One Covert Channel to Rule Them All: A Practical Approach to Data Exfiltration in the Cloud

Benjamin Semal
Information Security Group
Royal Holloway University of London
Egham, United Kingdom
benjamin.semal.2018@live.rhul.ac.uk

Konstantinos Markantonakis
Information Security Group
Royal Holloway University of London
Egham, United Kingdom
k.markantonakis@rhul.ac.uk

Keith Mayes
Information Security Group
Royal Holloway University of London
Egham, United Kingdom
keith.mayes@rhul.ac.uk

Jan Kalbantner
Information Security Group
Royal Holloway University of London
Egham, United Kingdom
jan.kalbantner.2018@live.rhul.ac.uk

***Abstract*—The sharing of hardware platforms in multi-tenant environments is a growing security concern. Microarchitectural timing-based covert channels allow tunneling information out of a compromised cloud instance, thus bypassing information flow policies. Significant research efforts have been carried out in order to address the super-set of timing channels. Nevertheless, new attacks keep on being published while disregarding the latest academic efforts, arguing that the relevant defences have not yet been deployed. In order to bridge the gap between vulnerabilities and countermeasures, we challenge state-of-the-art mitigation techniques by constructing the first cross-VM covert channel that is resilient against all known defences, whether they are already deployed or still theoretical. Defence strategies that are relevant with covert channels are surveyed, and a list of requirements is constructed for the new attack. Then, we re-visit the exploitation of the x86 memory bus lock, and launch the proposed covert communication channel across two AWS EC2 instances. While simple in design, the proposed implementation shows that x86 microarchitectures still present salient vulnerabilities, and that state-of-the-art defence strategies—even theoretical ones—remain unsuccessful at hindering data leakage in multi-tenant environments. Finally, a strategy to mitigate the remaining vulnerability is suggested, along with a comparison against the ARMv8 processor architecture.**

***Index Terms*—Covert channel, cloud security, data confidentiality**

I. Introduction

Microarchitectural timing-based attacks are software-launched exploits which leverage the sharing of a processor among multiple tenants, in order to compromise sensitive information. These attacks can either take the form of a side channel, where the victim is accidentally leaking information, or the form of a covert channel, where the attacker has infected the victim with a malicious sending-end that deliberately transmits information. Microarchitectural covert and side channels have been increasingly popular in the last decade, and even more since the release of the Spectre and Meltdown attacks [1], [2]. In response, a plethora of mitigation strategies has been proposed by academics, from new hardware designs through software partitioning to anomaly detection.

These defence strategies often aim at closing a Prime+Probe [3] covert or side channel, omitting attacks which are not based on cache exploitation. Unfortunately, authors of defence strategies rarely challenge their proposal with all the artifacts available to an adversary, such as shared buses, interconnects, and system-level resources (e.g. DRAM). In parallel, new attacks consider a set of countermeasures, usually the ones already deployed in the targeted environment, and aim at demonstrating a residual threat despite these existing countermeasures. A trend that we observe is that attacks often forget to take into account the latest developments in terms of defences, arguing that these are not deployed by OS or cloud providers. Therefore, it is difficult to assess the novelty of these attacks, as they might already have been addressed by recent works.

With the intent of bringing coherence in this cat-and-mouse game, we perform a retrospective analysis on state-of-the-art attack and defence techniques. More specifically, we propose a microarchitectural covert channel that allows cross-VM communication in a public cloud, while discarding the usage of artifacts which are theoretically made unavailable by recently proposed countermeasures. Covert and side channel attacks differ in the attack scenario, however they share the underlying mechanisms for leaking information. Therefore, the study accounts for all defence strategies, as long as they are relevant with the covert channel attack scenario (e.g. constant-time implementations of cryptographic algorithms are only suitable against side channel attacks).

Microarchitectural covert channels are particularly interesting when there is no alternative means of leaking information in a non-conspicuous manner, e.g. to avoid generating network traffic and associated logs [4]. They are relevant with advanced persistent threats, where the attacker employs cutting-edge techniques in order to maintain long-term intrusion and data exfiltration capabilities. Therefore, covert channels are ideal candidates for stealthy leakage on high-profile targets. This study shows that a motivated attacker can easily make his way around state-of-the-art mitigation strategies, even if these were actually implemented in the targeted environment.

Auditing strategies have been proposed against timing channels [5]–[8]. These aim at detecting abnormal behaviours at runtime, and deploying reactive measures accordingly (e.g. interrupting the suspected workload, migrating a VM, temporarily injecting noise, etc.). Because the sustainability of the auditing approach is highly correlated to the ability of avoiding false positives, multiple machine learning-based techniques have also emerged [9]–[13]. The main drawback of auditing is that it is usually tailored for specific workloads such as cryptographic computations. Thus its applicability against microarchitectural covert channels remains an open question, as they might not have an easily identifiable signature. Furthermore, auditing does not aim at closing a malicious behaviour, but at detecting it. While the decision is made to apply reactive measures, sensitive information such as cryptographic keys might have already been leaked. Auditing strategies are not capable of closing a microarchitectural covert channel in a deterministic way, and their practicality has already been questioned due to their performance cost [14]. Therefore, they are not considered viable countermeasures against such attacks.

