NetCAT: Practical Cache Attacks from the Network

Michael Kurth*§, Ben Gras*, Dennis Andriesse*, Cristiano Giuffrida*, Herbert Bos*, and Kaveh Razavi*

*Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands
m.kurth@vu.nl, beng@cs.vu.nl, da.andriesse@few.vu.nl
{kaveh, herbertb, giuffrida}@cs.vu.nl

§Department of Computer Science
ETH Zurich, Switzerland
kurthm@ethz.ch

***Abstract*—Increased peripheral performance is causing strain on the memory subsystem of modern processors. For example, available DRAM throughput can no longer sustain the traffic of a modern network card. Scrambling to deliver the promised performance, instead of transferring peripheral data to and from DRAM, modern Intel processors perform I/O operations directly on the Last Level Cache (LLC). While Direct Cache Access (DCA) instead of Direct Memory Access (DMA) is a sensible performance optimization, it is unfortunately implemented without care for security, as the LLC is now shared between the CPU and all the attached devices, including the network card.**

In this paper, we reverse engineer the behavior of DCA, widely referred to as Data-Direct I/O (DDIO), on recent Intel processors and present its first security analysis. Based on our analysis, we present NetCAT, the first Network-based PRIME+PROBE Cache Attack on the processor's LLC of a remote machine. We show that NetCAT not only enables attacks in cooperative settings where an attacker can build a covert channel between a network client and a sandboxed server process (without network), but more worryingly, in general adversarial settings. In such settings, NetCAT can enable disclosure of network timing-based sensitive information. As an example, we show a keystroke timing attack on a victim SSH connection belonging to another client on the target server. Our results should caution processor vendors against unsupervised sharing of (additional) microarchitectural components with peripherals exposed to malicious input.

I. Introduction

Different processes running on a CPU may share microarchitectural components such as CPU caches for reasons of efficiency. Given that these processes may belong to different security domains, this sharing violates process isolation at the microarchitectural level. Many existing attacks show that an attacker can examine the modifications made to the shared microarchitectural state by a victim operation to derive secret information. Examples include leaking secret information from victim processes [1, 2, 3, 4, 5, 6, 7] and cloud virtual machines [8, 9, 10, 11]. These attacks can even be launched from JavaScript in the browser [12, 13, 14]. The underlying assumption behind all these attacks is that the attacker needs either code execution or the ability to leverage victim code running on the target processor to be able to observe modifications in the microarchitectural state.

In this paper, we challenge this assumption and show that on modern Intel processors, any attached peripheral such as the Network Interface Card (NIC) can directly manipulate and observe the state of the processor's Last-Level Cache (LLC). This is possible because the processor enables peripherals to perform Direct Cache Access (DCA) instead of Direct Memory Access (DMA) for improved I/O performance. We explore the security implications of this widely-deployed mechanism for the first time and show that an attacker can abuse it to leak sensitive information from any peripheral that is exposed to malicious input. To exemplify the threat, our proof-of-concept exploit, NetCAT, can target a victim client of a DCA-enabled server to leak that client's private keystrokes in an SSH session.

Existing microarchitectural attacks To leak sensitive information with a microarchitectural attack, the attacker needs to be able to measure the modification that a victim makes to a part of the microarchitectural state. For example, in a PRIME+PROBE attack, the attacker first *primes* a shared resource to put it in a known state. In the second step, the attacker *probes* the same resource set by accessing it again. If the accesses are now slower, it means that a victim's secret operation has accessed the resource. This observation is enough to leak secret information like cryptographic keys from a process or a VM running on the same processor. Furthermore, similar attacks have been mounted by executing JavaScript [12, 13, 14, 15] and even through the network when interacting with a vulnerable process [16, 17]. The JavaScript-based attacks, while strengthening the threat model by not requiring native code execution, are amenable to sandbox-level mitigations [18] and still require the victim to execute the attacker's JavaScript code. Truly remote, network-only, attacks relax the requirement for code execution on the target machine, whether JavaScript or native code, by instead interacting with a remote vulnerable process that happens to contain specific gadgets (or otherwise cooperates with the client) on the remote processor. Because these attacks do not have direct visibility of the CPU cache state, they require a large number of time-consuming network measurements as well as a vulnerable (or cooperative) victim process, making them hard to use in practice.

