Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds

Dean Sullivan
University of Florida
deanms@ufl.edu

Orlando Arias
University of Central Florida
oarias@knights.ucf.edu

Travis Meade
University of Central Florida
travm12@knights.ucf.edu

Yier Jin
University of Florida
yier.jin@ece.ufl.edu

***Abstract*—We introduce a new microarchitectural timing covert channel using the processor memory order buffer (MOB). Specifically, we show how an adversary can infer the state of a spy process on the Intel 64 and IA-32 architectures when predicting dependent loads through the store buffer, called *4K-aliasing*. The 4K-aliasing event is a side-effect of memory disambiguation misprediction while handling write-after-read data hazards wherein the lower 12-bits of a load address will *falsely* match with store addresses resident in the MOB.**

In this work, we extensively analyze 4K-aliasing and demonstrate a new timing channel measureable across processes when executed as hyperthreads. We then use 4K-aliasing to build a robust covert communication channel on both the Amazon EC2 and Google Compute Engine capable of communicating at speeds of 1.28 Mbps and 1.49 Mbps, respectively. In addition, we show that 4K-aliasing can also be used to reliably detect multi-tenancy.

I. Introduction

Infrastructure as a Service (IaaS) clouds, such as Amazon EC2 and Google Compute Engine (GCE), are growing in popularity as increasingly powerful computing resources become more affordable. IaaS affordability is largely a result of many users sharing the same cloud infrastructure in a process known as *multi-tenancy*. Multi-tenancy allows disparate users with varying needs to deploy applications on-demand and at scale, while at the same time significantly improving utilization of data center resources [32].

Research has demonstrated, however, that malicious users can abuse multi-tenancy to leak information across virtual machine (VM) instances via covert- and side-channels. Attacks demonstrating extraction of sensitive information via side-channels across colocated VMs are prevalent [34], [43], [27], [26], [25], [41]. Other equally damaging attacks in the cloud include covert channels. These attacks occur when two cooperating, but isolated, parties communicate with one another. Often covert channels are used to extract information, but can also be effective in determining multi-tenancy [14], [39], [28], [38], [36].

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23221
www.ndss-symposium.org

Historically, IaaS timing channel attacks focus on cross-core leakage. In contrast, there has been less research on IaaS *same-core* timing attacks that exploit hardware hyperthreads, also known as symmetric multithreads (SMT), or shared resources. Under controlled experimental environments and desktop platforms, innovative work has demonstrated SMT covert- and side-channels on branch predictors [24], [16], [2], [15], CPU functional units [37], [4], [6], the translation-lookaside buffer [23], [20], and level one instruction [1], [3] and data caches [42], [33], [30], [35]. Despite the large number of such attacks, many have yet to be demonstrated on the public cloud. The lack of research on SMT-based IaaS timing channels follows from two assumptions: 1) SMT covert- and side-channels are usually considered trivial to prevent by simply disabling hyperthreading; and 2) It is relatively easier to colocate two VMs on the same package, as opposed to the same core.

The first assumption is *prima facie* true, but not without side-effects in that it impacts both end-user total cost of ownership (TCO) and performance. For example, benchmarking floating point intensive applications with SMT enabled typically degrades performance because of competition for limited floating point unit resources [8], [19]. In these cases, disabling hyperthreading may be a good option to improve performance. However, even when SMT is disabled in the VM the hypervisor is free to schedule it as a hyperthread on the processor.

Dedicated instances are offered as solutions ensuring that a user's application is physically isolated at the hardware level from VMs belonging to other accounts [5]. In turn, this yields a higher TCO due to the cloud provider's increased operational expenses incurred for allocating the dedicated hardware. In the past Microsoft Azure completely disabled hyperthreading by default, but has recently moved towards offering hyperthreaded VMs for general purpose workloads [22]. Similarly, the majority of Amazon EC2 and GCE instances come with SMT enabled. This implies, along with Azure's shift to offering SMT VMs, that decreasing the cost of operational expenses outweighs the need to manage specific workloads causing performance degradation. Hyperthreading is expected to become more popular on IaaS platforms in the near future in order to keep them affordable.

We address the second assumption by demonstrating a new SMT-based covert channel on both the Amazon EC2 and GCE IaaS platform. We show that it is capable of detecting multi-tenancy with equivalent success rate to other cross core multi-tenant detection schemes at comparable cost. Using two

TABLE I: Comparison of side and covert channels. Approaches with N/L in a field do not list that particular metric.

Method	Shared Resource	Covert Channel	Side Channel	Col. Detect.	Bitrate	Error rate	VM	SMT
Wang & Lee [37]	SMT/FU	●			500 kbps	N/L	●†	●
	SMT/Cache		●		3.2 Mbps	N/L	●†	●
	spec. load		●		200 kbps	N/L	●†	●
Aciiçmez et al [4]	BTB		●		N/L	21%		●
Hunger et al [24]	BPU	●			100 kbps	45.2%	●	
Xu et al [39]	LL Cache	●			3.2 bps	9.3%	●	
Ristenpart et al [34]	L2 Cache	●			0.2 bps	N/L	●	
Wu et al [38]	LL Cache	●			346 bps‡	0.39%‡	●	
Maurice et al [28]	LL Cache	●			751 bps	5.7%	●	
This work	Store buffer	●		●	1.28* & 1.49** Mbps	$< 8.7\%$	●	●

† Authors present a discussion that isolation through a hypervisor is not sufficient to prevent the described covert channel and side channels, but provide no metrics on their bandwidth or error rate in a cloud environment.
‡ Best case scenario metrics in a cloud environment.
* Amazon EC2.
** Google Computer Engine (GCE).

cooperating accounts, we demonstrate multi-tenant detection after launching 14 instance pairs on EC2 and 12 instance pairs on GCE using the placement strategies outlined in [36]. We further demonstrate that our SMT covert channel results in a 15*x* increase in channel capacity compared to other demonstrated IaaS covert channels once multi-tenancy has been established.

4K-Aliasing. Our new covert communication channel leverages Intel's memory ordering buffer (MOB). The MOB is an intermediate pipeline buffer that resides between the execution units and L1 data cache. It manages in-flight reads and writes that have not yet been written back to memory, henceforth referred to as committed. Intel's memory ordering model [9] guarantees program consistency on all processor families that execute instructions out of program order. However, out-of-order (OoO) execution causes several common data hazards such as when a later[1] write passes an earlier load [write-after-read (WAR)] or a later read passes an earlier write [read-after-write (RAW)]. As memory reads and writes are speculatively executed the MOB is checked prior to when an instruction is retired. When a data hazard is detected the instructions can be re-issued safely as the contents of the MOB have not yet been committed, otherwise they can be written to memory as normal.

We use a side-effect of managing the write-after-read data hazard by the MOB called *4K-aliasing*. WAR hazards occur when the address of a speculatively executed younger write is found to alias with an older read. The hazard is detected during memory disambiguation prediction by comparing the lower 12-bits of every load and store in the MOB. Upon a match, the load is re-issued with an associated performance penalty. However, a read address separated from a write by 4 KB will *falsely* match. We demonstrate that the falsely matching 4K-aliasing event incurs a deterministic performance penalty measureable across processes.

Contributions. To the best of our knowledge we are the first to investigate 4K-aliasing as a covert channel. We therefore extensively evaluate the associated timing and noise characteristics under ideal, single process conditions before moving on to isolated processes across cores, and then VM instances on IaaS public clouds. We demonstrate a robust covert channel on both Amazon EC2 and GCE clouds with a low bit error rate and channel capacity of 1.28 Mbps and 1.49 Mbps respectively. We further present a case-study on multi-tenancy detection on EC2 after launching 14 cooperative instance pairs, and on GCE after launching 12 cooperative instance pairs.

The remainder of the paper is organized as follows: In Section II we overview related works and provide comparisons with our 4K-aliasing timing channel. In Section III we establish the background and basis for 4K-aliasing. In Section IV we characterize the 4K-aliasing channel in a single process scenario. In Section V we present both a simple and robust 4K-aliasing covert communication channel. Section VI provides a case study when the 4K-aliasing covert channel is deployed on both Amazon EC2 and GCE. Section VII describes our multi-tenancy detection experiment and results. In Section VIII we analyze possible mitigations. Finally, Section IX concludes and provides directions for future work.

II. Related Works

A. Shared Resources Timing Channels

There have been several side-channels leveraging contention between functional unit resources while hyperthreading. Wang and Lee [37] demonstrated a covert channel due to exception handling during control speculation on loads using the IA-64 ISA. Aciiçmez and Seifert [4] show that hardware threads contending for a shared multiplier can form the basis for a side-channel capable of distinguishing multiplications from squarings in OpenSSL's implementation of RSA. Andrysco et al. [6] implement a timing attack capable of rendering victim web pages through the Firefox browser caused by floating-point unit slowdown when operating on subnormal values.

