Leaky Controller: Cross-VM Memory Controller Covert Channel on Multi-Core Systems

Benjamin Semal(✉), Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner

Royal Holloway University of London, Egham, United Kingdom
`benjamin.semal.2018@live.rhul.ac.uk`

Abstract. Data confidentiality is put at risk on cloud platforms where multiple tenants share the underlying hardware. As multiple workloads are executed concurrently, conflicts in memory resource occur, resulting in observable timing variations during execution. Malicious tenants can intentionally manipulate the hardware platform to devise a covert channel, enabling them to steal the data of co-residing tenants. This paper presents two new microarchitectural covert channel attacks using the memory controller. The first attack allows a privileged adversary (i.e. process) to leak information in a native environment. The second attack is an extension to cross-VM scenarios for unprivileged adversaries. This work is the first instance of leakage channel based on the memory controller. As opposed to previous denial-of-service attacks, we manage to modulate the load on the channel scheduler with accuracy. Both attacks are implemented on cross-core configurations. Furthermore, the cross-VM covert channel is successfully tested across three different Intel microarchitectures. Finally, a comparison against state-of-the-art covert channel attacks is provided, along with a discussion on potential mitigation techniques.

Keywords: Covert channel · Memory controller · DRAM · Microarchitectural attack · Cross-VM.

1 Introduction

The cloud computing model allows on-demand access to what seems an unlimited pool of storage and computing resource. In order to cope with the elastic demands of its customers, cloud providers rely on multi-tenancy. Infrastructure-as-a-service provides the service users with a virtual environment which maps dynamically to physical resource. These virtual machines (VMs) are co-located on a shared hardware platform, and separated virtually by the hypervisor. Thus, data confidentiality and integrity is enforced at the software level. Yet, because the underlying hardware is common to multiple VMs, attackers are left with means to exploit functional and timing vulnerabilities at the hardware level.

The functional behaviour of a system is usually well understood by designers. For example, the seL4 micro-kernel has been proposed as a general purpose solution, providing strong assurance of confidentiality, availability, and integrity enforcement from a functional perspective [11]. The identification of hidden leakage channels works by analyzing the system's resources or source-code. Yet, these identification methods rarely account for the system's temporal behaviour. Murray et al. [18] highlighted that seL4 micro-kernel formal proofs completely omit timing channels. *Microarchitectural timing attacks* aim at recovering data that is dependent on the timing behaviour of an application. More specifically, two processes can exploit timing variations to encode and leak sensitive data across isolation boundaries. *Covert channel attacks* employ this mechanism to violate information flow policies such as in cloud computing.

Cloud providers commonly disable support for simultaneous multi-threading [14] as well as memory deduplication [4], thus hindering a large range of microarchitectural timing attacks. Multiple academic proposals address the mitigation of timing channels in the cache memory [5,8,10,12,20,21,25,29]. Other internal (memory controller, on-chip memory bus) and external (DRAM) resource remain shared among cores, and processors. The sharing of these components has been exploited to design denial-of-service [17,33], covert and side-channel attacks [22,27]. Wang et al. [26] previously proposed a simulated version of memory controller-based leakage channels. In this paper, we present for the first time a practical implementation in both native and virtualized environments. Both attacks work in cross-core configuration, i.e. sender and receiver execute on different physical cores. We test our cross-VM covert channel attack on three different Intel microarchitectures, namely Ivy Bridge, Broadwell, and Skylake. The channel capacity is systematically evaluated and results are discussed against state-of-the-art covert channel attacks.

Contributions. This paper makes the following contributions:

1. We present two microarchitectural covert channel attacks using the memory controller channel scheduler. The first one is *privileged* and is tested in a native environment. The second one is *unprivileged* and can work under both native and virtualized configurations.
2. We evaluate the proposed covert channels under the binary symmetric model. Results of our experiments are reported in Table 3. A discussion of our memory controller-based covert channel against state-of-the-art covert is provided. We also discuss potential mitigation strategies.