NetCAT This paper shows that it is possible to detect a single LLC cache hit or miss on a remote processor from the network on modern Intel platforms that are equipped with DDIO since 2012. This is possible since data center networks have become increasingly fast, to the point that they allow a remote process

to observe the timing difference between a network packet that is served from the remote processor's cache versus a packet served from memory as we show for the first time. This basic capability allows us to build NetCAT, a PRIME+PROBE attack on a portion of the remote processor's LLC. NetCAT can observe the activity of the remote processor as well as other network clients that send traffic to the remote processor. For instance, using these observations, NetCAT can perform a keystroke timing analysis to recover words typed by a victim client in an SSH session with the target server. Compared to a native local attacker, NetCAT's attack from across the network only reduces the accuracy of the discovered keystrokes on average by 11.7% by discovering inter-arrival of SSH packets with a true positive rate of 85%.

For NetCAT to surgically manipulate the remote LLC state and perform our end-to-end keystroke timing analysis, we need to overcome a number of challenges. First, we must reverse engineer how DDIO [19, 20, 21] (Intel's DCA technology) interacts with the LLC, information that is undocumented and unknown prior to our work. Second, to perform PRIME+PROBE, we must blindly build remote eviction sets by crafting the right sequences of network packets. We show how a variation of the work by Oren et al. [14] can successfully build eviction sets over the network. Third, to perform our end-to-end attack, we must track when the remote machine receives SSH packets from the victim client. We describe a novel cache set tracking algorithm that recovers the state of the NIC's ring buffer, which we use to track distinct SSH traffic. We show that the extracted cache activity is strong enough to perform the necessary keystroke timing analysis and to recover typed words successfully.

Contributions In summary, our contributions are as follows:

- We reverse engineer and provide the first detailed analysis of Intel DDIO, a technology that directly places I/O traffic in the processor's LLC.
- We implement NetCAT, a practical network-based PRIME+PROBE attack on the LLC of a remote processor.
- We implement an end-to-end keystroke timing attack using NetCAT on the SSH session of a victim client on the target server. A demo of the attack and additional information about NetCAT is available at https://www.vusec.net/projects/netcat.

II. Background

In this section, we discuss the memory hierarchy, general cache attacks and DCA, all of which are building blocks for NetCAT. Furthermore, we discuss existing remote cache attacks.

A. Memory Hierarchy

In order to speed up accesses to main memory, most commodity processor architectures have multiple levels of caching. The caches that are accessed first (closer to the CPU core) are usually smaller but faster than the caches closer to the main memory. The caches are in place to leverage spatial and temporal access patterns. In recent commodity processors, we often find a three-level hierarchy in which each CPU core has a dedicated Level 1 (L1) and Level 2 (L2) cache. Moreover, the CPU cores share a single Last Level Cache (LLC). Because it is shared among cores, the LLC has a special role in cross-core data accesses and recently also in PCIe data exchange, as we discuss later in this section. Aside from cache speed and size, cache design involves key properties such as cache inclusivity versus exclusivity (or non-inclusivity) with respect to other caches. As an example, in prior Intel server processors, the LLC was inclusive with respect to L2, meaning that the LLC had copies of all L2 cache lines. This changed with the Skylake X microarchitecture, where the LLC is non-inclusive with respect to L2, so that a cache line from L2 may not exist in LLC.

B. Cache Attacks

Cache attacks belong to the more general class of microarchitectural attacks. The broad idea is to exploit the use of shared resources on or around the CPU. An attacker leverages these shared resources to typically steal (leak) information. In cache attacks, the attacker builds a side channel based on the timing information that can be observed in data fetches from the different levels of caches or main memory. Such timing information can be misused to gain information about other processes and therefore reveal secrets. A successful side-channel attack circumvents higher-level security mechanisms, e.g., privilege separation.