Timing channels caused by contention for the branch prediction or branch target buffer (BTB) also leverage hyperthreading. Aciiçmez et al. [2] demonstrated that RSA encryption keys can be partially recovered by monitoring execution latency after evicting branch target addresses in the BTB. Recently Evtyushkin et al. [15] demonstrated that BTB collisions could be exploited to leak kernel space addresses to break kernel address space layout randomization. Hunger et al. [24] demonstrate a covert channel based on the branch predictor as opposed to the BTB.

[1] We use the terms *later* and *earlier* to refer to program order.

Hund et al. [23] and Gruss et al. [20] both use contention in the translation lookaside buffer (TLB) to defeat kernel address space layout randomization. While not reliant upon hyperthreading, their work leverages the core-private TLB which is a shared resource in that it saves address translation data across isolated security domains, namely kernel-space and user-space addresses. Similarly, Aciiçmez [1] and Aciiçmez and Schindler [3] both demonstrate a `PRIME + PROBE` style attack using the instruction cache by filling it with dummy instructions, and then timing their re-execution after preemption from an RSA process. Osvik et al. [30] use memory access patterns in the L1 data cache to fully extract an AES key from a victim process. The work is extended in [35] by Tromer et al. Yarom et al. [42] demonstrate complete private key recovery against a constant time RSA implementation using L1 data cache bank collisions.

B. Covert Communication Channels

Xu et al. [39] demonstrate a covert channel using shared last level caches between cooperating VMs on Amazon EC2 with a bit rate of 3.2 bps and error rate of 9.3% on average. This improved upon a prior public cloud covert channel by Ristenpart et al. [34], which reported a bit rate of 0.2 bps using the shared L2 data cache. Wu et al. improve upon both schemes using atomic operations on last level caches to induce memory bus transactions in [38]. In house experiments by the authors observe a channel transmission rate of around 750 bps with an error rate of 0.09%. When deploying to an EC2 instance under best conditions the channel presents a transmission rate of 343.5 bps with an error rate of 0.39%. Under heavy noise conditions the channel presents a similar transmission rate but the error increases to 21.56%. The authors then change the protocol for better error handling effectively lowering the data transmission rate to about 110 bps but an improved error rate of 0.75%. Maurice et al. [28] again target the shared last level cache, but address the uncertainty of which cache lines the cooperating parties should transmit upon using inclusiveness. In doing so, they achieve a bit rate of 751 bps in a virtualized environment.

C. Multi-Tenancy Detection and Placement

The first work to demonstrate multi-tenant detection mapped externally available VM instance IP addresses to their internal IP addresses, allowing them to topologically map the data center and then colocate malicious VMs next to target VMs [34]. Herzberg et al. [21] extended the work of Ristenpart et al. by demonstrating new techniques for deanonymizing internal IP addresses, revealing the network topology, and testing for colocation. Xu et al. [40] again use network topology as a means for multi-tenancy detection, but do so systematically. Varadarajan et al. [36] demonstrated a new technique for multi-tenancy detection by creating memory bus contention using locked atomic operations. Their work also revealed common VM placement strategies employed by IaaS providers.

D. Comparison of Approaches

We show a high level comparison of previous approaches in Table I. As demonstrated, our covert channel offers a higher data transmission rate than the enumerated approaches while still keeping a low error rate. We should give special mention to Wang and Lee's cache side channel, which offers a comparable transmission rate of 3.2 Mbps. The authors discuss how their side channel can also be used as a covert channel. Unfortunately, they do not discuss the error rate in their approach, nor the performance of their approach in a cloud environment.

Furthermore our approach is noteworthy in that we present, to the best of our knowledge, the first multi-tenant detection scheme using hardware hyperthreading. Our demonstration in Section VII shows a clear detection threshold, while only requiring a relatively small number of VMs to be launched from cooperating accounts.

III. Background

A. Memory Order Buffer

The memory ordering buffer (MOB) has been a key microarchitectural component since the Intel Nehalem microarchitecture [11]. The MOB enables loads and stores to be issued speculatively and execute out-of-order, ensures that retired loads and stores occur in order with correct values, and enforces Intel's memory ordering model. Memory disambiguation and store-to-load forwarding are two features of the MOB that allow loads and stores to be speculatively issued and executed out-of-order, respectively.

Stores to memory are temporarily written into a *store buffer* prior to commitment enabling the processor to continue execution without having to stall, for instance because the store is waiting on a busy L1 data cache write port. Delaying writes also makes more efficient use of bus bandwidth via streaming. The store buffer comprises the virtual and physical store address and the store data of executed stores [12]. As long as a store has not been retired, it occupies a store buffer entry slot. Once the store address and data are known, the store data can be forwarded to any following load operations in a process called store-to-load forwarding. This is an important performance feature of modern processors because it saves cycles by allowing the load to obtain its data directly from the store without having to access the cache subsystem.

Both the store address and data must be known before store data can be forwarded to a dependent load. Accordingly these loads must wait, but younger loads that are independent of the store should be allowed to issue and execute ahead of the stalled load. Otherwise significant performance slowdown and underutilization of resources will occur in the case of a stalling, long store/load dependency chain. Intel's solution to optimizing available instruction level parallelism (ILP) is to allow younger loads to be speculatively issued and later disambiguated in a process called memory disambiguation. When a load speculatively issues, it takes its data from the L1 data cache, even when older store addresses are unknown, and updates its load buffer slot. Prior to commitment the prediction is verified. If a conflict exists, the load and all succeeding instructions in the loads dependency chain are re-executed.

B. Memory Disambiguation Prediction

Memory disambiguation prediction for loads occurs early in the Issue stage of the processor pipeline to optimize ILP

Processor Model	Microarchitecture	Clock Freq	Memory Order Buffer	OS/Kernel Version
Intel Xeon E5-2690	Sandy Bridge	2.9 GHz	SB: 36 / LB: 64	CentOS 2.6.32
Intel Core i7-3770	Ivy Bridge	3.4 GHz	SB: 36 / LB: 64	Arch Linux 4.10.8
Intel Core i7-4770	Haswell	3.5 GHz	SB: 42 / LB: 72	Arch Linux 4.8.13
Intel Core i7-6820HQ	Skylake	2.7 GHz	SB: 56 / LB: 72	Arch Linux 4.10.1

TABLE II: Experimental platform specifications. SB means store buffer and LB means load buffer.

by allowing loads to speculatively execute ahead of stores that have not yet been resolved. The exact details of the prediction algorithm are undocumented. However, the Intel Core® microarchitecture employed a hashed index lookup using the load's instruction pointer [13]. Each entry in the memory disambiguation predictor behaved as a saturating counter that was updated at instruction retirement. Prediction was verified by comparing the address of all dispatched store operations against the address of all younger loads. It is unclear, however, if the memory disambiguation prediction algorithm is still in place on newer Intel microarchitectures.

C. Coherency Snooping

Memory ordering is correctly maintained if a memory read (load) results in the same value that was written by the most recent memory write (store). Ordering must be maintained between earlier loads and later writes as well as earlier and later loads. Chowdhury and Carmean [7] outline a method for maintaining ordering between memory operations in a multiprocessor by snooping the load buffer before a store operation completes using per-core *memory snoop logic*. In response to committing a store to the L1 data cache, the snoop logic compares the store address with every load address in the load buffer. If a match is found, then an ordering violation is triggered for the corresponding load instruction. This results in the load, and preceeding speculatively executed instructions in the load dependency chain, to be aborted and re-issued. The snoop of the load buffer in the embodiment outlined in [7] reveals that it is implemented at cache-line granularity, e.g., the lower 12-bits of the virtual and physical address.

D. 4K-Aliasing

Intel's documents assume dependency between 4 KB separated loads and stores and calls this *4K-aliasing* [11]. 4K-aliasing occurs when the lower 12-bits of the address of a load issued after a preceding store *falsely matches* in the store buffer. However, the cause of 4K-aliasing is undocumented. Recall that memory disambiguation prediction will attempt to issue loads early to speculatively execute ahead of independent stores. Perhaps the memory disambiguator prevents all loads whose lower 12-bits match a store address in the store buffer from being issued early. This is a relatively cheap decision considering the stall cycles caused by the incorrect prediction on independence.

Alternatively, 4K-aliasing could be the result of coherency snooping. In this scenario, the front-end might allow all loads to speculatively execute. Prior to commitment of any store, the snoop logic will find a false match in the load buffer on the lower 12-bits of a load address in the load buffer. This will cause the load to abort, and any instructions in the load's dependency chain to be re-issued. Perhaps Intel reasons that loads to unique page frames are rare, or that virtual address aliasing is uncommon.