Outline. The remainder of the paper is organized as follows. Section 2 provides background on the memory controller and the DRAM organization. In Section 3, we present a new memory controller native, privileged covert channel attack. In Section 4, we describe a memory controller cross-VM, unprivileged covert channel. In Section 5, we evaluate the capacity of both covert channels under the binary symmetric model. Section 6 reviews related works. Section 7 examines potential mitigation strategies. Finally, we conclude in Section 8.

2 Background

The (integrated) memory controller handles memory accesses to DRAM. Such access occurs when the data requested by a CPU is not contained in the cache(s). Before serving a memory access, the memory controller must translate the requested data's physical address into a DRAM map. A map is a selection of channel, rank, bank, row, and column. The physical-to-DRAM address translation is performed according to DRAM addressing functions. Once the DRAM map is recovered, the request is then buffered according to the bank and channel that it targets.

The memory controller contains storage and scheduling resources to arbitrate memory accesses (see Figure 1). First, a request is stored in the buffer matching the DRAM bank that it targets. Then, the bank scheduler will prioritize one request or another, according to pre-determined scheduling algorithm. Once a request wins bank arbitration, it is rescheduled by a channel scheduler. Again, the scheduling algorithm determines priorities. Usually, requests that target open-pages are served first, so as to mitigate the latency incurred by updating a row-buffer.

The memory controller's page policy dictates the aliveness of data in the row-buffer. If a close-page policy is enforced, the row-buffer will systematically be cleared after serving a request. Thus, each memory access results in a row-miss, preventing timing variations, but globally slowing down the execution of programs. If an open-page policy is enforced, the row-buffer will retain data until it must be updated with a new row. Thus, it allows the occurrence of row-hits, reducing the global execution time of programs, but introducing exploitable timing variations.

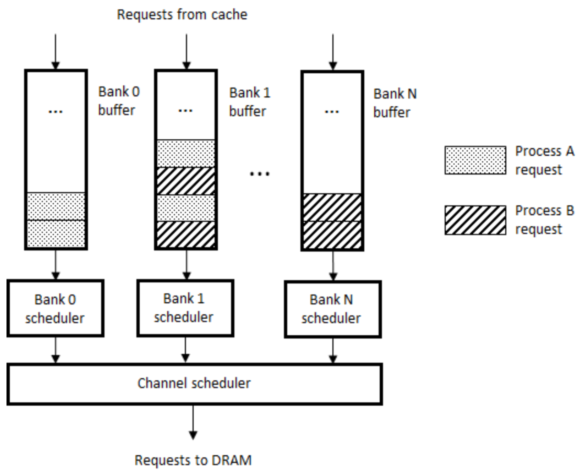

Fig. 1. Organization of a memory controller.

Several sources of contention exist in the memory controller. First, delays can be caused via the bank scheduler, as requests from different processes are mixed in the same bank buffer. If process A is the only one requesting data in a bank, its memory accesses will be served immediately. However, if another process B starts requesting data in the same bank as process A, requests of A and B will compete for scheduling. Because the load on the bank scheduler increases, requests of process A can be delayed.

Second, delays can be caused via the channel scheduler, since it arbitrates requests for several banks. If there are no other requests then for bank i, these will systematically win arbitration and be served immediately. However, if other requests for bank j, with $j \neq i$, compete for access to the channel, the load on the channel scheduler will increase, resulting in requests for bank i to be delayed.

Finally, contention can be caused via the DRAM row-buffer, as long as the memory controller has an open-page policy. Within the same bank, if there are no other requests then for row m, these will systematically result in row-hits. However, if other requests in row n interfere, with $m \neq n$, the row-buffer will alternatively be updated with rows m and n, resulting in frequent row-misses.

3 A Privileged Native Covert Channel

This section presents the basic concept to generating contention via the channel scheduler through a privileged covert channel in a native environment[1].

3.1 Threat Model

The threat model assumes two processes, a *receiver* and a *sender*, who want to share information illegitimately. The security policy forbids these two entities from communicating directly. The sender possesses sensitive information and intends to transmit this information to the receiver. Entities each have their own address space, which is dissociated in physical memory. They are running on different cores. Both entities require root privileges, and have knowledge of the DRAM addressing functions.