Osvik et al. [1] pioneered the idea of PRIME+PROBE in the context of L1 cache attacks. The PRIME+PROBE algorithm consists of three steps: (1) Build cache eviction sets, (2) *Prime*: Bring the cache to a known state by accessing the eviction sets, (3) *Probe*: Access the eviction set again, during a victim secret operation. Higher access times imply sets which the victim process accessed.

Ristenpart et al. [22] used the PRIME+TRIGGER+PROBE load measurement technique to detect keystroke activity on L1 and L2 caches, allowing an attacker to infer activity on virtual machines (VMs) that timeshare a core. Liu et al. [9] extended PRIME+PROBE to the LLC, allowing the attacker to extract secrets from co-hosted VMs without the need to share the same core.

Browser-based Cache Attacks Browser-based cache attacks strengthen the threat model of native code cache attacks by executing from sandboxed JavaScript environments. They exploit the fact that this JavaScript code executes logically sandboxed but not at the microarchitectural level. Oren et al. [14] introduced a non-canonical PRIME+PROBE attack from JavaScript without any direct access to physical or virtual addresses. Our eviction set building for NetCAT is based on their approach. Gras et al. [13] used an EVICT+TIME attack to break ASLR from JavaScript. Lipp et al. [23] launched a keystroke timing attack from JavaScript to spy on the user typing in the address bar. Frigo et al. [12] exploited the integrated GPU with microarchitectural attacks to escape the Firefox JavaScript sandbox on Android. All these attacks face a number of practical challenges such as needing a

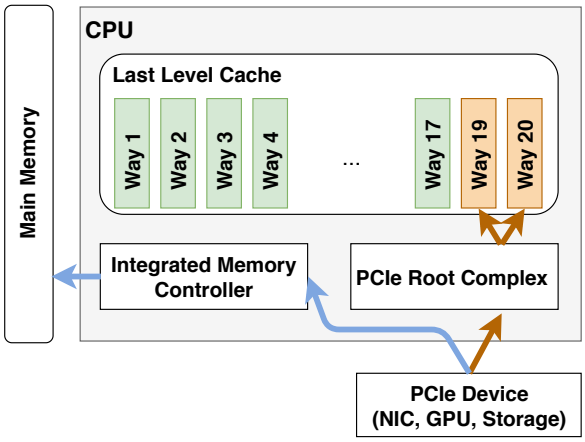

Fig. 1: Difference between direct cache access (orange) and direct memory access (blue). Additionally, the available write allocation cache lines for direct cache access in orange versus the others in green.

high-precision timer and the presence of other sandbox-level mitigations in modern browsers [18] and also require the victim to execute the attacker's JavaScript code.

C. Remote Cache Attacks

Existing, remote, network-only cache attacks use the fact that they can observe the overall execution time after sending a request to a web server. [24] shows a realistic remote-only attack on OpenSSL. Bernstein [17] showed complete AES key recovery from known-plaintext timings which Neve et al. [25] further improve. Schwarz et al. [16] demonstrated a network-based Spectre attack targeting specific code patterns (or gadgets) in a remote victim to disclose information. All of these attacks are highly target-specific, and take a long time (hours to days) due to the need to average over large amounts of network packets to remove the network noise. Furthermore, they both require a vulnerable (or otherwise cooperative) victim server program containing specific gadgets. Such gadgets must ensure the input-dependent operation accounts for most of the overall execution time in order to leak information. In contrast, our work is a generic, remote cache channel for which we show a realistic and non-exhaustive list of example attack scenarios that do not depend on the software target.

D. Direct Cache Access

In traditional architectures, where the NIC uses DMA, the memory latency alone quickly becomes the bottleneck in network I/O-focused workloads on 10 Gb/s interfaces [19]. To alleviate the DRAM bottleneck, [19] proposes DCA, an architecture where PCIe devices can directly access the CPU's LLC. The DCA cache region is not dedicated or reserved in the LLC, but allocating writes are statically limited to a portion of the LLC to avoid trashing caused by I/O bursts or unconsumed data streams. Figure 1 illustrates a traditional DMA access (the blue access flow) versus a DCA access (the orange access flow).