In both case, however, the responsible microarchitectural logic does not distinguish between threads or process. This is stated explicitly in the patent describing coherency snooping [7]. We surmise that distinguishing between processes is likely not employed during memory disambiguation prediction based on other prediction logic at the front-end, e.g., the branch target buffer and branch predictor.

IV. 4K-Aliasing within a Single Process

In this section, we aim to evaluate the timing characteristics due to 4K-aliasing within a single process. This achieves several goals. First, it verifies the performance penalty due to the 4K-aliasing event. Second, it allows us to determine the conditions under which 4K-aliasing is observable. Finally, we aim to establish a baseline of expected behavior within a controlled environment prior to demonstrating the 4K-aliasing covert communication channel.

A. Initial Benchmark and Experimental Setup

We initially want to verify the performance penalty caused by 4K-aliasing without forcing the event to occur. The Intel optimization guide suggests that this event will likely be observable during a memory copy routine where the source and destination buffer addresses are separated by 4 KB. Each time the data to be copied is read, a 4 KB aligned base address will *falsely match* with the 4 KB aligned base address of the source buffer. A deterministic performance degradation should be observable as memory disambiguation prediction will incorrectly predict a dependency between the copy.

Figure 1 shows the performance penalty of 4K-aliasing in the memory copy routine when the source and destination buffers are separated by 4 KB. We evaluate the 4K-aliasing effect on four recent Intel microarchiture families: Sandy Bridge, Ivy Bridge, Haswell, and Skylake. Table II shows the configuration for the platforms tested. All systems use the Intel 64-bit ISA and 64-bit GNU libc 2.24 libraries. The results show that the copy bandwidth drops every time an address is aligned on 4 KB boundary. Subsequent copies to addresses that do not align on a 4 KB boundary rapidly recover. Note that the performance falls off after 16 KB as two array references cannot be serviced within the same 32 KB L1 data cache.

The granularity of observable 4K-aliasing events for the memory copy routine is too coarse to offer much insight into the cycle latency of a falsely aliasing load. Ideally, we need to be able to clearly distinguish the number of cycles required to service a 4K-aliasing load versus the number of cycles required to service all other memory load events. This will allow us

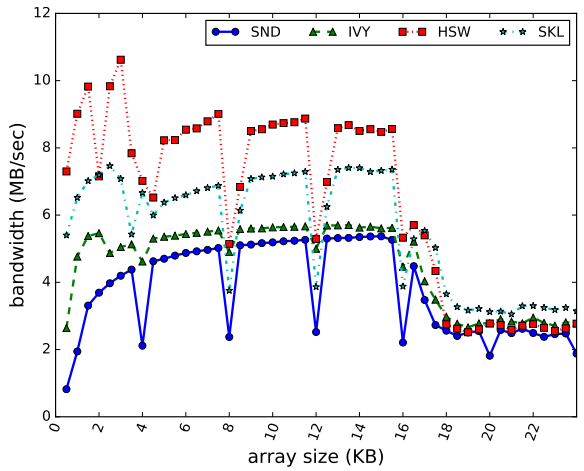


Fig. 1: Effect of 4K-aliasing in memory order buffer. SND: Sandy Bridge, IVY: Ivy Bridge, HSW: Haswell, SKY: Skylake.

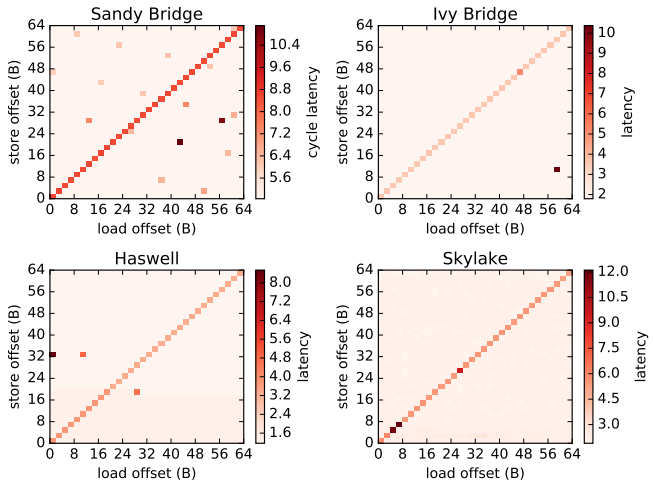


Fig. 2: Load latency due to 4K-aliasing in memory order buffer when the load address is parametrically swept by 2 bytes with respect to a constant store address.

to reliably convert the 4K-aliasing event into a stable covert communication channel.

In order to understand the observable 4K-aliasing cycle latency in an ideal setting we need to clarify the conditions required for this scenario. We are interested in measuring the cycle latency of a load issued after a preceding store that falsely aliases in the store buffer. This can occur in two situations. A later (in program order) load executes after an earlier (in program order) store, or when a later write passes an earlier store. The latter is referred to as a write-after-read (WAR) data hazard. WAR hazards are undesirable because executing a later store ahead of a load when their memory addresses match will cause the read to load incorrect data. In fact, Intel's VTune performance analysis guideline [10] describes 4K-aliasing as a side-effect of the memory order buffer preventing WAR hazards.

Therefore, for our analysis we opted for measuring WAR events. Listing 1 presents the baseline benchmarking code used to measure the overhead due to 4K-aliasing within a single process. The benchmark parametrically sweeps load addresses by 2 bytes with respect to a constant store address, measuring the associated latency. It is written such that the lower 12-bits of one load address within the load array in the inner loop will falsely alias with the lower 12-bits of a preceding store address in the store buffer.

The actual measurement code is contained within the inline assembly section of Listing 1. We use `AT&T` syntax to write the assembly, which is read as `inst. dest, src`. The first two assembly instructions (lines 7 & 8) move the array pointers for `store_arr` and `load_arr` to local integer register `r14` and `r15` respectively. An immediate value is stored at the array index pointed to by `store_arr` at line 11. Line 12 derefences `load_arr` and adds the value to the value in `r9`. Note that we add a variable number of single cycle add immediate instructions [17] to ensure that we do not incorrectly alias with memory operations at lines 7 & 8. In our experiments, we set the number of add instructions to equal the average cycle latency required to access the L1 data cache for the given microarchitecture family. When the load address at line 12 aliases on the lower 12-bits with the store at line 11, we expect the timestamp counter to report more cycles. We use the `rdtscp` instruction to record cycles following the Intel guideline [31].

Listing 1: Baseline 4K-aliasing latency benchmark.

```
1  for (uint64_t i = 0; i < ARR_SZ; i += 2) {
2      store_arr = &sarr[i];
3      start = rdtscp();
4      for (uint64_t j = 0; j < ARR_SZ; j += 2)↩
           {
5          load_arr = &larr[i];
6          asm volatile (
7              "movq   %0, %%r14        \n\t"
8              "movq   %1, %%r15        \n\t"
9              "add    $0x1, %%r9       \n\t"
10             ...
11             "movq   $0x2, (%%r14)    \n\t"
12             "add    (%%r15), %%r9    \n\t"
13             :
14             : "r" (store_arr), "r" (load_arr↩
                   )
15             : "%r9", "%r14", "%r15"
16             );
17     }
18     stop = rdtscp() - start;
19 }
```

B. *Single Process 4K-Aliasing Results*

Figure 2 depicts the measured load latencies as a heatmap. For a given row (y-axis) the store address is held constant, while the load is incremented by 2 bytes. Longer latency measurements are depicted as dark squares and shorter latency measurements as light squares. For each store address, the lower 12-bits of a single load address falsely aliases creating a measurable performance degradation. The latency is reported as cycles.

The Sandy Bridge and Haswell microarchitecture families show several high latency measurements off the 4K-aliasing

diagonal. We do not employ any scheduling constraints on the executed benchmarks, so these are likely due to external noise caused by kernel scheduling events. We calculated the average penalty for the results to be 4.3 cycles across all families. This is lower than reported by Intel [10], which reports a best-case expected 5 cycle penalty for the Sandy Bridge, Ivy Bridge, and Haswell microarchitectures. The measurement error for the Skylake microarchitecture, however, is more significant which has an expected 7 cycle penalty.

C. Refined Single Process Benchmark and Results

To eliminate this error, we ran another experiment that introduces additional load instructions in the measurement code. Our goal was to increase the number of 4K-aliasing events within the window prior to the store's retirement. All four microarchitecture families can service at least two load or store address calculations via their load/store address functional units [12]. The benchmark in Listing 1 executed one store and one load, which allows them to execute in parallel and more quickly recovery from disambiguation misprediction. The new load instructions added to the measurement code quickly exhaust the load/store address generation functional units resources, producing a longer latency. Figure 3 shows these results. In effect, adding 5 load instructions has increased the cycle latency in the 4K-aliasing measurement by roughly 4x. This allows 4K-aliasing latency to be clearly distinguished from normal memory load operations that do not 4K-alias with preceding stores.