3.2 Principle

The proposed covert channel exploits timing variations upon uncached memory accesses. The receiver and sender both occupy space in N, the set of DRAM banks served by a single channel. The receiver "listens" to the channel by continuously performing uncached memory accesses at a pre-determined address, i.e. bank i with $i \in N$. The sender writes on the channel by creating conflicts on the resource involved in the memory accesses of the receiver. The sender generates bit values as follows,

[1] The source code of our native covert channel is available at https://github.com/bsepage/mc2c.git

- A zero is written by performing uncached memory accesses in bank j, with $j \neq i$ and $j \in N$. Because the channel scheduler only serves banks i and j, contention is negligible. Thus, the receiver's memory access in bank i will result in a "normal" latency, which is interpreted as a zero.
- A one is written by performing uncached memory accesses in all the banks comprised in N, except for bank i. This operation causes the channel scheduler to serve requests for every bank within N, which generates an observable contention. Thus, the receiver's memory access in bank i will increase in latency, which is interpreted as a one.

Table 1 summarizes how bit values are encoded and decoded across the covert channel. A **Read** (a) operation consists in performing an uncached memory access in bank a. A **Probe** (a) operation is equivalent, at the exception that the elapsed time of the operation is returned to its caller. In order to write a zero, the sender needs to perform memory accesses in a different bank than the receiver. Doing so prevents interference from the DRAM row-buffer. Indeed, if both entities were to read from the same bank, they would most likely read from different rows (a bank contains thousands of rows). As a result, reading alternatively from the sender's row and the receiver's row would cause the row-buffer to be systematically updated. Thus, memory accesses would result in a majority of row-misses, and dramatically increase in latency. Because our attack exploits exclusively the memory controller, such interference is undesirable. Furthermore, it is preferable to keep the sender active upon sending a zero, in order to compensate the effect of other microarchitectural components (e.g. memory bus). Our objective is to demonstrate the vulnerability in the memory controller, therefore we need to isolate its effect from other sources of contention.

Table 1. Modulating the load on the memory controller channel scheduler. Banks i, j, and k belong to N, the set of banks served by a single channel.

Receiver	Sender	Bit value
Probe (i)	**Read** $(j \mid j \neq i)$	0
Probe (i)	**Read** $(k = 0, ..., k = N - 1 \mid k \neq i)$	1

3.3 Design Considerations

The native, privileged covert channel works in two phases. First, each entity must identify a virtual address which maps to the desired DRAM bank(s). Then, both processes synchronize to exchange information covertly.

In the first phase, processes read the restricted */proc/self/pagemap* page translation table to compute the pointer's physical address. Physical-to-DRAM address translation (channel, rank, bank, row, column) requires knowledge of the DRAM addressing functions. These vary from one processor to another, and must be reverse-engineered if not disclosed by the manufacturer[2]. The DRAM address mapping was computed with the reverse engineering tool first presented in [22]. Prior to launching the covert channel, entities can decide on which DRAM banks to use specifically.

In the second phase, entities use the operating system wall-clock to synchronize. The `clflush` instruction is used to flush the cache upon each memory access, so as to force the request to be served from DRAM. Because an uncached memory access is higher in latency than a cached one, the `cpuid` instruction is used to prevent out-of-order execution of the time-stamp reads. Finally, timestamps are read with the `rdtsc` and `rdtscp` instructions.

4 An Unprivileged Cross-VM Covert Channel

In this section, we present a cross-VM covert channel using the memory controller. We present a strategy to discard root privileges, as well as the necessity to learn the DRAM addressing functions.

4.1 Threat Model

Consider two application processes, a *trojan* and a *spy*, running in concurrent VMs. The hardware platform features a multi-core processor, such that the hypervisor schedules each VM on a different core. The security policy enforced ensures memory isolation, access control, and does not present any software vulnerability. The *trojan* transmits a bitstream across the covert channel, and the *spy* captures the data by probing memory accesses in its own address space (see Figure 2).