Initially, Intel implemented DCA using a prefetch hint approach, in which a DMA write would trigger a memory prefetch into the LLC after arriving in main memory, but this required support from the device to hint at DCA and device driver support to prefetch these DCA hints. Starting with the Intel Xeon E5 and Xeon E7 v2 processor families in 2011, server-grade CPUs implement DCA under the name of Data Direct I/O Technology (DDIO) [20, 21, 26, 27], which is entirely transparent to software and hardware. With DDIO, a server machine is able to receive and send packet without any trips to main memory in the optimal case. We will further describe DDIO and reverse engineer its behavior in the next sections.

III. Threat Model

Our threat model targets victim servers with recent Intel processors equipped with DDIO, enabled transparently by default in all Intel server-grade processors since 2012. We assume the attacker can interact with a target PCIe device on the server, such as a NIC. For the purpose of instantiating our attack in a practical scenario, we specifically assume the attacker is on the same network as the victim server and can send packets to the victim server's NIC, thereby interacting with the remote server's DDIO feature. In particular, in our example we launch a cache attack over the network to a target server to leak secret information (such as keystrokes) from the connection between the server and a different client. While we mostly focus on client to client attacks in this paper, DDIO could be exploited in other settings as well. Section IX looks at other threat models where our NetCAT can potentially apply to applications on the target server processor (rather than other clients) as well as other PCIe devices.

Our example attack (Section IV) abuses RDMA technology in the NIC to control the memory location which a transmitted packet accesses, as well as the low-latency offered by today's high-speed networks. RDMA is now available in the clouds of many major providers and many data centers such as Azure, Oracle, Huawei and Alibaba [28, 29, 30, 31]. In virtualized cloud settings, NetCAT can target *any* VM on the target server, as long as it can communicate with *only* one of these VMs through a virtualized RDMA interface. In addition, if the attacker's VM (or virtualized server) is connected to a storage server with RDMA using protocols such as SMBDirect [32] or NFS [33], then NetCAT enables an attacker to spy on other clients that connect to the storage server. Similarly, cloud key-value service [34] and applications that integrate RDMA to improve their performance, including big data [35, 36], machine learning [37], and database [38] could be abused by NetCAT-like attacks.

IV. Attack Overview

Our goal is to exploit the fact that the DDIO-enabled application server in Figure 2 has a shared resource (the LLC) between the CPU cores and the PCIe devices. We will show that by abusing the sharing, we can leak sensitive information from the LLC of the application server. There are many

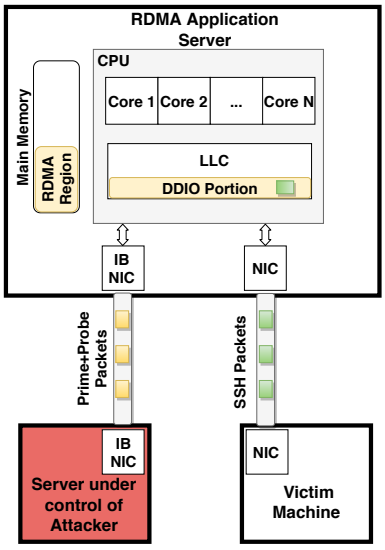

Fig. 2: Assumed topology for attacks

potential ways to exploit DDIO. For instance, an attacker with physical access to the victim machine could install a malicious PCIe device to directly access the LLC's DDIO region. Our aim in this paper is to show that a similar attack is feasible even for an attacker with only remote (unprivileged) network access to the victim machine, without the need for any malicious PCIe devices.

To this end, we make use of RDMA in modern NICs. RDMA bypasses the operating system at the data plane, providing remote machines with direct read and write access to a previously specified memory region. The OS is responsible for setting up and protecting this RDMA region. However, as we show later in more detail, when DDIO is enabled, RDMA reads and writes have access not only to the pinned memory region but also to *parts* of the LLC. Mellanox further motivates the use of RDMA [39] for minimizing the performance-degradation due to defenses required to protect against the latest speculative execution attacks [15, 40]. Ironically, RDMA makes it easier to perform network-based cache attacks as we show in this paper.