In this study, we first analyse a chosen set of mitigation techniques, and extract the requirements for bypassing them. Secondly, we demonstrate how all existing covert channels can be closed with countermeasures suggested in academia. Thirdly, we revisit the x86 memory bus covert channel and test it against the requirements previously established. The proposed attack is then deployed across two AWS EC2 instances and tested on several microarchitectures. Finally a discussion on how to close the remaining covert channel is provided, along with a comparison against the ARMv8.2-A architecture that has recently arrived on the Infrastructure-as-a-Service (IaaS) market. Overall, the study shows that there is no known countermeasure that could mitigate the proposed covert channel, despite an extensive review of defence strategies.

The contributions of this paper are the following,

- We evaluate state-of-the-art covert channel attacks against recently proposed covert and side channels defence techniques, and discuss how all cross-VM channels proposed in academia can effectively be closed.
- We revisit the x86 bus lock vulnerability in order to bypass recently proposed countermeasures. The covert channel is tested in the AWS EC2 environment on three different x86-64 microarchitectures.
- We propose alternative mitigation strategies and discuss the challenges of deploying the covert channel on ARMv8.2-A microarchitectures.

The paper is organised as follows. Section II provides the necessary background on hardware architecture, as well as related literature on cross-VM covert channel attacks. Section III consists of an analysis of state-of-the-art defences against side and covert channels, along with a discussion on the feasibility of covert channel attacks against these defences. Section IV presents our new instance of the memory bus covert channel. Section V reports the results of the covert channel evaluation. SectionVI provides suggestions on mitigating the covert channel, along with a comparison with the ARMv8.2-A architecture. We conclude in Section VII.

II. Background & Related Work

A. Processor Overview

In this paper, the term processor refers to the entire die, which contains the shared last-level cache (LLC), the integrated memory controller, and the cores. Cores contain individual instruction (L1-I) and data (L1-D) caches, and potentially a unified L2 cache depending on the processor model. Cores also contain one or two physical CPU(s) each. Finally, the CPU is the set of execution units and other logic required for instruction execution, e.g. translation lookaside buffer (TLB), branch target buffer (BTB), branch history buffer (BHB), return-stack buffer (RSB), etc. We note that before the advent of non-uniform memory access (NUMA), the front-side bus architecture was prominent (see Figure 1). This memory bus quickly became a bottleneck as CPU clock speeds kept increasing. With the NUMA architecture, the front-side bus was replaced by an interconnect between processors, with each processor managing its own portion of DRAM through a memory controller integrated directly into the die.

B. Memory Operations

During the execution of a program, data might be loaded from or stored to caches. Upon such operation, the memory management unit (MMU) translates the virtual address into a physical address, computes an index from the address of the requested data, and computes the tag of the cache line that contains the requested data. The index is used to point to a set of cache lines, and the tag is used to point to a specific cache line within this set. Finally, an offset computed from the variable's address is used to point to a specific portion of the cache line.

If the cache line is not present at any cache level, known as a *cache miss*, a request is issued to the memory controller in order to fetch the data from DRAM. The data is then stored into the cache, known as a *cache line fill*, and is sent back to the CPU. The next access to the cached data will result in a *cache hit*. A store operation consists of modifying a cache line, and storing it back to memory (depending on the write-policy) via the store buffer. Prior to modifying data, the cache line must be loaded. If it is not present at any cache level, it is called a *write miss*, which triggers a cache line fill. Otherwise it is a *write hit*.

C. Cross-VM Covert Channel Attacks

Ristenpart et al. [15] studied the problem of VM co-location on the AWS EC2 service. They used the LLC to assert the co-residency between two communicating VMs. Similarly, Xu et al. [16] explored the vulnerability of L2 caches for covert channel attacks on an EC2 instance. Wu et al. [17] proposed exploiting the memory bus as an alternative to cache-based covert channels, thus overcoming the addressing uncertainty.

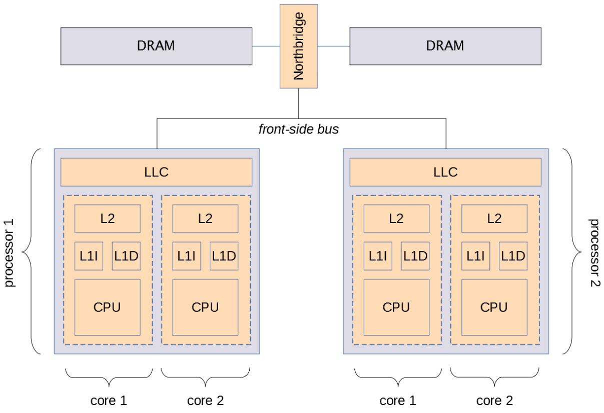


Fig. 1: Front-side bus architecture.

Later, the memory bus attack was revisited by Liu et al. [18] to use non-temporal instructions on the receiving-end, so as to mitigate the effect of cache pollution. Yet, they demonstrate that their covert channel can be closed by introducing noise on high resolution timers. Pessl et al. [19] suggested using the DRAM row-buffer as a communication medium between two VMs. Liu et al. [20] re-used a PRIME+PROBE primitive in order to build a cross-VM covert channel as a vector for side channel attacks against GnuPG libraries. Maurice et al. [21] designed a robust LLC-based covert channel attack, allegedly enabling the establishment of a rogue SSH session across AWS EC2 instances. Sullivan et al. [22] revisited the exploitation of simultaneous multithreading using the memory order buffer for cross-VM leakage in the AWS EC2 and GCE services. More recently, Semal et al. [23] proposed a cross-VM covert channel entirely based on the integrated memory controller.