Such a scenario is plausible if the adversary can infect a software with a malicious code. The (accidental) compromising of open-source software has been demonstrated, for instance, with the infection of the OpenSSL cryptographic library in 2014 (Heartbleed) [6]. Compromising of corporate software has also occurred, for example, with the multiple WhatsApp bugs [1,3]. Furthermore, whether it is open-source or proprietary, the software supply chain involves a growing number of developers and corporations. It is hard (if not impossible) for users to control that the different parties involved in the development process apply suitable security practices (e.g. a code is reviewed by a different person than its developer). Therefore, it is reasonable to assume that commercial and open-source software are not immune to malicious insiders.

With regards to the co-location problem, previous studies have shown that it is possible to create a topology of the data centre's network [24,32], even when network isolation countermeasures are employed (e.g. virtual private clouds). As a result, an attacker is capable of co-locating itself with the victim instance on a shared hardware platform.

[2] DRAM addressing functions on the Ivy Bridge test platform (see Table 2): BA0= $b_{13} \oplus b_{17}$; BA1= $b_{14} \oplus b_{18}$; BA2= $b_{16} \oplus b_{20}$; and Rank= $b_{15} \oplus b_{19}$

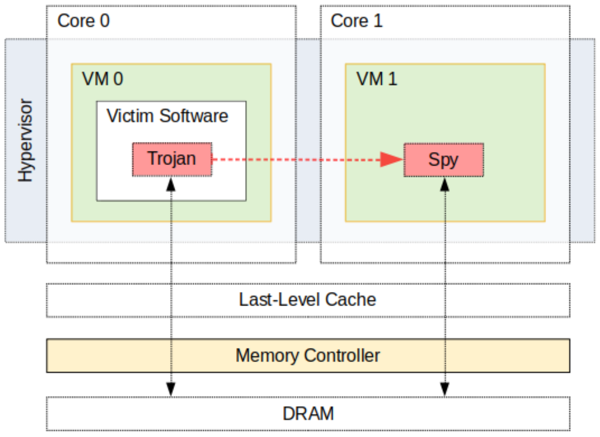

Fig. 2. Cross-VM covert channel.

4.2 Loosing Privileges & Principle

In Section 3, the channel scheduler covert channel is limited to a privileged adversary model. Indeed, an unprivileged attacker is unable to read the */proc/-self/pagemap* file, which is necessary for virtual-to-physical address resolution. Yet, the attacker needs to find addresses in its virtual address space which map to different DRAM banks.

Rather than searching for specific banks in a process' address space, we mapped several virtual pages, and observed how these were spread across physical memory. We iterated through the pages, and translated each virtual pointer into a physical address. These addresses were then converted into bank addresses, according to the platform's DRAM addressing functions. The following observations were made,

1. A single (page-aligned) virtual page is mapped to a single bank.
2. Different virtual pages tend to be mapped to different DRAM banks.

These observations suggest that the sending-end only requires to declare several virtual pages, and that each page will map to a different bank. However, there is a probability that one page will be mapped to the same bank as the one accessed by the receiving-end. Such scenario would cause row-buffer conflicts to occur. Accessing different rows triggers row-buffer updates, which would add a significant delay into the receiver's accesses.

By using a smaller amount of memory pages, the probability of having pages mapped in the same bank is reduced. The proposed methodology is detailed in Algorithm 1. The sending-end is designed such that it uses only 3 pages. Upon writing a one, the sender performs accesses in 3 different banks. Thus, the channel scheduler serves accesses in 4 different banks at once (3 for the sender and 1 for the receiver). Upon writing a zero, the sender performs accesses in 1 single page. Thus, the channel scheduler serves accesses in 2 different banks at once. If a row-buffer interference was to occur, the noise would tamper with the bitstream and the transmission would be discarded. Each bit value is repeated several times in order to improve the visibility of the contention. The value of *rep* determines the bit rate.