D. Analysis of Multithreaded 4K-Aliasing

The memory-order buffer is local to a processor core such that any covert channel based upon 4K-aliasing should occur between hyperthreads. However, in Intel Hyper-Threading Technology [11] a single processor core splits its execution resources between two processes. This includes the available

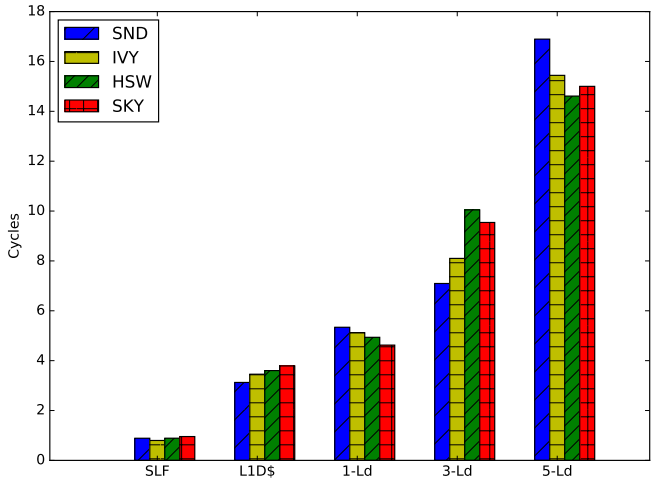


Fig. 3: Cycle latency when (from left to right) a load can get its data from i) the store buffer via store-to-load forwarding; ii) L1 data cache; iii) L1 data cache after one 4k-aliasing event; iv) L1 data cache after three 4K-aliasing events; and v) L1 data cache after five 4K-aliasing events.

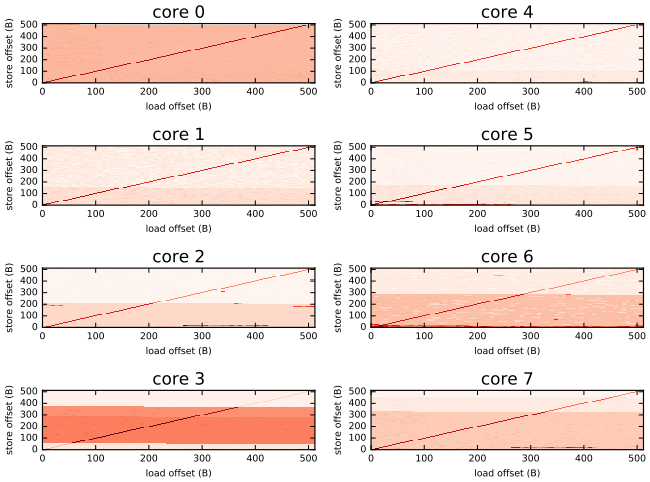


Fig. 4: Effect of 4K-aliasing when different processes are scheduled on the same core as hyperthreads.

slots in the load and store buffer respectively. Recall, however, that the snoop logic outlined in Section III-C operates on both a *cross-thread* and *external* store commits. The *memory snoop logic* responsible for triggering 4K-aliasing, therefore, should be measureable for both hyperthreads and cross-core threads, but not cross-core processes. The latter is disambiguated because they do not share common memory. Accordingly, we aim to establish the latency of 4K-aliasing due a multithreaded process and later establish if cross-core 4K-aliasing is measureable.

We evaluate two scenarios for multithreaded processes: 1) We launch multiple processes each of which executes two threads. One thread executes only 4KB aligned stores and the other thread executes only 4KB aligned loads; 2) We schedule these threads such that they hyperthread on different cores. For example, we schedule P1-T1-C0/P1-T2-C2 and P2-T1-C0/P2-T2-C2 [2] such that C0 and C2 are not hyperthreads. The latter scenario addresses a necessary condition for establishing the 4K-aliasing covert communication channel, namely *whether 4K-aliasing is observable across processes*.

Figure 4 shows the results of running this benchmark on an Intel Core® i7-4770 CPU with 4 physical cores capable of supporting 2 hyperthreads per core. For this platform core 0 and core 4 are hyperthreads, so too are core 1 and core 5, core 2 and core 6, and core 3 and core 7.

The experiment used code similar to Listing 1 except that the sweep range for load and store addresses was much wider to collect a more inclusive series of measurements. Both the wider search range and execution across available virtual cores induces system noise caused by both other processes and OS scheduling events.

The results of this experiment establish several features of the 4K-aliasing event. First, it is indeed possible for load addresses from one process to 4K-alias with preceding store

[2] Here `P` refers to a process, `T` refers to a thread launched from that process, and `C` refers to the processor core the thread is executed on. So P1-T1-C0 should be read as process 1, thread 1, executes on core 0.

addresses from another unique process. Second, system noise is clearly observable and dampens the measureable latency caused by 4K-aliasing. The effects of noise on the measurement is most apparent on core 0 and core 3, but is also apparent on cores 6 and 7.

E. Analysis of Cross-core 4K-aliasing

The coherency snoop logic outlined in Section III-C indicates that 4K-aliasing occurs across threads that share common memory to maintain memory ordering in a multiprocessor. We evaluated whether the 4K-aliasing timing channel could be measured across processes executed on separate cores. In all of our experiments, we failed to measure the 4K-aliasing timing channel when executing two processes scheduled on separate cores, one of which executed 4KB aligned stores and the other 4KB aligned loads.

V. Multi-Process 4K-Aliasing

Based on our analysis in Section IV, memory disambiguation prediction does not distinguish between processes when predicting a dependency between 4 KB separated loads and stores. Given that, we set out to show first a simple 4K-aliasing covert communication channel. We use the simple protocol to then address the challenges in designing a robust covert channel, and finally characterize its error rate and channel capacity.

A. Threat Model and Assumptions

Before outlining the 4K-aliasing covert communication, we elaborate upon the assumptions we make and threat model under which covert communication is applicable.

We assume that two cooperating applications are running on the same system. We use *trojan* and *spy* to refer to these processes as is common in the literature [14]. We also assume a scenario in which the trojan wishes to communicate a secret to the spy and no other communication channel exists between the two parties. We further assume that the trojan and spy processes are collocated on the same physical CPU as hyperthreads. The system software, including the operating system and runtime, are considered to be secure so that neither the trojan nor the spy can bypass isolation and access controls. Finally, we assume both the trojan and spy have only user-level privileges.

B. Simple 4K-Aliasing Covert Communication Channel

A malicious user can communicate with a cooperating party via a pre-determined protocol using the 4K-aliasing timing channel. For example, the *sender* can set all of the store buffer entries to 4 KB aligned addresses to communicate a 1. At some later time, the *receiver* process can schedule itself to the same core and read the store buffer state by performing reads on 4 KB boundaries while measuring the load latency. This initial protocol is shown in Listing 2.

In order to demonstrate the simple covert communication channel, we execute two processes on the same processor core as hyperthreads. We refer to these as the *trojan* and *spy* processes hereafter. The trojan aims to communicate a secret to the spy as a string of binary values. In order to do so, the trojan fills the store buffer with addresses aligned on 4 KB boundary. For the Intel Haswell microarchitecture, the trojan is responsible for storing at least 42 addresses in the store buffer. This value will change per microarchitecture but not significantly, see Table II. The spy aims to load values that will 4K-alias with the trojan's store addresses such that the measured execution time in cycles can distinguish between a 1 and 0.

The trojan repeatedly executes stores separated by 4 KB to transmit a 1, and empties the store buffer to transmit a 0. The trojan drain the store buffer by executing a memory ordering instruction such as `mfence` or executing a busy-wait loop such that the store instructions retire normally. In our initial experiments, we saw no difference between either option. However, as will be discussed in Section V-D, the speed with which the store buffer is drained is integral to modulating the pulse width during bit transmission. This in turn has consequences on the covert channel's capacity via its bit error rate.

The spy process intermittently probes the store buffer context for 4 KB aligned addresses by executing loads aligned on a 4 KB boundary. The cycle time is measured during each probe block. If the spy process measures an increased cycle time, then it reasons the trojan is transmitting a 1 bit. If the spy process measures a decreased cycle time, then the trojan is transmitting a 0 bit.