Figure 2 illustrates our target topology, which is common in data centers. The attacker controls a machine which communicates over RDMA to an application server that supports DDIO and also services requests from a victim on a separate NIC. With this, we show that we can successfully spy on another PCIe device. However, we do not rely on such separation, i.e., we could also spy on the same PCIe device where we issue our PRIME+PROBE packets. In our adversarial attack, we will assume that a victim client types in sensitive information over an `ssh` connection. The aim of the attacker is finding out the keystrokes typed by the victim client using the PRIME+PROBE packets. There are three main challenges that we need to overcome for implementing our attack:

C1 Inner workings of DDIO. Our attack requires a precise knowledge of the effects of DDIO operations, the DDIO allocation limitation, and the feasibility of detecting cache hits and misses over the network.

C2 Remote PRIME+PROBE**.** Our attack requires us to remotely build cache eviction sets for our PRIME+PROBE attack, without knowledge of virtual or physical addresses of the RDMA memory region on the remote machine, introducing unique challenges in measuring cache activity over the network.

C3 End-to-end attack. To implement an end-to-end attack, we require a solid understanding of what sensitive data may reside in the DDIO-reachable part of the LLC and is eligible for leaking.

We address these challenges in the following sections:

C1: Inner workings of DDIO. Section V analyzes DDIO in depth. First, we find suitable remote read and write primitives using DDIO. Next, we show that it is possible to detect LLC hits over the network via DDIO. Furthermore, we confirm the known DDIO restrictions on allocating writes, and we discover that the precise percentage of the LLC accessible to allocating writes differs between Intel CPU models.

C2: Remote PRIME+PROBE In Section VI, we use our newly obtained understanding of DDIO to remotely create cache eviction sets. We adapt existing PRIME+PROBE algorithms to cope with the challenges of network noise and with the slower read/write operations compared to native code.

C3: End-to-end attack Finally, we showcase various DDIO attack scenarios in Section VII. First, we build a covert channel between a network client and an unnetworked, cooperating sandboxed process on a remote machine. Second, we build a covert channel between two cooperating network clients running in two separate networks, without any direct communication paths. Third, we describe an adversarial keystroke timing attack on a victim SSH connection of another client by remotely measuring cache activity caused by SSH packets, described in Section VIII. Our adversarial setup is sketched in Figure 2.

V. Reverse Engineering DDIO

To remotely measure cache activity, we require remote read/write primitives provided by the PCIe device's DDIO capabilities. This section discusses how we build these required primitives to mount our attack, while in the process elaborating on the relevant details of DDIO.

A. Access Latencies

The first step in implementing our attack is to determine whether we can measure the timing difference between cache hits and memory reads over the network. We used two servers (Intel Xeon Silver 4110) running Ubuntu 18.04.1 LTS, each with a Mellanox ConnectX-4 Infiniband NIC (produced in 2016). We used one of the servers as an RDMA server and the other one as a client. As a baseline, the *ib_read_lat* latency benchmark measured an average latency between our two machines of 1,550 ns, with a standard deviation of 110ns and

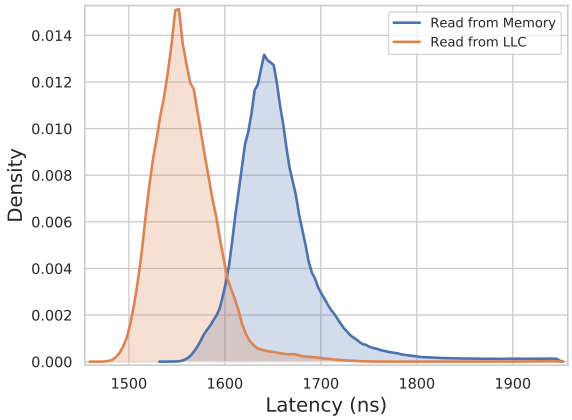

Fig. 3: Distributions of DDIO reads served from the LLC and from main memory as measured over the network. Distribution estimated with kernel density over 99th percentile data.

a 99th percentile of 1,810 ns. To send one-sided RDMA reads and (in later experiments) writes, we use *libibverbs*.