Algorithm 1: Transmitting bit values

```
input: message to transmit $msg$, number of repetitions $rep$
init  : map and lock 3 memory pages $P1$, $P2$, $P3$
for $i \leftarrow 0$ to $msg\_len$ do
    $bit \leftarrow msg[i]$;
    if $bit$ then
        for $j \leftarrow 0$ to $rep$ do
            $access(P1, P2, P3)$;
        end
    else
        for $j \leftarrow 0$ to $rep$ do
            $access(P1, P1, P1)$;
        end
    end
end
```

Figure 3 shows the latency of the receiver's memory accesses, with the sender alternatively being active and inactive. The latency graph shows that when the sender is active, the receiver presents an overhead of 6.5 CPU cycles on its accesses. The timing variation indicates that the proposed strategy is valid for creating a covert channel. This new approach has the benefit that it completely discards the virtual-to-bank address translation procedure. Therefore, the attacker neither requires privileges, nor knowledge of the platform's DRAM addressing functions. In this configuration, the attack can be applied to virtual environments, where physical addresses are virtualized by the hypervisor.

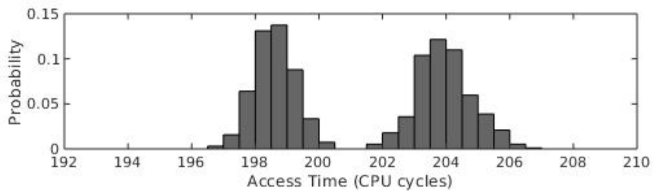

Fig. 3. Effect of active sender upon latency of receiver's memory accesses (Ivy Bridge setup).

4.3 Design Considerations

The cross-VM, unprivileged covert channel also works in two phases. In the first phase, the trojan maps and locks memory pages without reverse-engineering their physical location. We note that spy and trojan no longer require agreeing on specific DRAM banks. In the second phase, entities read or probe their memory accesses to encode and decode bit values. Probing and accessing is performed using the `clflush`, `cpuid`, `rdtsc`, and `rdtscp` instructions.

5 Characterizing the Channel Capacity

This section details our testing environment and provides an evaluation of both covert channel attacks.

5.1 Experimental Setup

The experimental setups used for characterizing the channel capacity and noise ratio are presented in Table 2. The Kernel Virtual Machine [2] hypervisor is used to manage virtual machines, and each VM is operated by a Debian distribution (Linux kernel version 4.19.0). All setups feature a dual-core processor, allowing us to lock the *trojan* and *spy* VMs onto different cores. The `virsh edit` command is used to assign a specific `cpuset` to the `vcpu` attribute. All our setups feature 16 DRAM banks. Note that a commercial infrastructure- or platform-as-a-service server system will likely feature greater amounts of DRAM, i.e. the occurrence of row-buffer interference will drop accordingly. Therefore, the proposed setup represents a worst-case scenario for the attacker.

Table 2. Experimental setups.

Setup	Processor	CPU Frequency	Memory	#DRAM banks
Ivy Bridge	Intel i5-3210M	2.5 GHz	1×4GB DDR3	16
Broadwell	Intel i7-5500U	2.4 GHz	1×8GB DDR3	16
Skylake	Intel i5-6300U	2.4 GHz	1×8GB DDR4	16

5.2 Evaluation

In order to evaluate the effective capacity, we model our covert channel as a binary symmetric channel. Under the binary symmetric model, and given a bit error probability p, the probability of correctly transmitting a bit is $1-p$. Therefore, if $p = 0.5$, it is assumed that the channel has reached maximum entropy, i.e. the probabilities of a bit being erroneous (p) and correct ($1 - p$) are equal. The binary entropy $H_2(p)$ is defined as follows,