C. Results of Simple 4K-Aliasing Covert Channel

Figure 5 presents the results using the simple covert channel. The x-axis represents time (in ns) with respect to the start of the spy program. The y-axis displays cycles. The results show that: 1) 4K-aliasing can be used as a covert communication channel; 2) given a naive protocol, a separable cycle latency threshold based on 4K-aliasing events can be established, e.g. in this case the threshold can be placed at 9 cycles; 3) the simple covert channel suffers from stability issues apparent between times 100 and 200 ns; and 4) the

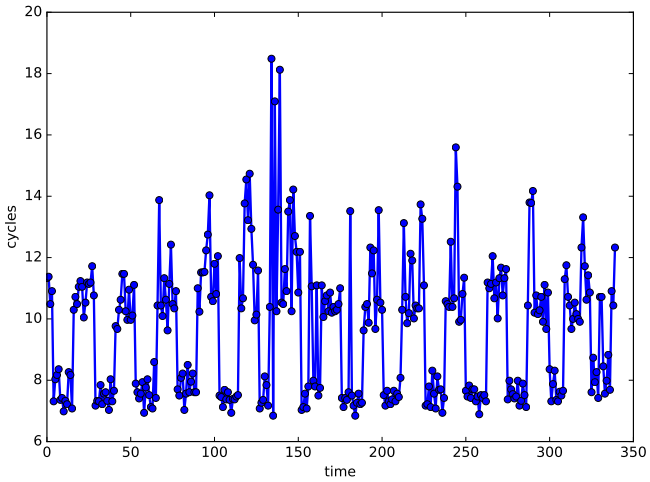


Fig. 5: Timing results of 4K-aliasing communication channel between trojan and spy.

Protocol: Simple 4K-Aliasing Covert Communication Channel

$Data_{send}$[N], $Data_{recv}[N]$: data bits to be transmitted/received by the trojan and spy
$addr_{4k}$: A 4 KB aligned address
latency: Cycle length of servicing 4 KB aligned load
threshold: Experimentally set cycle length capable of distinguishing 4K-aliasing

Trojan's operation:

```
while true do
    for i = 0 to N − 1 do
        if Data_send[i] = 1 then
            store ← addr_4k
        else
            flush_store_buffer()
        end if
    end for
end while
```

Spy's operation:

```
for i = 0 to MaxProbes − 1 do
    start = rdtscp()
    load ← addr_4k
    stop = rdtscp()
    if latency > threshold then
        Data_recv[i] = 1
    else
        Data_recv[i] = 0
    end if
end for
```

Listing 2: Simple protocol for establishing a 4K-aliasing covert communication channel between a trojan and spy process.

achievable bit rate while using this protocol is roughly 77 Mbps given that 27 bits were transmitted over 350 ns.

D. Characterizing a Robust 4K-Aliasing Communication Channel

Given the previous analysis, there are several challenges that need to be addressed in establishing a robust 4K-aliasing covert channel. The first involves improving the bit rate and bit error of the channel. Another challenge is how the trojan and spy will detect one another. This is necessary as both a precursor to the trojan knowing when to send the secret, but also the spy acknowledging its receipt. Finally, the trojan and spy must be able to recover from failed transmission. The latter can occur, for instance, when the trojan or spy is descheduled by the OS. Ideally we would like to construct a protocol capable of acknowledging and recovering from a failed transmission with a low bit error rate while maximizing the channel capacity. We address these challenges in the following.

Theoretical Bit Transmission Window. Ideally, the bit rate for the 4K-aliasing communication channel is limited by the time required to fill the store buffer with 4 KB aligned addresses plus the time to issue and measure the 4K-aliasing event. We assume that the worst-case lifetime of a single 4 KB aligned address available in the store buffer is bounded by accessing the L1 data cache, which takes 4 clock cycles on our Intel Core® i7-4770.

In practice, the store will retire more quickly [17]. Therefore, each store takes roughly 1.2 ns to retire given a 3.4 GHz clock. In the best case then, a 4K-aliasing load can be issued within 4 cycles of the store in order to allow a bit to be transmitted every 1.2 ns resulting in a theoretically maximum channel bit rate of 833 Mbps.

In practice, however, there are several limiting factors preventing the channel from achieving this bit rate. To demonstrate this, we perform an experiment that aims to measure the number of cycles it takes for a store buffer entry to be evicted, which bounds how quickly the spy can recover a trojan bit. In the trojan process, we repeatedly loop over a store to a 4 KB aligned address. In the spy process we first execute only a 4 KB aligned load, then a single cycle instruction and a 4 KB aligned load, then two single cycle instructions and 4 KB aligned load. We stop executing intermediate single cycle instructions when the observed cycle latency due to 4K-aliasing is masked by the number of single cycle instruction latencies.

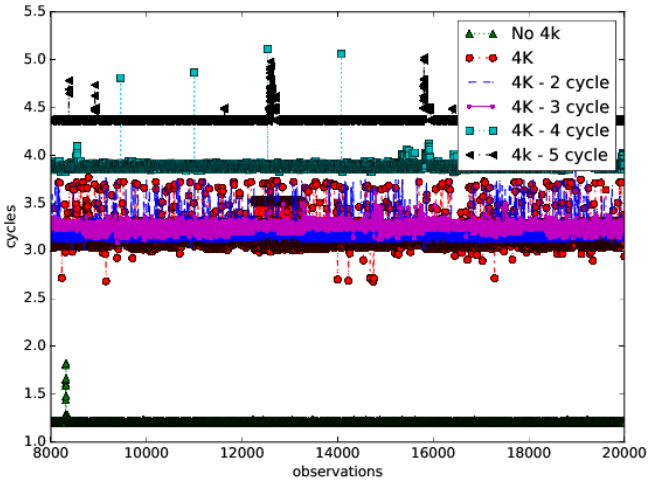
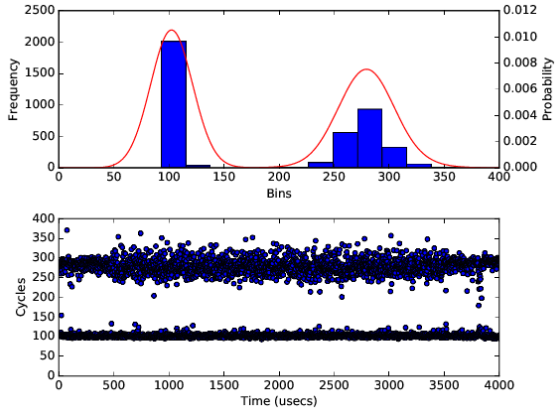


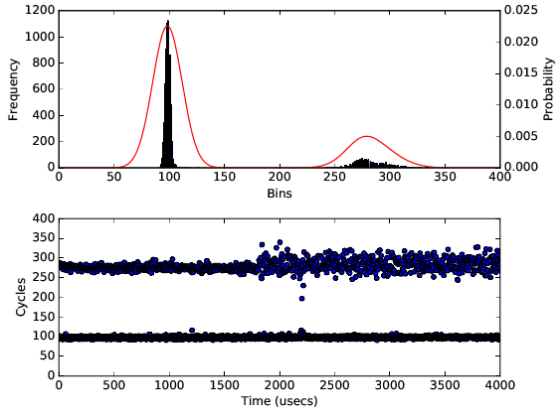
Fig. 6: Practical store bandwidth of 4K-aliasing communication channel.

The results of this experiment are shown in Figure 6. Two features stand out from the results. First, issuing successive 4 KB aligned stores and loads in a tight loop is prone to noise. Many of the measurements oscillate between 3 to 4 cycle when 4K-aliasing is the dominant feature. This indicates that some measurements show aliasing, while others show cumulative effects due to functional unit resource starvation. Second, the effect of 4K-aliasing is indistinguishable when four intermediate single cycle instructions are executed before the 4 KB aligned load. At five intermediate instructions, 4K-aliasing is no longer visible as it is dominated by the latency of the intermediate instructions.

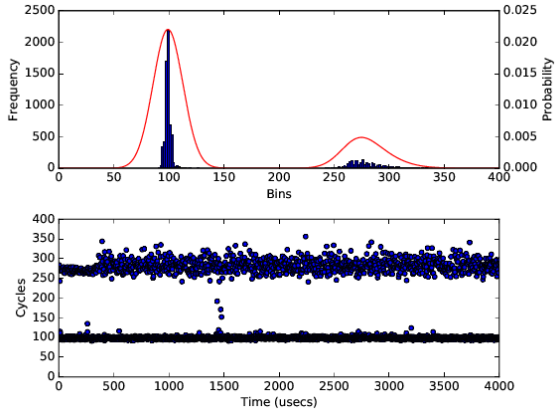
Another aspect of this experiment is that we repeatedly executed 4 KB aligned stores such that the store buffer was always maximally filled with 4 KB aligned addresses. This implies the 4K-aliasing event should then be measureable after



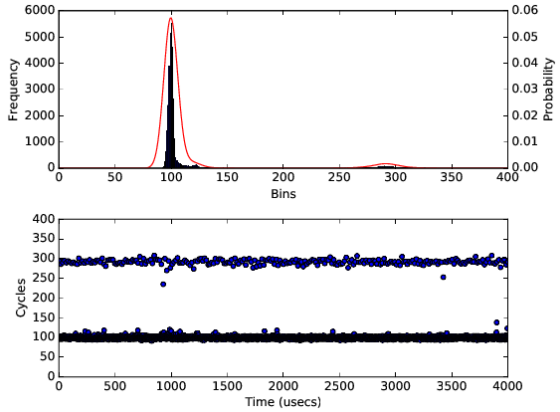
(a) Distribution of 4K-aliasing latency as the interval of probed addresses increases by 2 KB.