III. ANALYSIS OF STATE-OF-THE-ART DEFENCES

This section surveys and analyses relevant mitigation techniques against microarchitectural leakage channels, namely noise injection, software partitioning, and hardware partitioning. Whether a timing variation is created accidentally or intentionally, the mechanisms to modulate microarchitectural states remains similar. Therefore relevant countermeasures against timing-based side channels are also considered. Other countermeasures which are not relevant include constant-time execution, symbolic execution, state flushing, and noise injection within cryptographic implementations.

A. Noise Injection

Noise injection consists of downgrading the accuracy of timing variations' measurement, either by injecting noise in the high-resolution clock sources, or by injecting randomness in the cache replacement policy:

1) Noise injection on timers: This approach consists of jittering the timestamps of high-resolution timers [18], [24]–[26]. Being able to measure the latency of a single memory operation is crucial in timing channel attacks, as it leads to the interpretation of the activity of the victim (or sender). The x86 ISA features the `rdtsc` and `rdtscp` instructions which capture a time-stamp from the time-stamp counter (TSC), allowing timing measurements with a sub-nanosecond resolution. These are accessible from any non-privileged user program. Other timing sources, such as the wall clock provided by the operating system, are usually not accurate enough to measure a timing variation of a few clock cycles. For example, in [23], the sender's activity generates an overhead of only 6.5 CPU clock cycles. At a frequency of 2.4 GHz, this amounts to a time span of 2.7 ns. Therefore, the attacker can neither rely on the high-resolution timer which can be made unreliable by countermeasures, nor on the operating systems clock sources which lack accuracy. In order to account for noise injection on high-resolution timers, we set the following condition:

Requirement 1: Noise injection on timers
The covert channel shall not rely on the `rdtsc` nor `rdtscp` instruction for measuring timing variations.

2) Noise injection on caches: This approach aims at preventing an attacker from learning about the victim's working cache set. Wang and Lee [27] suggested integrating permutations in the cache index computation, while Qureshi et al. [28] used randomised mappings based on the encryption of the cache line's physical address. These will result in the victim's accesses to stop conflicting with the attacker's cache sets. Alternatively, Fang et al. [29] suggested having the prefetch controller issuing requests to the L1 cache in order to tamper the timing observations of the receiving-end. For instance, in an m-way set associative cache, if m cache misses are observed when sending a 1, and none are observed when sending a 0. the prefetch controller will bring this number to $m/2$ all the time, such that the receiver is no longer capable of distinguishing a 1 from a 0. If generalised, these strategies can hinder cache-based covert channels that depend on the ability to find congruent addresses. Other proposals [30]–[32] studied bespoke cache replacement policies as an alternative to the vulnerable on-demand policy. Taking the example of the random-fill approach [30], if a cache miss occurs, the requested cache line is sent to the CPU but it is not necessarily stored in the cache. Instead, a "neighbour" cache line is randomly selected within a fixed address range around the requested cache line. If the same cache line is requested thereafter, it might result in a cache hit. The uncertainty contributes to inhibiting the leakage of information as to whether the victim accessed a specific cache line or not. This countermeasure is also relevant to cache covert channels such as FLUSH+RELOAD [33]. In order to account for noise injection on caches, we set the following condition:

Requirement 2: Noise injection on caches
The attacker cannot rely on the latency of cache accesses. Therefore, data caches such as the L1-D, the L2, and the LLC shall not be used as a communication medium.

B. Software Partitioning

Software cache partitioning, also known as cache colouring, consists of isolating sensitive data by means of isolating a set of cache lines for a given security domain [27], [34], [35]. Recall that in order to address data in (set-associative) caches, the MMU computes an index and an offset from the physical address. The bits that belong to both the physical page number and the cache line index are the *colour bits*. Figure 2 is an example of virtual-to-physical translation of a 64-bit address, with 6 bits of offset (i.e. cache line size is $2^6 = 64$ bytes), 9 bits of index (i.e. way size is $2^9 = 512$ entries), and 3 colour bits. Cache colouring states that physical pages which differ in any of the colour bits can never be mapped in the same cache set. That is, if the physical memory pages of two processes have at least one different colour bit, these can never exploit congruency to launch cache attacks such as Prime+Probe or Evict+Reload [3]. In a sense, cache colouring behaves like a dynamic clustering technique which guarantees that two clusters can never share a cache set. Liu et al. [36] suggested another form of software cache partitioning by leveraging Intel's Cache Allocation Technology (CAT) [37], in order to lock down portions of the LLC during execution. As for Flush+Reload, Zhou et al. [38] proposed a state machine which prevents a shared memory page being accessed by two security domains at the same time.

Beyond cache colouring, other forms of software partitioning have been proposed. Disabling page sharing [39] hinders attacks which depend on the availability of shared memory such as Flush+Reload and Flush+Flush [14]. Disabling simultaneous multi-threading (SMT) [40] prevents two hardware threads from exploiting contention among CPU-level resources such as execution units [41], the BTB [42], the RSB [43], or the MOB [22].