$$H_2(p) = -p \log_2 p - (1-p) \log_2(1-p) \tag{1}$$

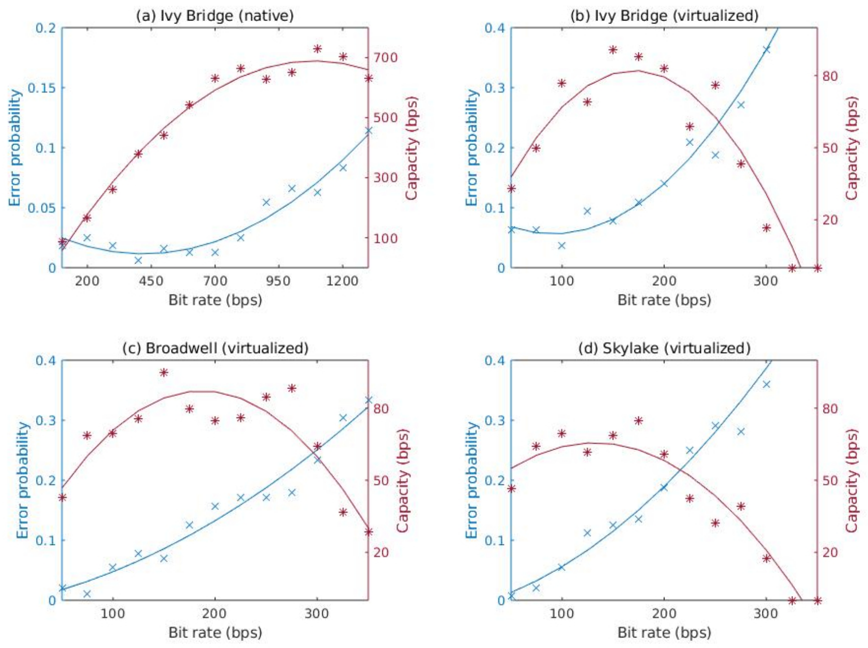

Fig. 4. Effective capacity and error probability measured against raw bit rate.

The channel capacity $C(p, r)$, defined as the quantity of information that can be transmitted reliably, is a function of the entropy $H_2(p)$ and the raw bit rate r,

$$C(p, r) = r(1 - H_2(p)) \tag{2}$$

Figure 4 compares the effective capacity and error probability against a raw bit rate ranging from 100 to 1300 bps for the native scenario (Figure 4.a), and from 50 to 350 bps for the virtualized scenarios (Figures 4.b, 4.c, and 4.d). Measures were taken by sending a fixed-size message and counting the number of bit flips on the receiving-end. The error probability p was then calculated as the number of bit flips divided by the number of bits sent. The capacity $C(p, r)$ was computed with equations (1) and (2).

In the native scenario (Figure 4.a), the error probability stays below 0.1 for bit rates of up to 1250 bps. The channel capacity reaches up to 729 bps, with an error probability of 6.25%. In the virtualized scenarios, the Ivy Bridge (Figure 4.b), Broadwell (Figure 4.c), and Skylake (Figure 4.d) setups respectively achieve a maximum capacity of 90, 95, and 69 bps. The error probability remains below 0.1 for a raw bit rate of up to 175 bps across the three setups. Results are reported in Table 3.

Virtualization has a significant impact on the effective channel capacity, as it brings additional sources of noise. First, sender and receiver compete with each other to be scheduled by the hypervisor. Second, the sender and receiver are not able to use the operating system wall-clock to synchronise, as they run

Table 3. Experimental results.

Setup	Environment	Bit rate	Error rate	Capacity
Ivy Bridge	Native	1100 bps	6.25%	729 bps
Ivy Bridge	Virtualized	150 bps	7.8%	90 bps
Broadwell	Virtualized	150 bps	7%	95 bps
Skylake	Virtualized	100 bps	5.6%	69 bps

in separate VMs. The receiver might sample at a different rate than the sender can transmit, with the bias increasing over time. Third, programs executing concurrently (e.g. hypervisor) can alter the state of the channel scheduler, bank scheduler, or row-buffer. We note that our implementation is free from any fault recovery technique.

6 Discussion & Related Work

At the highest level, simultaneous multi-threading (SMT) allows concurrent threads to share execution units and CPU caches. Percival [21] demonstrated a covert channel between two threads, based on contention within the L1-D and L2 caches. Shortly after, Wang and Lee [28] designed a covert channel that leverages contention on multipliers. More recently, Sullivan et al. [23] demonstrated a high-speed covert channel between two hyperthreads in Amazon EC2 and Google Compute Engine instances. In a virtualized environment, core-level co-residency is hard to achieve as VMs tend to be isolated onto different cores. Furthermore, this class of covert channels is only relevant to cloud platforms where hyperthreads are enabled.