(b) Distribution of 4K-aliasing latency as the interval of probed addresses increases by 1024 B.



(c) Distribution of 4K-aliasing latency as the interval of probed addresses increases by 512 B.



(d) Distribution of 4K-aliasing latency as the interval of probed addresses increases by 256 B.

Fig. 7: Distribution of 4K-aliasing latency.

42×4 cycles[3], or 49.4 ns per 1.2 ns. Since we execute 4 KB aligned loads in a tight loop, on average they should execute and 4K-alias before any store in the trojan process is retired achieving a more practical bit rate of 19.8 Mbps. The results confirm this analysis in that 4K-aliasing is observable as long as a 4 KB aligned load is issued within four cycles of a 4KB aligned store.

Theoretical Bit Transmission Frequency. Another practical limitation of the 4K-aliasing covert channel is the frequency at which the measurements are taken. The frequency is a function of the offset between each successive 4 KB aligned store/load address. For example, both store and load addresses can be separated by 4096 B, 2048 B, 1024 B, 512 B. As long as they agree on the step size for successive memory operations the trojan and spy will 4K-alias. Note, that this metric is different than determining the theoretical bit rate per the analysis above because any practical covert channel will have to oversample in order to capture all of the trojan transmitted bits. The 4K-alias protocol will have to agree on some variation of address offset (step size) to meet the sampling criteria.

In order to determine the timing characteristics and distribution of 4K-aliasing events given a predetermined address offset, we ran the following experiment. For each benchmark, we measure 4K-aliasing events such that the lower 12-bits of the store and load addresses will be aligned every other load (when the load address is swept at intervals of 2048 B), every fourth load (when the load address is swept at intervals of 1024 B), every eighth load (when the load address is swept at intervals of 512 B), and every sixteenth load (when the load address is swept at intervals of 256 B). The results are shown in Figure 7.

Figure 7a shows the results when load addresses are measured at 2 KB intervals. Half of these measurement should, therefore, be slow during a 4K-aliasing event and the other half should be relatively fast. The top half of Figure 7a plots the distribution of the data and fits it using a Gaussian kernel density estimate. While not uniformly bimodal, the probability of finding a high cycle measurement is nearly equal to the probability of finding a low cycle value. However, it is limited

[3]The number of store buffer entries available on the Haswell microarchitecture family is 42.

Protocol: Robust 4K-Aliasing Covert Communication Channel

$Data_{send}$[N], $Data_{recv}$[N]: data bits to be transmitted/received by the trojan and spy
$Detect_Probes$: The number of probes used to detect the presence of the spy/trojan
$addr_{4k}$: A 4 KB aligned address
$latency$: Cycle length of servicing 4 KB aligned load
T_{one}: Experimentally set cycle length capable of distinguishing a trojan one bit from a trojan zero bit
$T_{tr,detect}$, $T_{sp,detect}$: Cycles to detect presence of trojan and presence of spy, respectively
C_{one}, C_{zero}: Cycles during which a one and zero data bit are sent, respectively

Trojan's operation:

```
for i = 0 to Detect_Probes − 1 do
    start = rdtscp()
    store_to_load_forward_loop()
    stop = rdtscp()
    if latency > T_sp,detect then
        break
    else
        continue
    end if
end for
for i = 0 to N − 1 do
    if Data_send[i] = 1 then
        for j = 0 to C_one do
            store ← addr_4k
        end for
    else
        for j = 0 to C_zero do
            flush_store_buffer()
        end for
    end if
end for
```

Spy's operation:

```
for i = 0 to Detect_Probes − 1 do
    start = rdtscp()
    trojan_probe(load ← addr_4k)
    stop = rdtscp()
    if latency > T_tr,detect then
        break
    else
        continue
    end if
end for
while Trojan is transmitting do
    start = rdtscp()
    SFENCE
    load ← addr_4k
    stop = rdtscp()
    if latency > T_one then
        Data_recv[i] = 1
    else
        Data_recv[i] = 0
    end if
end while
```

Listing 3: Robust protocol for establishing a 4K-aliasing covert communication channel between a trojan and spy process.

by a relatively noisy signal caused by toggling rapidly between 4K-aliasing/no 4K-aliasing measurements. In fact, it appears as if the cycle latency measurements take time to transition.

Figure 7d records a 4K-aliasing event every sixteenth measurement and exhibits improved stability compared to every other plotted sample. In general, these results indicate that the 4K-aliasing covert channel can be modulated depending on the frequency at which a 4 KB aligned store is executed, or equivalently, when a 4 KB aligned load is executed.

Detection of Cooperating Parties. The prior analysis outlined the characteristics of the 4K-aliasing covert communication channel, but detection of cooperating parties remains undecided. In effect, the trojan can detect the presence of the spy by executing a tight loop of instructions that take advantage of store-to-load forwarding, see Section III-A. When the spy is absent, the store-to-load forwarding loop will execute with a deterministic latency. When the spy is present, however, the store-to-load forwarding path will be interrupted by the spy competing for functional unit and MOB resources. We found that a store-to-load forwarding loop in the trojan process could detect the spy's presence within roughly 200 cycles on average.

For the spy to detect the trojan, we incorporate a 1-wire communication protocol, which wraps a data bit in a 1-bit header and footer. Each trojan data bit is prepended with 0 and appended with a 1, such that the spy receives either 001 and 011 for each bit transmitted by the trojan. This technique inherently allows the spy to distinguish the trojan's absence; while the trojan is sending a 0 bit, 4K-aliasing will be regularly observable as a 001 message whereas when the trojan is idle 4K-aliasing will not be observable at all.

Recovery from Failed Transmission. We use initialization and completion messages to recover from failed transmissions. The initialization phase is entered upon mutual detection so that the communication is synchronized from a known starting point. A successful message is indicated by receipt of an agreed upon completion message. Otherwise, either the trojan or spy failed and the message must be resent from initialization.

E. A Robust 4K-Aliasing Covert Channel

Equipped with methods for detection, synchronization, and failure recovery, we are now in a position to redefine the simple covert channel presented in Section V-B with the protocol defined in Listing 3. We incorporate several features in an attempt to correct for the noise characteristics shown in Figures 6 and 7, while at the same trying not to unnecessarily limit the capacity.

Both the trojan and the spy operate in one of two stages: detect and transmit. There are four parameters affecting the practical bitrate given the 4K-aliasing protocol in Listing 3: i) $T_{tr,detect}$, ii) $T_{sp,detect}$, iii) T_{one}, and iv) $Detect\_Probes$. The first is used to trigger transmission once the presence of the spy has been detected. Effectively, the trojan can detect the presence of the spy by executing a tight loop of instructions that take advantage of store-to-load forwarding, see Section III-A. In our in-house experiments. We found that it took less than 200 cycles on average to detect the spy's presence after it is scheduled as a hyperthread with the trojan.

Within the trojan process we iterate over the spy detection loop for an interval of $Detect\_Probes$. This can be any length with out loss of applicability. The only requirement is that the trojan and spy can schedule themselves for execution. Accordingly, we decided to set this to a multiple of the bit length of an n-bit message. We experimentally determined that the trojan can transmit 1-bit every 58731 cycles. Hence, we set

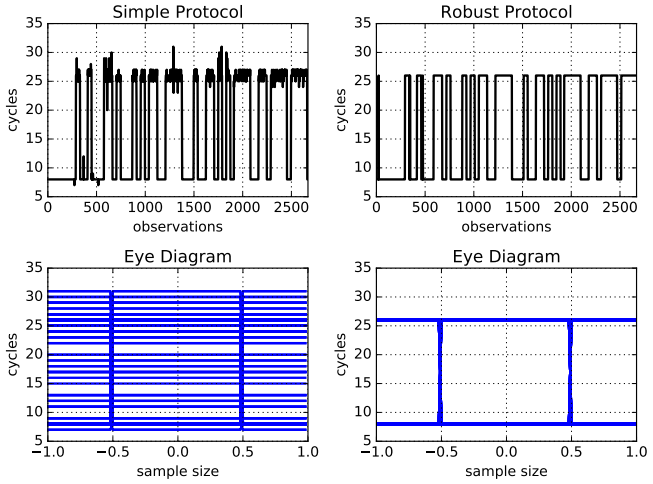


Fig. 8: Intersymbol interference improvement in 4K-aliasing covert communication channel. The plots on the left illustrate the signal and eye diagram for simple 4K-aliasing protocol. The plots on the right illustrate the signal and eye diagram after incorporating the robust communication channel for the 4K-alias covert channel.