Requirement 3: Software partitioning
The covert channel must remain functional when shared memory and SMT are disabled. Also, set-associative caches shall not be used as a communication medium.

C. Hardware Partitioning

Hardware cache partitioning consists in providing physical isolation among the working cache sets of each tenant [27], [44], [45]. For example, Wang and Lee [27] suggested a cache line locking mechanism, by means of an ISA extension, which prevents another process from evicting the cache line. An L tag indicates whether the cache line is locked, and an ID tag indicates the process to whom the cache line belongs. Fundamentally, hardware cache partitioning results in the same effects as software cache partitioning. The main difference lies in the deployment of the countermeasure. Therefore, hardware cache partitioning does not result in additional requirements. As for other components than caches, Wang et al. [46] proposed a time-division multiplexing technique in order to prevent the exploitation of the shared integrated memory controller. Similarly, Wang et al. [47] devised a priority-based mechanism for the shared on-chip network. These approaches consist in scheduling accesses to the memory controller and the interconnect such that different security domains cannot conflict with each other. We note that the effect of this countermeasure on DRAM-based covert channels that target external NUMA nodes remains an open-question.

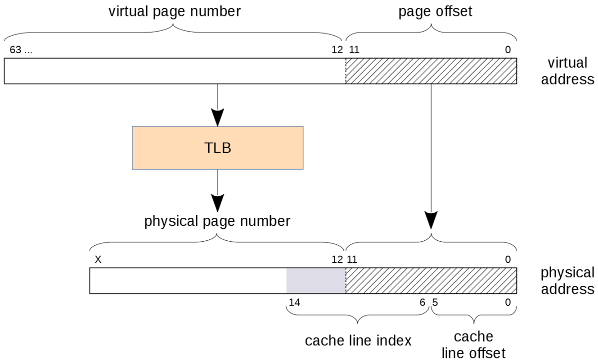


Fig. 2: Virtual-to-physical translation of a 64-bit address on an Intel E6550 processor [48]. The cache line offset is determined by bits 0 to 5, the cache line index is determined by bits 6 to 14, and the page offset is determined by bits 0 to 12. Colour bits range from bit 12 to bit 14.

Additionally, Gruss et al. [14] advocated making the `rdtsc` and `clflush` instructions privileged. While it would not completely close the covert channels which rely on these instructions, it would severely question the practicality of the attack. The adversary model (see Section IV-A) requires for the environment of the victim to be compromised with a malicious colluding software. The above-mentioned countermeasure would force this malware to be executing with privileges. Also, the adversary model assumes that the attacker does not have privileges. Therefore, using such instructions is not allowed. In order to account for hardware partitioning, we set the following condition:

Requirement 4: Hardware partitioning
The covert channel shall not rely on either the memory controller or the interconnect as a communication medium. Furthermore, the attacker cannot execute privileged code. The `rdtsc` and `clflush` instructions are considered privileged and are thus unavailable.

D. Evaluation Against Attacks

All LLC-based cross-VM covert channels [15], [16], [20], [21], [49] fail to meet requirement 2. These cannot meet requirement 3 as they exploit caches' set-associativity. Ristenpart et al. [15] and Xu et al. [16] require accessing (privileged) page tables in order to find congruent addresses, thus they also fail to meet requirement 4.

The memory order buffer (MOB) covert channel [22] depends on the availability of SMT. This attack fails to meet requirements 1 and 3. The DRAM row-buffer [19] and memory controller [23] attacks fail to meet requirements 1 and 4. Both rely on cache flushing in order to force memory accesses

TABLE I: Cross-VM covert channel attacks against covert channel countermeasures.

Attack	Exploited resource	R1	R2	R3	R4
[15]	Last-level cache	-	-	-	-
[16]	Last-level cache	-	-	-	-
[17]	Memory bus	-	●	●	-
[18]	Memory bus	-	●	●	-
[19]	Row-buffer	-	●	●	-
[20]	Last-level cache	-	-	-	-
[21]	Last-level cache	-	-	-	-
[22]	MOB	-	●	-	-
[23]	Memory controller	-	●	●	-
[49]	Last-level cache	-	-	-	-
Ours	Memory bus	●	●	●	●

being served from DRAM, and the memory controller covert channel exploits a forbidden microarchitectural component.

To the best of our knowledge, all existing covert channels rely on the `rdtsc` instruction and thus fail to meet requirement 1 and 4. Also, the memory bus covert channel proposed by Wu et al. [17] doesn't meet requirement 4 as they did not specify how they implemented uncached memory accesses—hence we assume that they proceeded with the `clflush` instruction. Liu et al. [18] claimed that their own memory bus covert channel can be closed by injecting noise in timers. We show in this paper that it is still possible to design the covert channel while bypassing their defence strategy.

More generally, the memory bus covert channel was initially designed in order to overcome the drawbacks of cache-based attacks, namely the addressing uncertainty (e.g. unprivileged virtual-to-physical address translation in virtualised environment), the scheduling uncertainty (e.g. synchronisation errors), and cache physical limitations (e.g. exploiting the L1-D doesn't allow cross-core communication). In this paper, we demonstrate that it also enables bypassing state-of-the-art countermeasures against timing channels.