Other works proposed exploiting the LLC cache as it is fast and shared among cores. Xu et al. [31] proposed exploiting conflict in the LLC. They used a covert channel to achieve co-location in an Amazon EC2 setting. Further works followed based upon the *Prime+Probe* [13,15,19] or *Flush+Flush* technique [9]. Maurice et al. [16] implemented a robust covert channel capable of establishing a rogue SSH connection across Amazon EC2 instances. A number of academic works have been proposed in order to tackle timing vulnerabilities emanating from the cache, including hardware cache partitioning [12,20,21,29], software cache partitioning [5,8,10], and noise injection [25,29]. It is difficult to assess whether these covert channels could bypass such countermeasures, and calls for further analysis.

Wu et al. [30] exploited the memory bus as a high-bandwidth covert channel medium. Their Amazon EC2 experiment achieves an effective capacity of 340 bps, with the use of a robust communication protocol. In order to prevent timing channels over the on-chip memory bus (or network), Wang and Suh [27] proposed two approaches. The first one consists in prioritizing traffic from high-security domains over lower ones, thus isolating sensitive software from a potential spy.

The second one consists in assigning separate hardware resources to different security domains, thus inhibiting timing channels on the memory bus.

Pessl et al. [22] built a high-speed covert channel based on on the DRAM row-buffer. Their channel reaches up to 596 kbps in virtualized environment. Mitigating row-buffer covert channels could be achieved by enforcing a close-page policy on the memory controller. As a result, every memory access would result in a row-miss, thus inhibiting the timing channel. Furthermore, authors relied on a privileged adversary model, and both entities need to undergo an initialization phase in order to agree on a specific DRAM bank. This agreement cannot be performed online without incurring additional memory usage side-effects. We propose an alternative to the row-buffer exploitation, with a weaker adversary model and less operational constraints.

The vulnerability of the memory controller was previously demonstrated by Moscibroda et al. [17]. Their work shows that by combining all timing channels detailed in Section 2, a malicious process can slowdown the execution of a concurrent process by a factor of 190%. It is worth noting that their denial-of-service attack exploits both the memory controller, and the DRAM row-buffer. Furthermore, they do not address the problem of encoding and decoding information across virtual machines via the channel scheduler.

7 Mitigation

Memory controller-based (and DRAM row-buffer covert channels) rely on uncached memory accesses. Therefore, one countermeasure consists in disabling or restricting access to the `clflush` instruction. This mitigation technique would require architectural changes, thus adding in complexity. However, it would go a long way to making shared platforms resilient against this class of microarchitectural covert channels.

Auditing-based techniques have been proposed in the past [7,9]. The systematic flushing of the cache causes a very high number of cache misses, which can be monitored in order to detect abnormal behaviours. However, auditing usually results in high numbers of false positives. Further work is required to assess whether this is a suitable approach.

Wang et al. [26] proposed an alternative hardware design of a memory controller. They achieve temporal isolation between different security domains, at the cost of a memory latency ranging from 60% to 150%. So far, there haven't been any countermeasures relying on spatial isolation, or noise injection.

8 Conclusion & Further Work

In this paper, we presented two instances of microarchitectural covert channel attacks using the memory controller channel scheduler. The first attack is privileged and was tested in a native environment. It achieved a capacity of up to 729 bps (raw bit rate of 1100 bps). The second attack is unprivileged and was tested in a virtualized environment. It achieved a capacity of up to 95 bps (raw bit rate of 150 bps). In further work, we aim to develop countermeasures to prevent exploitation of the memory controller and the DRAM row-buffer resource. We also intend to expand the study to multi-processor x86-64 server platforms, as well as investigating mechanism for bi-directional communication.

