevents a thread from evicting lines that belong to other threads. Although they state that L2/L3 attacks and defense are out of the scope of their paper, if the policy is applied to the LLC, the sender cannot evict all lines of the receiver, so it may partially mitigate our covert channel too. However, the performance degradation of the method on L1 cache is about 1% on average, up to 5% on some benchmarks. The mitigation might impact even more the performances if done also on the LLC. Wang and Lee [25] propose two new cache architectures. The Partition Locked Cache avoid cache interference by locking cache lines, and preventing processes from evicting cache lines of sensitive programs. Changes to the system are necessary to manage which lines should be locked, and would target specific programs, so it may not mitigate our covert channel. The Random Permutation Cache randomizes the interferences such that information about cache timings is useless to the attacker. It is done by creating permutations tables so that the memory-to-cache-sets mapping are not the same for sensitive programs as for others. However, our covert channel is agnostic to this mapping since we target the whole cache, so this solution may not mitigate our covert channel. These hardware-based solutions are currently not implemented.

Zhang et al. [31] designed a countermeasure for side channels in the cloud. The guest VM itself repeatedly cleans the L1 cache. This introduces noise on the timing measurements of the attacker, thus rendering them useless. As the mitigation is only performed on the L1 cache, it may not mitigate our covert channel that exploits the LLC. Furthermore, applying this countermeasure to the whole cache hierarchy would lead to an important performance penalty, as this would nullify the purpose of the cache.

The above mitigations do not take into account the specificities of the C5 covert channel. Exclusive caches, generally used by AMD processors, mitigate our covert channel as it prevents the LLC from invalidating sets of private L1 caches. Other mitigations might be implemented in the context of virtualization. Similar to Kim et al. [15], the hypervisor can partition the LLC such that each virtual machine works on dedicated sets within the LLC. This way the sender cannot evict the lines of the receiver that is running in a different virtual machine. Of course these mitigations might degrade the overall performance of the system. These mitigations are subject of future work.

6 Related work

Covert channels using caches have been known for a long time. Hu [9] in 1992 is the first to consider the use of cache to perform cross-process leakage via covert channels. Covert channels in the cloud were introduced by Ristenpart et al. [23] in 2009, and were thus performed on older generations of processors. In particular, it was not possible to perform a cross core channel using the cache. Ristenpart et al. built a covert channel for Amazon EC2, based on L2 cache contention that uses a variant of *prime+probe* [21, 19]. Despite its low bitrate of 0.2bps, this covert channel shows deficiencies in the isolation of virtual machines in Amazon EC2. However, this covert channel has some limitations: the sender and receiver must synchronize and share the same core. Xu et al. [27] quantify the achievable bit rate of such a covert channel: they reach 215bps in lab condition, but only 3bps in the cloud. The dramatic drop is due to the fact that the covert channel does not work across cores, and thus the channel design has to take into account core migration. In contrast with these works, we leverage the properties of modern hardware to build a covert channel that works across cores.

To make cache-based covert channels across cores in a virtual environment, the protocol has to resolve or bypass the issues brought by *addressing uncertainty*. Wu et al. [26] observe that the data transmission scheme has to be purely time-based. This contrasts with the covert channel designed by Percival [21] for a native environment that used cache regions to encode information. To illustrate the time-based transmission scheme, Wu et al. propose a cache-based covert channel for which the sender and receiver are not scheduled in a round-robin fashion, but simultaneously. However, the sender and receiver have to agree on a set to work on, which ignores the addressing issue. Their experiment has been tested on a non-virtualized environment, and on a CPU with an older microarchitecture that does not feature complex addressing. They further assume that cache-based covert channels are impractical due to the need of a shared cache. However, modern processors – including those used by Amazon – have all the right properties that make cache-based covert channels practical, and thus this assumption needs to be revisited. Moreover, complex addressing on the LLC is now a common feature. The main contribution of Wu et al. is a new covert channel that is based on the memory bus, using the lock instructions, that works across processors. Their experiment performed in the cloud obtains a bitrate of over 100bps.