IV. A Resilient Memory Bus Covert Channel

A. Adversary Model

There are two communicating entities, a *sender* and a *receiver*. The sender exists in the victim's environment in the form of a trojan or any other form of malicious program. The receiver exists in the attacker's environment. Both communicating entities execute without privileges. The instances of the victim and the attacker are scheduled on separate cores of the same processor. The hypervisor is assumed to be free of any software vulnerability, and instances are logically separated. Thus sender and receiver do not share any memory region. Finally, it is assumed that state-of-the-art countermeasures are operating in the environment of both the sender and the receiver, and that these countermeasures impose the requirements listed in Section III.

B. The Memory Bus Lock

In a multi-threaded application, shared memory regions may be accessed concurrently. In order to prevent undesirable situations such as race conditions, instructions can be performed atomically. In an atomic memory operation, the requested cache line is locked in order to prevent its modification by another thread. A singularity occurs when accessing a memory region which spans across two cache lines.

Wu et al. [17] observed that, upon accessing a cache line-crossing region (a.k.a. exotic), atomicity was enforced by locking the memory bus. By guaranteeing exclusive access of the shared bus to one thread, others would be unable to modify the cache lines of interest. When the exotic operation is completed, the memory bus is unlocked.

Moreover, Wu et al. [17] noticed that a similar behaviour happens on modern processors, where the front-side bus has been removed (i.e. NUMA processors). Atomic accesses to exotic regions result in every outstanding load/store operation to be completed across all CPUs before the atomic operation is performed [50]. This strategy effectively guarantees that no other memory operation can affect the cache lines of interest. However, it also introduces significant timing variations which are visible across all CPUs.

A covert channel can be created based on the effect of exotic memory accesses: a one is transmitted by generating atomic operations on a cache line-crossing region, a zero is transmitted by remaining idle for a fixed amount of time. Concurrently, the receiving-end probes its own accesses and interprets low and high latency accesses as zeroes and ones.

C. From Timing Variations to Binary Information

Wu et al. [17] designed a cross-VM covert channel based on the memory bus lock behaviour. However, as described in Section III-D, their covert channel can be closed with various countermeasures. Here, we demonstrate how to design the memory bus covert channel in a way that meets requirements 1 to 4. The covert channel can be broken down into two primitives:

1) Sending-end: in order to force atomicity, a `lock` prefix can be attached to an instruction. The lock signal can only be applied to read-modify-write operations whose destination operand is a memory location. Read-modify-write operations combine a load, an arithmetic, and a store operation. We choose the `xchg` instruction which simply swaps the contents of its two operands, and automatically asserts a lock signal if the first operand is a memory location. In order to transmit a one, contention is generated by passing to the assembly function (Listing 1) a pointer with a base address aligned on a cache line boundary added with an offset of 63 bytes. In order to transmit a zero, the same assembly function can be passed a pointer with a base address aligned on a cache line boundary. Also, promoting the operation to 64-bit wide with the `rex.w` prefix allows reducing the global time of execution by half.

```
; RDI = pointer to exotic or "normal" region
REX.W XCHG [RDI], RAX
RET
```

Listing 1: Transmitting a symbol.

2) Receiving-end: the x86 Streaming SIMD Extension provides instructions to perform direct read and write operations to main memory without affecting the cache. A non-temporal store of double quadword from an xmm register into a 128-bit memory address is performed with the movntdq instruction [51], [52]. The receiver can use this instruction to accelerate the probing and reduce errors due to cache pollution of other processes (see Listing 2). More importantly, it prevents the cache-miss hardware performance counter from incrementing, inhibiting countermeasures based exclusively on the monitoring of cache activity. The mfence (lines 4 and 6) instruction plays two important roles. Firstly, it prevents re-ordering between the non-temporal store (line 5) and the reading of the counter (lines 3 and 7). Secondly, it allows flushing the write-combining (WC) buffer, thus ensuring of the execution of the non-temporal store in-order. Non-temporal operations follow WC semantics, which specify that data must not be cached so as to reduce cache pollution (i.e. when data is used only once). Non-temporal operations are combined in the WC buffer, and delayed until the buffer becomes full, or upon a serialising event (e.g. mfence, cpuid, lock, etc.) [51]. We note that the size of the WC varies from one microarchitecture to another, however it can take the size of several cache lines and one single movntdq might not be enough to fill it up. Thus the second mfence instruction (line 6) ensures that the non-temporal store is not delayed until the WC buffer is full.

```
1  ; RSI = pointer to counter
2  ; RDI = pointer to "normal" region
3  MOV RDX, [RSI]
4  MFENCE
5  MOVNTDQ [RDI], XMM0
6  MFENCE
7  MOV RAX, [RSI]
8  SUB RAX, RDX
9  RET
```

Listing 2: Receiving a symbol.

In order to discard the usage of the TSC counter, we replace it with a counting thread using the inc instruction (see Listing 3). The counter value is systematically written to memory, so as to make it visible to the receiving-end. We note that this will require the receiver to have access to a second logical CPU, and that it does not alter the resolution of measurements—in fact it can even improve it [53].

```
; RDI = pointer to counter
XOR RBX, RBX
_LOOP:
INC RBX
MOV [RDI], RBX
JMP _LOOP
```

Listing 3: Counting thread.