References

1. Hackers used whatsapp 0-day flaw to secretly install spyware on phones. `https://thehackernews.com/2019/05/hack-whatsapp-vulnerability.html`, last accessed 18 Feb 2020
2. Kernel virtual machine. `https://www.linux-kvm.org`, last accessed 18 Feb 2020
3. New whatsapp bug could have let hackers secretly install spyware on your devices. `https://thehackernews.com/2019/11/whatsapp-hacking-vulnerability.html`, last accessed 18 Feb 2020
4. Base, V.K.: Security considerations and disallowing inter-virtual machine transparent page sharing. VMware Knowledge Base **2080735** (2014)
5. Cock, D., Ge, Q., Murray, T., Heiser, G.: The last mile: An empirical study of timing channels on sel4. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 570–581. ACM (2014)
6. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al.: The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement. pp. 475–488. ACM (2014)
7. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**(1), 1–27 (2018)
8. Godfrey, M.M., Zulkernine, M.: Preventing cache-based side-channel attacks in a cloud environment. IEEE transactions on cloud computing **2**(4), 395–408 (2014)
9. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+ flush: a fast and stealthy cache attack. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 279–299. Springer (2016)
10. Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In: The 21st USENIX Security Symposium. pp. 189–204 (2012)
11. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an os microkernel. ACM Transactions on Computer Systems (TOCS) **32**(1), 2 (2014)
12. Liu, F., Ge, Q., Yarom, Y., Mckeen, F., Rozas, C., Heiser, G., Lee, R.B.: Catalyst: Defeating last-level cache side channel attacks in cloud computing. In: 2016 IEEE international symposium on high performance computer architecture (HPCA). pp. 406–418. IEEE (2016)
13. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy. pp. 605–622. IEEE (2015)
14. Marshall, A., Howard, M., Bugher, G., Harden, B., Kaufman, C., Rues, M., Bertocci, V.: Security best practices for developing windows azure applications. Microsoft Corp p. 42 (2010)
15. Maurice, C., Neumann, C., Heen, O., Francillon, A.: C5: cross-cores cache covert channel. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 46–64. Springer (2015)

16. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the other side: Ssh over robust cache covert channels in the cloud. In: NDSS. vol. 17, pp. 8–11 (2017)
17. Moscibroda, O., Mutlu, T.: Memory performance attacks: Denial of memory service in multi-core systems. In: 16th USENIX Security Symposium (2007)
18. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: from general purpose to a proof of information flow enforcement. In: 2013 IEEE Symposium on Security and Privacy. pp. 415–429. IEEE (2013)
19. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: Practical cache attacks in javascript and their implications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1406–1418. ACM (2015)
20. Page, D.: Partitioned cache architecture as a side-channel defence mechanism (2005)
21. Percival, C.: Cache missing for fun and profit (2005)
22. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In: 25th USENIX Security Symposium. pp. 565–581 (2016)
23. Sullivan, D., Arias, O., Meade, T., Jin, Y.: Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In: NDSS (2018)
24. Varadarajan, V., Zhang, Y., Ristenpart, T., Swift, M.: A placement vulnerability study in multi-tenant public clouds. In: 24th USENIX Security Symposium. pp. 913–928 (2015)
25. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating fine grained timers in xen. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 41–46. ACM (2011)
26. Wang, Y., Ferraiuolo, A., Suh, G.E.: Timing channel protection for a shared memory controller. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). pp. 225–236. IEEE (2014)
27. Wang, Y., Suh, G.E.: Efficient timing channel protection for on-chip networks. In: 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip. pp. 142–151. IEEE (2012)
28. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: 22nd Annual Computer Security Applications Conference (ACSAC'06). pp. 473–482. IEEE (2006)
29. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. ACM SIGARCH Computer Architecture News **35**(2), 494–505 (2007)
30. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. IEEE/ACM Transactions on Networking **23**(2), 603–615 (2014)
31. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of l2 cache covert channels in virtualized environments. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 29–40. ACM (2011)
32. Xu, Z., Wang, H., Wu, Z.: A measurement study on co-residence threat inside the cloud. In: 24th USENIX Security Symposium. pp. 929–944 (2015)
33. Zhang, T., Zhang, Y., Lee, R.B.: Memory dos attacks in multi-tenant clouds: Severity and mitigation. arXiv preprint arXiv:1603.03404 (2016)