To bypass the *addressing uncertainty*, Yarom and Falkner [28] rely on deduplication offered by the hypervisor. With deduplication, common pages use the same cache lines. They build a side channel on the GnuPG implementation of RSA and extract more than 90% of the key in a cross-VM attack. They use the `clflush` instruction that flushes a line from the whole cache hierarchy, and also exploit the inclusive feature of LLC caches. This attack has also been used to target AES in a cross-VM setting [14]. However, using deduplication imposes constraints on the platform where the attack can be performed. For instance, to the best of our knowledge, the Xen version used in Amazon EC2 does not allow deduplication. Thus the attacks [28, 14] do not work on Amazon EC2. In contrast with these papers, we tackle the *addressing uncertainty* without any shared memory. Hund et al. [10] resolve the addressing uncertainty by reverse engineering the function that maps a physical address to a slice in order to circumvent the kernel space ASLR. While this is a first step to resolve the addressing uncertainty brought by complex addressing on modern processors, the authors only reversed the function for a given Sandy Bridge processor. It is unknown if the function differs for processors of the same micro-architecture, or for processors of different micro-architecture. Our covert channel is agnostic to this function, hence it applies to a large range of modern processors.

Most covert channels are used in offensive scenarios. Zhang et al. [29] propose to use cache covert channels in a defensive scenario. The goal is to detect the co-residency of foe virtual machines on a physical machine that is supposed to be exclusively owned by a user. The user coordinates its VMs to silence them, avoiding using portions of the cache.

7 Conclusion

Virtualized setups are becoming ubiquitous with the adoption of cloud computing. Moreover, modern hardware tends to increase the number of cores per processor. The cross-core and cross virtual machines properties become mandatory for covert channels. In this paper, we built the C5 covert channel that transfers messages across different cores of the same processor. Our covert channel tackles *addressing uncertainty* that is in particular introduced by hypervisors and complex addressing. In contrast to previous work, our covert channel does not require any shared memory. All these properties make our covert channel fast and practical.

We analyzed the root causes that enable this covert channel, *i.e.*, microarchitectural features such as the shared last level cache, and the inclusive feature of the cache hierarchy. We experimentally evaluated the covert channel in native and virtualized environments. We successfully established a covert channel between virtual machines despite the CPU scheduler of the hypervisor. We measured a bitrate one order of magnitude above previous cache based covert channels in the same setup.

Future work will investigate specific countermeasures against the C5 covert channel. Countermeasures should be investigated at different levels, ranging from the microarchitectural features of processors to the memory management of the hypervisor.

References

1. O. Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *Proceedings of the 1st ACM Computer Security Architecture Workshop (CSAW'07)*, 2007.
2. O. Acıiçmez, J.-P. Seifert, and c. K. Koç. Predicting secret keys via branch prediction. In *CT-RSA 2007*, 2007.
3. Amazon Web Services. Amazon EC2 Instances. `https://aws.amazon.com/ec2/instance-types/`. Retrieved April 21, 2015.
4. D. J. Bernstein. Cache-timing attacks on AES. Technical report, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.
5. L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2011.
6. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, 1996.
7. B. Farley, V. Varadarajan, K. D. Bowers, A. Juels, T. Ristenpart, and M. M. Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)*, 2012.
8. D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P'11*, 2011.
9. W.-M. Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–61, 1992.
10. R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, 2013.
11. Intel. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper, 2010.
12. Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2014.
13. G. Irazoqui Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Fine grain Cross-VM Attacks on Xen and VMware are possible! *Cryptology ePrint Archive, Report 2014/248*, 2014.
14. G. Irazoqui Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14*, 2014.
15. T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, 2012.
16. T. Kohno, A. Broido, and K. Claffy. Remote Physical Device Fingerprinting. In *IEEE Transactions on Dependable and Secure Computing*, volume 2, pages 93–108, 2005.

17. S. J. Murdoch. Hot or Not: Revealing Hidden Services by their Clock Skew. In *CCS'06*, 2006.
18. M. Neve and J.-P. Seifert. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th international conference on Selected areas in cryptography (SAC'06)*, 2006.
19. D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA 2006*, 2006.
20. Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *HotCloud'12*, 2012.
21. C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.
22. M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381, 2007.
23. 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.
24. E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, July 2010.
25. Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494, June 2007.
26. Z. Wu, Z. Xu, and H. Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.
27. 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.
28. Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.
29. Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P'11*, 2011.
30. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS'12*, 2012.
31. Y. Zhang and M. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS'13*, 2013.