Fig. 9: Percent error rate as a one and zero trojan bit transmission frequency.

Detect_Probes in the trojan to 10× this amount. As a side-effect of this the trojan's bandwidth is 57 kbps using a 3.4 GHz clock.

To detect the trojan, we incorporate a 1-wire communication protocol, which wraps a data bit in a 1-bit header and footer. The parameters $T_{tr,detect}$ and T_{one} are both limited by the number of bits they can decipher given a single bit transmission from the Trojan under the 1-wire protocol. However, we have much more freedom for sampling the trojan transmission because the best case theoretical 4K-aliasing bit rate is 19.8 Mbps, see Section V-D. We set the spy to sample the trojan every 580 cycles. We set $T_{tr,detect}$ to 100 × this amount to provide a reasonable amount of time to detect the trojan's presence. T_{one} is also set to 580 cycles resulting in an effective sampling rate of 5.91 Mbps. Given that for every trojan data bit sent three total bits need to be received by the spy to stably recover the transmission, our expected 4K-aliasing covert channel bit rate is roughly 2 Mbps.

F. In-House Robust 4K-Aliasing Covert Channel Results

We ran an experiment wherein we transmitted a known message using both the simple and robust protocols to compare both their resilience to noise and bit rate. The results are shown in Figure 8 and depict a side-by-side comparison of the protocols along with the messages intersymbol interference diagram. Intersymbol interference distorts a digital signal such that the previous bits in the message warp subsequent signals in the message [18]. We found through this in-house experiment that the 1-wire communication protocol eliminated interference in the trojan bit transmission during transitions from binary 1 to 0 and binary 0 to 1. The eye diagram for the robust protocol has a clean opening indicating that the sampling rate is adequate. The eye's zero crossing meet the Nyquist criterion such that it allows maximum robustness against sampling phase offsets.

The 1-wire communication protocol does not come without a cost, however, and will decrease our channel capacity. To find the experimental channel capacity we need to know the channel's error rate, which we compute by performing an experiment wherein we generate 1 million random bits and transfer them through the 4K-aliasing covert channel. We then compute the error rate of the received signal for zeros and ones separately. We perform this measurement over varying trojan signal pulse widths (e.g. trojan signal frequency). The pulse widths are determined by the number of 4 KB aligned stores executed per bit. We send 3 bits every sixteenth, eighth, fourth, and second time unit. A plot of the frequency versus error rate is shown in Figure 9.

Interestingly, executing a 4 KB aligned store every sixteenth time unit (256 B) shows a poor error rate for transmitting a one compared to executing 4 KB aligned stores every eighth time step (512 B). This contradicts our visual inspection of frequency distributions shown in Figure 7d versus Figure 7c. We reason that this is because transmitting a one bit every sixteenth time unit distorts the 1-wire communication protocol due to the pulse width of the one signal, making it more difficult to distinguish transmitting a zero (001) from a one (011). We conclude from this analysis that the ideal error rate for sending a one should occur at step sizes of 1024 B, and step sizes of 2048 B for sending a zero. These form the parameters C_{zero} and C_{one} in Listing 3.

These results also show that the covert channel error for sending zeroes and ones is asymmetric. Hence, the channel can be characterized as a binary asymmetric channel with noise. The capacity of this channel is given by the following equation [29]:

$$C = \frac{H_b(\varepsilon_1)\varepsilon_0}{1-\varepsilon_0-\varepsilon_1} - \frac{H_b(\varepsilon_0)(1-\varepsilon_1)}{1-\varepsilon_0-\varepsilon_1} + log_2\left(1+2^{\frac{H_b(\varepsilon_0)-H_b(\varepsilon_1)}{1-\varepsilon_0-\varepsilon_1}}\right)$$

where ε_0 is the probability of the spy receiving a 1 given a 0 was sent, ε_1 is the probability of the spy receiving a 0 given

a 1 was sent, and $H_b(p)$ is the binary entropy function for probability p defined as:

$$H_b(p) = -plog_2 p - (1-p)log_2(1-p)$$

The channel capacity given the experimentally determined error rates is presented in Table III. While not ideal, the effective channel capacity given the calculated error rates is still above 1.6 Mbps.

	256 B	512 B	1024 B	2048 B
ϵ_0	0.007539	0.002891	0.009258	0.005664
ϵ_1	0.0502734	0.0158984	0.0133984	0.0267188
Bit per channel	0.824027	0.926979	0.918127	0.886071
Channel capacity (Mbps)	1.62	1.83	1.81	1.75

TABLE III: Channel capacity given a binary asymmetric noisy channel transmitting at 1.96 Mbps

VI. IaaS Public Cloud 4K-Aliasing Covert Channel

Given that the robust channel demonstrated in our in-house experiments showed reasonable results, we then verify its use on Amazon EC2 and Google Compute Engine (GCE) IaaS public clouds. The instance configurations we experimented upon are shown in Table IV. We used the prior research in instance placement vulnerabilities [36] to successfully colocate VMs from two different accounts. This required launching 10 VMs from each account and then scaling up a designated account in step sizes of 5 VMs until they were physically hosted on the same core.

Cloud Provider	Instance Type	Processor
EC2	`m4.large`	2.4 GHz Intel Xeon E5-2676 v3
GCE	`n1-standard-1`	2.3 GHz Intel Xeon E5 v3

TABLE IV: Instance configurations and architecture used for demonstrating the 4K-aliasing covert channel on the public cloud. Note, GCE does not reveal the exact microarchitecture for the instance type.

Similar to our in-house 4K-aliasing experimental setup, we establish the error rate, bit rate, and channel capacity using the robust communication channel presented in Listing 3. We evaluate using 4K-aligned address in step sizes of 1024 B for transmitting a 1-bit and 2048 B for transmitting a 0-bit per the error rate analysis shown in Figure 9. The results are given in Figure 10.

We measured a 1.28 Mbps and 1.49 Mbps channel capacity while communicating across VM instances from different accounts on both the Amazon EC2 and GCE clouds. The channel capacity drops by 30% on the EC2 testbed and by 18% on the GCE testbed compared to our in-house results. This is expected as the virtualization layer on a hosted cloud induces external noise. This is reflected in the increased error rate recorded. Nevertheless, our results demonstrate the practical use of the 4K-aliasing covert channel once the trojan and spy instances are colocated on the same machine.

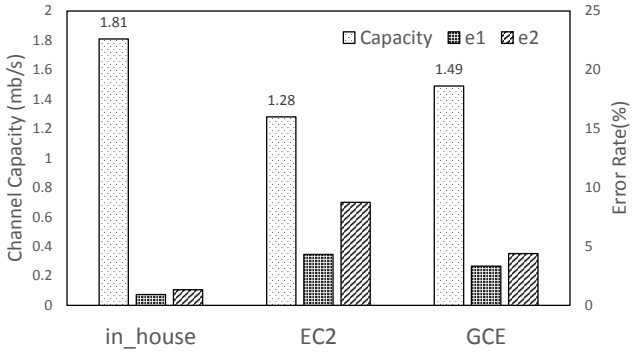


Fig. 10: Error rate and channel capacity results from in-house, EC2, and GCE 4K-aliasing covert channel experimentation.

VII. Multi-Tenancy Detection

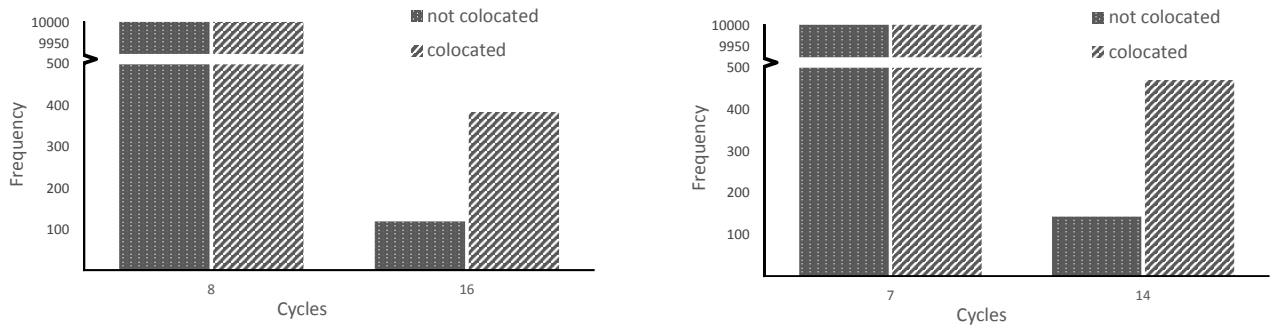
In our IaaS channel capacity experiment on the EC2 and GCE public cloud we relied upon prior colocation detection methods. Now we describe a new method for colocation detection using our 4K-aliasing timing channel. For the purposes of our analysis, we need to be able to distinguish between 4K-aliasing caused by background noise and induced 4K-aliasing. We ease the experimentation by assuming cooperative accounts, which allows us to control the number of VMs launched and the time between launches. The analysis largely follows the methodology presented by Varadarajan et al. [36] and we encountered similar challenges.