In our first experiment, we iterated over 50,000 memory addresses 150 times. In each iteration, we issued two RDMA reads to the same memory address and measured the time taken for each result to arrive back at the client. We measured no significant difference between the two accesses. Closer inspection revealed that this is because an address read via DDIO that is absent in the LLC is served directly from main memory *without* being allocated in the LLC (i.e., subsequent reads to an uncached memory location remain uncached).

In a second experiment, we instead issued the following sequence of operations in each iteration: Read(x) - Write(x) - Read(x). The idea is that the first read is served from main memory, while the read after the cache-allocating write is served from the LLC, allowing us to measure a baseline difference between memory reads and cache hits. Figure 3 shows that the resulting distributions of the two types of reads are distinguishable. Section VI discusses mechanisms to further distinguish LLC-based reads from memory reads.

B. DDIO Cache Ways

As previously discussed, DDIO limits write allocations to prevent cache trashing from PCIe devices. Because this limitation impacts our ability to create eviction sets and mount cache attacks, we study the mechanics of the limitation. To this end, we build a pool of addresses that map to the same cache set and are accessible via RDMA. We achieve this by allocating a large buffer on the RDMA server and then applying the method of Maurice et al. [41] to find pages with the same LLC color. We then remap the RDMA buffer so that the RDMA client can directly access these addresses via DDIO, allowing us to remotely create eviction sets without yet knowing the exact algorithm needed to achieve this in the general case. With the help of this colored RDMA buffer, we are able to explore the layout of DDIO ways in the LLC.

More specifically, our experiment repeatedly writes to n addresses in the colored buffer (within the same cache set) and then reads those same addresses, measuring whether these reads are served from cache. We start with $n = 0$ and increment n after each round. The expectation is that this allows us to determine the DDIO write allocation limit by finding the n where the number of cache hits becomes constant.

TABLE I: Overview of CPU models used in our experiments, and summary of our experimental findings on the DDIO allocation ways and allocation limit.

CPU	LLC	DDIO
Xeon Haswell E5-2630 v3	20 MB (20 ways), incl	2 ways (10%)
Xeon Skylake Silver 4110	11 MB (11 ways), n-incl	2 ways (18%)

We perform this experiment on two machines equipped with Intel Xeon E5-2630 v3 processors running CentOS 7.4, each with a Mellanox ConnectX-3 Infiniband NIC (produced in 2014). Each machine's LLC has a size of 20 MB and is 20-way set associative according to the specifications. As shown in Figure 4, starting with $n = 2$ (Write 0-1), we see a constant pattern of two addresses being served from the cache and the rest of the addresses being served from main memory. The memorygram is darker for low latencies and lighter for high latencies. This experiment yields strong evidence that there are two DDIO ways on our test machines. This is also supported by the original Intel documentation [26], which states that the write allocation limit is 10% of the LLC (i.e., 2 ways out of a total of 20 ways is 10% of the LLC). On the Intel Xeon Silver 4110, our experiments also reveal two DDIO ways, which, given that this model uses an 11 MB and 11-way set associative LLC, means that the DDIO write allocation limit is instead around 18.2% of the LLC, as shown in Table I.

Figure 4 additionally yields insights into the cache replacement policy used for the DDIO region in the LLC. As we can see, the last two written values are served from the LLC. Our further experiments with random write and read operations suggest that the replacement policy is most likely evicting the least recently used (LRU) cache lines in the DDIO region.

VI. Remote Prime+Probe

In order to launch a successful remote PRIME+PROBE attack, we need write and read primitives on a memory region on the remote machine. As described in Section V, RDMA gives us these capabilities on the LLC.