The proposed protocol is presented in Algorithm 1. The sender performs atomic memory accesses either in a cache line-crossing region (i.e. exotic) or in a single cache line (i.e. normal), with the *Access* function referring to Listing 1. Meanwhile the receiver starts the counting thread, and probes memory accesses into a single cache line of its own userspace, with the *Probe* function referring to Listing 2. We note that sender and receiver perform multiple accesses per bit value.

Algorithm 1: Memory Bus Covert Channel Protocol

```
$M_{S1}$: exotic memory region (sender userspace);
$M_{S0}$: normal memory region (sender userspace);
$M_R$: normal memory region (receiver userspace);
$send[N], recv[N]$: respective buffers of sender and receiver;

Sender
for all $i \in [0; N]$ do
    if $send[i] == 1$ then
        {Exotic access}
        Access( $M_{S1}$ );
    else
        {Normal access}
        Access( $M_{S0}$ );
    end if
end for

Receiver
for all $i \in [0; N]$ do
    {Timed normal access}
    $t$ = Probe( $M_R$ );
    if $t > threshold$ then
        $recv[i]$ = 1;
    else
        $recv[i]$ = 0;
    end if
end for
```

The entire premise of the covert channel is based on the ability for the receiver to observe a timing variation depending on the sender's activity. In our AWS EC2 m5d.large instance pair (see Table II), the receiver's accesses to DRAM take an average of 935 increment iterations when the sender is inactive, and 2403 increment iterations when the sender is active. Therefore, there is an average performance overhead of 1468 increment iterations per memory access. An increment iteration is the time that it takes for the counting thread to perform one increment operation (see Listing 3). On the AWS EC2 m5d.large, we measured that it takes 1498034 CPU cycles to perform 2^{20} iterations, that is an average of 1.42 CPU cycles per increment operation. This means that overhead caused by the sender's activity amounts to 2084 CPU cycles. Thus it is trivial for the receiver to differentiate the binary values sent across the covert channel.

V. Evaluation

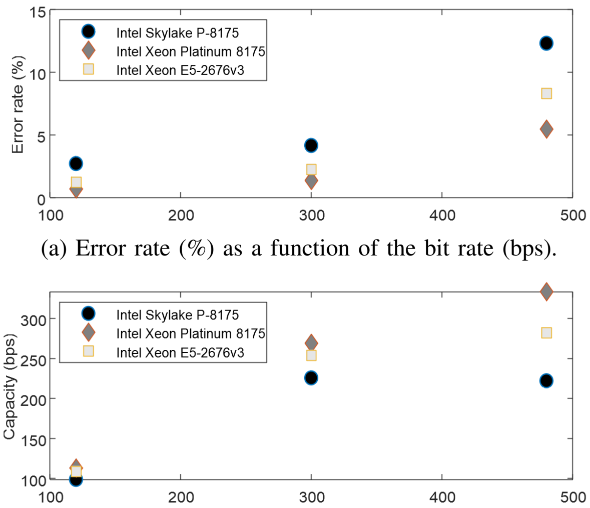
A. *Channel Capacity*

The testing environments are summarised in Table II. It consists of three AWS EC2 instance pairs featuring different x86-64 microarchitectures, namely Intel Xeon E5-2676v3 (released in 2015), Intel Xeon Platinum 8175 (released in 2017), and AMD EPYC 7571 (released in 2019). The tests are repeated on each instance pair. Both the sender and the receiver run in their own instance and have access to two virtual CPUs (vCPUs). Furthermore, dedicated instances are used in order to ensure that sender and receiver are scheduled on the same processor.

TABLE II: Hardware configuration (AWS EC2).

Instance type	Microarchitecture	vCPUs	Frequency
m4.large	Intel Xeon E5-2676v3	2	2.4 GHz
m5a.large	AMD EPYC 7571	2	2.5 GHz
m5d.large	Intel Xeon Platinum 8175	2	3.1 GHz

The error rate (Figure 3a) is computed by counting the number of bit flips over a 256-bit message. At a bitrate of 480 bps, the covert channel reaches an error rate as low as 5.46% on the Intel Xeon Platinum 8175 platform. The channel



(a) Error rate (%) as a function of the bit rate (bps).

(b) Channel capacity (bps) as a function of the bit rate (bps).

capacity (Figure 3b) is computed under the binary symmetric model [54]. It measures the quantity of information that can be reliably transmitted by accounting for the error rate[1]. At a bitrate of 480 bps, the covert channel reaches a capacity of up to 333 bps on the Intel Xeon Platinum 8175 platform. The same order of magnitude as the original proposal is achieved [17]. Results are summarised in Table III.

TABLE III: Error rate and capacity (raw bitrate of 480 bps).

Instance type	Microarchitecture	Error rate	Capacity
m4.large	Intel Xeon E5-2676v3	8.31%	281 bps
m5a.large	AMD EPYC 7571	12.3%	221 bps
m5d.large	Intel Xeon Platinum 8175	5.46%	333 bps

B. Effects on Microarchitectural States

The proposed covert channel successfully meets all the requirements previously established:

Requirement 1: Noise injection on timers. A high resolution timer is required in order to measure the latency of performing memory accesses. A timing channel can effectively be mitigated if the values read from this counter are too noisy. Requirement 1 intends to prevent such countermeasure. In order to gain assurance that neither the `rdtsc` nor the `rdtscp` instructions are used, we disassembled the entire binaries and were able to confirm that these instructions are not present. The receiving-end relies exclusively on the counting thread in order to benchmark the execution of memory accesses, and there are no other benchmarking operations taking place in the code. Only the evaluation of the channel capacity requires using the TSC counter. Beyond the testing phase, this operation is not required.