Separating 4K-aliasing from Noise. To accurately detect a cooperative multi-tenant, we must distinguish unintended 4K-aliasing events from intentional ones. To manage such noise, we run an experiment that measures 4K-aliasing without a cooperative VM sending a 4K-aliasing signal and then another with a 4K-aliasing signal being sent. We scale up the number of VMs from 1 instance to 20 instance pairs and repeat the measurement 5 times.

Launch Strategy. We launch pairwise sender and receiver VMs with the prior colocation placement vulnerabilities in mind in mind. We first launch a sender VM and then wait 1 hour before launching a receiver VM in the same zone to ensure a best case colocated launch. In all test cases, we use `us-east-1` for our EC2 testbed and `us-central1-c` for our GCE testbed. Each instance is configured as a single vCPU which is executed as a single hardware hyperthread.

Detection Tests. When the sender VMs launch, they continuously sends an oscillating 1-bit and 0-bit. The receiver polls the 4K-aliasing event for roughly 10 seconds. To decrease the testing time, we employ a naive methodology of launching all sender messages at once and then sequentially launching receiver VMs. As only one sender and receiver VM will colocate as a hyperthread, we can accurately detect multi-tenancy if any of the receiver VMs display the 4K-aliasing event outside of the noise threshold.

Experimental Results. The results of our multi-tenant detection scheme using 4K-aliasing are shown in Figure 11. They reveal two features. First, the cycle latency degradation



(a) EC2 (b) GCE

Fig. 11: Frequency distribution of 4K-aliasing event recorded for non-colocated instance pairs (8/7 cycles) and colocated instance pairs (16/14) cycles. Multi-tenancy was detected after launching 12 (EC2) and 14 (GCE) sender/receiver instance pairs.

due to 4K-aliasing between sender and receiver is clearly distinguishable compared to background noise. Roughly between 100 and 150 4K-aliasing events are measured within a 10 second window while all sender VMs are launched, but not transmitting a 4K-aliased store. On the other hand, while the sender VM is transmitting a 4K-aliased store we record roughly 360 to 480 4K-aliasing events. Second, the detection threshold was achieved reliably after scaling the number of sender and receiver VMs to 14 instance pairs. During experimentation we found that beyond 4 instance pairs we were able to achieve multi-tenant detection. However, detection was measureable in only 1 of the 5 test cases so they were discounted. We considered multi-tenant detection to be successful upon agreement for the majority of the test cases.

Limitations. Despite the positive results, they rely upon the fact that our multi-tenant demonstration requires cooperative account holders. This allows us to utilize prior work on placement vulnerability strategies to optimize our chances of launching colocated instances. Further, our results were not collected under a heavy load and potentially at non-peak hours. Finally, we largely used free trial accounts, which could possibly be underutilized and therefore less affected by disparate workloads.

VIII. Mitigating the 4K-aliasing Side Channel

Clearly, disabling hyperthreading is a straightforward method to mitigate the 4K-aliasing timing channel. However, this will largely result in increased end-user costs as dedicated instances result in increased operational expenses for the cloud provider. For security minded cloud users willing to pay extra this is a ideal solution. On the other hand, end-users using IaaS public clouds for general purpose workloads requiring little uptime will end up overpaying because they underutilize datacenter resources.

The IaaS market appears to agree with this analysis. The majority of EC2 and GCE instance types enable hyperthreading by default. Specialized instances can of course be purchased, but it is unclear how many users opt for these options. Further, Microsoft Azure, which previously disabled hyperthreading in all instance types it offered are now migrating towards SMT enabled instances [22]. In other words, the IaaS market is likely to keep default instance types SMT enabled.

In addition, CPU vendors are likely to continue leveraging hyperthreading for its various performance benefits. The underlying causes of the 4K-aliasing timing channel leverages an integral component of Intel microarchitecture, which allows significant speed-up when handling memory operations. Memory reads and writes must be allowed to issue speculatively and execute out-of-order lest we revert CPU design to single cycle pipelines. The memory order buffer saves CPU cycles, improves instruction throughput, makes better use of memory traffic bandwidth, and frees resources allowing more computation to be performed on average. Complete elimination of the underlying mechanism causing 4K-aliasing is, therefore, unlikely.

IX. Conclusion and Future Work

We have demonstrated, for the first time, a novel 4K-aliasing timing channel. We demonstrate through extensive analysis a robust covert communication channel deployable in IaaS clouds capable of transmitting at up to 1.49 Mbps. We also show the 4K-aliasing timing channel can be used in multi-tenancy detection while only launching a relatively small number of cooperating VMs on both Amazon EC2 and GCE. As future work, we aim to evaluate the timing channel as a practical side-channel and to further investigate the applicability of same-core shared resources as timing channels in the public cloud.

Acknowledgment

This work is partially supported by the Department of Energy through the Early Career Award (DE-SC0016180). Mr. Orlando Arias is also supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 1144246. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy or the National Science Foundation.

REFERENCES

[1] O. Aciiçmez, "Yet another microarchitectural attack:: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*. ACM, 2007, pp. 11–18.

[2] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[3] O. Aciiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *CT-RSA*, vol. 8. Springer, 2008, pp. 256–273.

[4] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*. IEEE, 2007, pp. 80–91.

[5] Amazon, Inc., "Amazon ec2 dedicated instances," 2017. [Online]. Available: https://aws.amazon.com/ec2/purchasing-options/dedicated-instances/

[6] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 623–639.

[7] M. F. Chowdhury and D. M. Carmean, "Maintaining processor ordering by checking load addresses of unretired load instructions against snooping store addresses," Feb. 3 2004, uS Patent 6,687,809.

[8] P. Church and A. Goscinski, "Iaas clouds vs. clusters for hpc: A performance study," in *Cloud Computing 2011: The 2nd International Conference on Cloud Computing, GRIDS, and Virtualization*. [IARIA], 2011, pp. 39–45.

[9] I. Corporation, "Intel® 64 and ia-32 architecture memory ordering white paper," 2007.

[10] ——, "Using intel® vtune™ amplifier xe to tune software on the 4th generation intel® core™ processor family," 2013.

[11] ——, "Intel® 64 and ia-32 architectures optimization reference manual," 2017.

[12] ——, "Intel® 64 and ia-32 architectures software developers manual," *Volume 3A: System programming Guide, Part 1*, vol. 3A, 2017.

[13] J. Doweck, "Inside intel® core microarchitecture," in *Hot Chips 18 Symposium (HCS), 2006 IEEE*. IEEE, 2006, pp. 1–35.

[14] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 843–857.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[16] ——, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 10, 2016.

[17] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, 2011.

[18] G. Forney, "Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference," *IEEE Transactions on Information theory*, vol. 18, no. 3, pp. 363–378, 1972.

[19] G. Galante, L. C. E. De Bona, A. R. Mury, B. Schulze, and R. da Rosa Righi, "An analysis of public clouds elasticity in the execution of scientific applications: a survey," *Journal of Grid Computing*, vol. 14, no. 2, pp. 193–216, 2016.

[20] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 368–379.

[21] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl, "Cloudoscopy: Services discovery and topology mapping," in *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*. ACM, 2013, pp. 113–122.

[22] Hillger, Brian, "Price reductions on l series and announcing next generation hyper-threaded virtual machines." [Online]. Available: http://bit.ly/2gYVunn

[23] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[24] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 639–650.

[25] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud." *IACR Cryptology ePrint Archive*, vol. 2015, p. 898, 2015.

[26] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[27] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[28] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: cross-cores cache covert channel," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 46–64.

[29] S. M. Moser, "Error probability analysis of binary asymmetric channels," *Dept. El. & Comp. Eng., Nat. Chiao Tung Univ*, 2009.

[30] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[31] G. Paoloni and I. Corporation, "How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures," 2010.

[32] D. A. Patterson, "The data center is the computer," *Communications of the ACM*, vol. 51, no. 1, pp. 105–105, 2008.

[33] C. Percival, "Cache missing for fun and profit," 2005.

[34] 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 *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[35] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[36] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift, "A placement vulnerability study in multi-tenant public clouds." in *USENIX Security Symposium*, 2015, pp. 913–928.

[37] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 473–482.

[38] 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 (TON)*, vol. 23, no. 2, pp. 603–614, 2015.

[39] 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 workshop on Cloud computing security workshop*. ACM, 2011, pp. 29–40.

[40] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud." in *USENIX Security Symposium*, 2015, pp. 929–944.

[41] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security Symposium*, 2014, pp. 719–732.

[42] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[43] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.