A. Creating a Remote Eviction Set

The first step of PRIME+PROBE is to build cache eviction sets [1]. In our case, under the write allocation restrictions of DDIO, we do not build eviction sets for all cache sets in the LLC, but only for the limited number of cache ways accessible by DDIO. Building eviction sets and later on using it to leak data relies on basic RDMA operations, so any application that uses one-sided RDMA and allows an RDMA client to write data can be used for NetCAT attacks. We exemplify this on RDMA-memcached [42], a key-value store with RDMA support. RDMA-Memcached implements GET and SET operations where memory allocation is split into

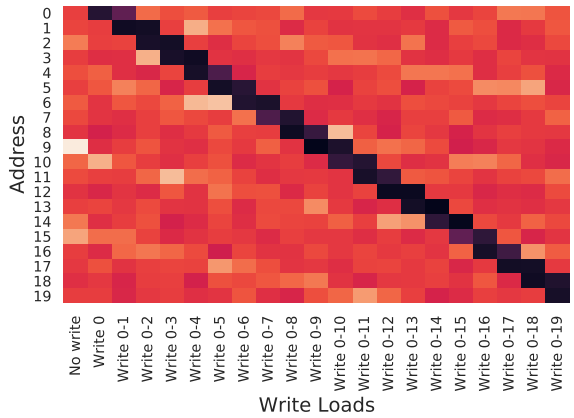

Fig. 4: Memorygram of DDIO-ways experiment. Darker colors imply faster, and lighter colors imply slower access times. From left to right, we increase the number of n addresses written before reading all addresses back (between 0 and 20). The latencies correspond to read access times.

1 MB sized chunks. In order to allocate a large enough memory region to build an eviction set, we allocate multiple 1 MB sized key-value items. Once the objects are allocated, they can be accessed at arbitrary offsets with basic one-sided RDMA operations.

One challenge in building the eviction sets is that we have no knowledge of the virtual or physical addresses of the RDMA memory region on the remote machine. However, we can control our accesses through an offset relative to a base, combined with the knowledge that allocated memory chunks are page aligned. Oren et al. [14] engineered a non-canonical PRIME+PROBE for a similar problem when they attacked the LLC from JavaScript. We base our approach on their algorithm, with the caveat that we must address challenges resulting from running the algorithm over the network. These challenges include resilience against timing shifts caused by network variance, and the involvement of a second machine in the measuring process. Moreover, read and write operations over the network are orders of magnitude slower than run locally.

The broader idea of the algorithm [14] is to use a set S of page-aligned addresses, all with the same offset from the page start, and a candidate address x. The set is initially quite large so that it naturally forms an eviction set for address x. The algorithm then reduces the set by iteratively removing addresses and checking whether the set still forms an eviction set. Using this *backwards-selection* strategy, the algorithm creates a minimal eviction set of size equal to the number of cache ways. After finding one eviction set for the given offset within a page, the algorithm can build eviction sets for the rest of the offsets. With a page size of 4KB and a cache line size of 64B, this yields an additional 63 eviction sets.

Our first naïve approach used multiple rounds to measure whether the set S still forms an eviction set, to account for the measurement noise over the network. However, this made the algorithm quite slow, especially when profiling all the available cache sets (magnitude of hours). We therefore introduced a number of optimizations.

Optimization 1 As a first optimization, we introduce a *forward-selection* algorithm that creates a possible smaller set S that evicts address x. We start from an empty set S and on each iteration we add one more addresses to S until we measure an eviction. This selection process reduces the number of addresses in S from thousands to hundreds on average. The optimization works well because the DDIO ways are a subset of the full cache ways, e.g., on an Intel Xeon E5-2630 v3 CPU, we only have to find two out of the potential twenty addresses that form an eviction of address x. This reduced set S is then the input of the *backwards-selection* algorithm. The *forward-selection* algorithm is detailed in Appendix A.

Optimization 2 The second optimization concerns the *backwards-selection* algorithm. In the original algorithm, the set S is first written completely and then written again while leaving one address s out on each iteration. This compares the cache miss time of x to the miss/hit time of x depending on whether $S \setminus s$ is still an eviction set. In our approach, we instead first measure a cache hit on x by writing and reading x, and then compare the access time to $S \setminus s$. This works because S always evicts x on a successful profiling run, while reducing the number of write operations in this step by a factor of two.