Requirement 2: Noise injection on caches. The covert channel is based on the ability for the sender and receiver to manipulate and observe microarchitectural states. In the case of cache-based covert channels, this microarchitectural state is the presence of a cache line in a cache set. Injecting noise in caches, such that the receiver and sender can loose the above-mentioned capability, can effectively mitigate the attack. Requirement 2 intends to thwart such countermeasures. The sending-end exploits the bus locking behaviour for atomic accesses to cache-line crossing regions (see Section IV-B). The resulting performance cost is generated system-wide. Thus all memory accesses are impacted, whether they target caches or DRAM. Also, the receiving-end benchmarks uncached accesses only. On x86 microarchitectures, non-temporal instruction are designed to fetch data directly to DRAM. Therefore, noise injection in caches does not affect the microarchitectural state leveraged by the sender and receiver, since they only communicate via DRAM accesses. Noise injection on caches would be completely oblivious to this covert channel.

Requirement 3: Software partitioning. Software partitioning enforces spatial isolation over certain processor resources, such that co-tenants cannot share a vulnerable microarchitectural state. Requirement 3 accounts for such countermeasures, some of which are already deployed by cloud providers. We do not have control over the disabling of SMT on the commercial platform, and dedicated instances from the same AWS account may share hardware CPUs. Therefore, we reproduce the covert channel in a lab environment, such that SMT can be disabled and processes can be pinned to separate hardware CPUs. The covert channel was successfully launched across two native processes on an AMD Ryzen Threadripper 1950X processor (Zen), which features 16 cores. SMT was disabled from the BIOS menu, and each communicating was set to different cores via the (privileged) `taskset` command. We note that this command was only used for the testing of this requirement, and that it is not necessary when launching the attack across VMs. This shows that the covert channel allows cross-core communication, hence it cannot be mitigated by disabling SMT. Finally, the cache architecture (e.g. set-associative) is not relevant with the memory bus. Firstly, because it does not rely on modulating a shared cache set. And secondly, because non-temporal accesses bypass the cache on x86 platforms. Therefore, cache colouring cannot have a mitigating effect on the proposed covert channel.

Requirement 4: Hardware partitioning. This requirement considers different forms of partitioning which would be enforced at the hardware level, from isolated cache partitions through time-multiplexing on certain scheduling resources to privileged instructions. The tests were performed from user accounts, and the disassembling of our binaries showed that neither the `rdtsc` nor the `clflush`—theoretically privileged—are used. While the memory controller is solicited in DRAM

[1] In brief, the capacity of a binary symmetric channel is a function of its binary entropy, i.e. a function of the error rate which represents the uncertainty over the quality of the information received. Maximum entropy is reached when the probabilities of a bit being erroneous and correct are equal.

accesses, it is not responsible for generating timing variations. Thus time-multiplexing over the memory controller or the interconnect cannot conceal timing variations caused by the bus lock behaviour. As for cache partitions, these are irrelevant with the proposed attack since timing variations do not cause any cache accesses.

VI. Discussion

A. Closing the Memory Bus Covert Channel

A new feature known as Memory Bandwidth Allocation (MBA) has been introduced in Intel Xeon Scalable processors [55]. This feature allows controlling the memory bandwidth of each core, and could be leveraged in order to inhibit the memory bus covert channel. The advantage of this approach is that it relies on existing hardware support, much like Liu et al. [36] used the Intel Cache Allocation Technology (Intel CAT) feature in order to close Prime+Probe cache attacks. We note that Intel CAT was also used by Lipp et al. [56] as a Rowhammer enhancer, who then suggested to modify Intel CAT in order to mitigate the vulnerability. It is possible that Intel MBA could also lead to new vulnerabilities, as many other hardware optimization techniques did, e.g. out-of-order execution, SMT, prefetching, etc. Finally, Intel MBA is not available on all Intel microarchitectures.

B. The Case of ARMv8.2-A

ARM processors have recently arrived on cloud platforms with the Neoverse microarchitectures. Thus we can expect that the share of x86 processors in IaaS will decrease for the benefit of ARM architectures. We tried reproducing the memory bus covert channel across two AWS EC2 instances platform featuring a 64-bit ARM architecture. It was possible to re-create the receiving-end, the only exception being that non-temporal instructions might not be guaranteed to be served from DRAM. As for the sending-end, the A64 `swp` instruction—equivalent of the x86 `xchg`—is deprecated since the ARMv6 ISA. To the best of our knowledge, this instruction should be re-introduced in the ARMv8.2-A ISA or upcoming versions, but it will no longer generate the desired system-wide "bus lock". We suspect that its behaviour will be similar to the load-acquire store-release concept. Meanwhile, it has not been possible to reproduce the timing channel on the Graviton and Graviton2 processors.

VII. Conclusion & Further Work

In this paper, we analysed the existing set of covert channel countermeasures, and identified how an adversary could bypass these defences. The x86 bus lock vulnerability was then revisited in order to avoid influencing microarchitectural states rendered unavailable by the latest mitigation strategies. The covert channel was deployed in a commercial cloud environment, and tested across three different microarchitectures. This study demonstrates that the proposed attack allows establishing a rogue transmission channel across two cloud instances, even if the extensive range of existing countermeasures was already deployed. Finally, an approach to mitigating the remaining vulnerability is proposed.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*. IEEE, 2019, pp. 1–19.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *CT-RSA*. Springer, 2006, pp. 1–20.

[4] "Monitoring your instances using CloudWatch," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-cloudwatch.html, last accessed 29 Jun 2020.

[5] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*. Springer, 2016, pp. 118–140.

[6] Z. Allaf, M. Adda, and A. Gegov, "A comparison study on Flush+Reload and Prime+Probe attacks on AES using machine learning approaches," in *UKCI*. Springer, 2017, pp. 203–213.

[7] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *IEEE/ACM MICRO*. IEEE, 2014, pp. 216–228.

[8] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "Spydetector: An approach for detecting side-channel attacks at runtime," *IJIS*, vol. 18, no. 4, pp. 393–422, 2019.

[9] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*. Springer, 2016, pp. 138–154.

[10] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 564, 2017.

[11] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, M. Yousaf, U. Farooq, V. Lapotre, and G. Gogniat, "Machine learning for security: The case of side-channel attack detection at run-time," in *IEEE ICECS*. IEEE, 2018, pp. 485–488.

[12] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "Cacheshield: Detecting cache attacks through self-observation," in *ACM CODASPY*, 2018, pp. 224–235.

[13] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "WHISPER: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83 871–83 900, 2020.

[14] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *DIMVA*. Springer, 2016, pp. 279–299.

[15] 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 *ACM CCS*. ACM, 2009, pp. 199–212.

[16] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *ACM CCSW*. ACM, 2011, pp. 29–40.

[17] 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*, vol. 23, no. 2, pp. 603–615, 2014.

[18] W. Liu, D. Gao, and M. K. Reiter, "On-demand time blurring to support side-channel defense," in *ESORICS*. Springer, 2017, pp. 210–228.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security*, 2016, pp. 565–581.

[20] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P*. IEEE, 2015, pp. 605–622.

[21] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: SSH over robust cache covert channels in the cloud." in *NDSS*, vol. 17, 2017, pp. 8–11.

[22] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in IaaS clouds." in *NDSS*, 2018.

[23] B. Semal, K. Markantonakis, R. N. Akram, and J. Kalbantner, "Leaky controller: Cross-VM memory controller covert channel on multi-core systems," in *IFIP SEC*, vol. 580. Springer Nature, 2020, p. 3.

[24] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[25] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *ACM CCSW*, 2011, pp. 41–46.

[26] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *ISCA*. IEEE, 2012, pp. 118–129.

[27] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007, pp. 494–505.

[28] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *IEEE/ACM MICRO*. IEEE, 2018, pp. 775–787.

[29] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prodact: Prefetch-obfuscator to defend against cache timing channels," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 571–594, 2019.

[30] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM MICRO*. IEEE, 2014, pp. 203–215.

[31] A. Fuchs and R. B. Lee, "Disruptive prefetching: impact on side-channel attacks and cache designs," in *ACM SYSTOR*, 2015, pp. 1–12.

[32] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks," in *ACM/IEEE ISCA*. IEEE, 2017, pp. 347–360.

[33] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[34] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *DAC*, 2016, pp. 1–6.

[35] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *USENIX Security*, 2012, pp. 189–204.

[36] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE HPCA*. IEEE, 2016, pp. 406–418.

[37] "Improving real-time performance by utilizing cache allocation technology," Intel Corporation (2015).

[38] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *ACM CCS*, 2016, pp. 871–882.

[39] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci, "Security best practices for developing windows azure applications," *Microsoft Corp*, vol. 42, 2010.

[40] V. K. Base, "Security considerations and disallowing inter-virtual machine transparent page sharing," *VMware Knowledge Base*, vol. 2080735, 2014.

[41] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE S&P*. IEEE, 2015, pp. 623–639.

[42] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *CT-RSA*. Springer, 2007, pp. 225–242.

[43] Y. Bulygin, "CPU side-channels vs. virtualization malware: the good, the bad or the ugly," *ToorCon: Seattle, Seattle, WA, US*, 2008.

[44] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005.

[45] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM TACO*, vol. 8, no. 4, pp. 1–21, 2012.

[46] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *IEEE HPCA*. IEEE, 2014, pp. 225–236.

[47] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *IEEE/ACM NOCS*. IEEE, 2012, pp. 142–151.

[48] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *ACM CCS*, 2014, pp. 570–581.

[49] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: cross-cores cache covert channel," in *DIMVA*. Springer, 2015, pp. 46–64.

[50] "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide," Intel Corporation (2019).

[51] "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture," Intel Corporation (2016).

[52] M. Guri, A. Kachlon, O. Hasson, G. Kedma, Y. Mirsky, and Y. Elovici, "GSMem: Data exfiltration from air-gapped computers over GSM frequencies," in *USENIX Security*, 2015, pp. 849–864.

[53] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *DIMVA*. Springer, 2017, pp. 3–24.

[54] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 1999.

[55] "Introduction to memory bandwidth allocation," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html, last accessed 29 Jul 2020.

[56] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing rowhammer faults through network requests," *arXiv preprint arXiv:1805.04956*, 2018.