Optimization 3 As a third optimization, instead of only removing one address s from set S in the *backwards-selection* process, we implement a dynamically adjusting algorithm that removes multiple addresses from S at the same time. The algorithm increases the number of addresses to be removed by ten after the previous iteration successfully decreases S. Contrary, the algorithm decreases the number of addresses to be removed by one if the previous iteration did not decrease S. The number of addresses to be removed is bound to a maximum of half the size of S. The adjustment algorithm is disabled when the size of S is small, as adjusting it then can impact the runtime negatively with additional iterations needed. We outline the updated *backwards-selection* algorithm in Appendix B. In recent research, Vila et al. [43] provide an optimized algorithm to reduce eviction sets to minimal eviction sets. Applying the new algorithm could further improve the performance of the *backwards-selection*.

Optimization 4 Our final optimization introduces a clean-up step. After successfully building an eviction set for one cache set, we iterate over the whole pool of addresses to find other addresses that also map to this same cache set. Either they were not part of S in the first place, or they were redundant in S and can be removed from the minimal eviction set. This clean-up step helps to shrink the pool of addresses considered by the *forward-selection* algorithm (and the rest of the pipeline) for the subsequent cache sets.

Resilience Our experiments employ multiple strategies to cope with network noise, network queuing, and the side effects of

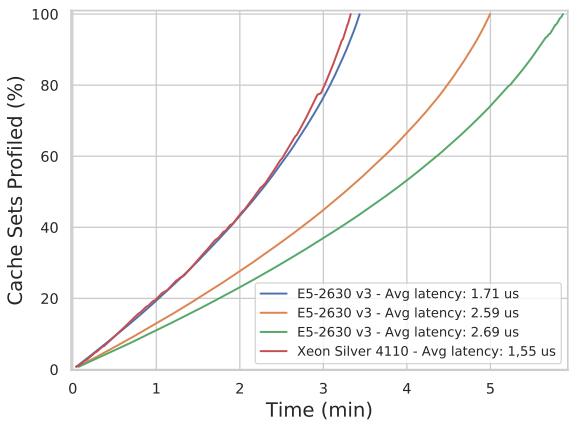

Fig. 5: Cumulative cache set profiling evaluation with different machine combinations in a data center.

the measurement machine itself. First, we use multiple measurement rounds and take the median latency measurement. This simple yet effective approach significantly improves the stability of building the eviction sets. The number of rounds is a trade-off between performance and reliability, and can be adjusted to the noise factors in different environments. However, note that we can only use this approach if we control the operation we want to measure. This is the case when building eviction sets, but as we will see later, it is not the case for keystroke detection.

Second, as shown in Section V, DDIO reads do not cause cache allocations if they are served from main memory. Therefore, we know a priori that a DDIO read does not change the state of the LLC. We can use this behavior to our advantage by reading the same address multiple times consecutively and taking the median latency of these *micro rounds*. Such *micro rounds* are especially useful when complete rounds are not possible.

Finally, the three different stages (forward-selection, backward-selection, and clean-up) have multiple built-in sanity checks. In case a test fails, the pipeline either goes back to the previous stage or completely restarts the profiling for this cache set.

B. Evaluation

We evaluated the remote eviction set building algorithm on the DAS-5 cluster [44]. This allowed us to test the algorithm with machine pairs that have different latencies, depending on where they are located in the data center and with different number of switch hops. All machines have the same processor (Intel Xeon E5-2630 v3) and machine configurations. Furthermore, we evaluated the algorithm on a second Intel Xeon Silver 4110 cluster to show the generality of our attack. We used an initial pool of 5,000 page-aligned addresses to build the eviction set. We profiled a total of 16,384 cache sets (256 colors, 4KB page size).

As shown in Figure 5, the total profiling time was between 3 minutes and 19 seconds, and 5 minutes and 52 seconds. We can see that network latency has a direct influence on the time it takes to profile the whole LLC. Furthermore, the performance of the algorithm increased when fewer addresses were in the pool. This speedup is due to the clean-up step where addresses that belong to the same cache set are removed from the pool, thus reducing the search space of the algorithm over time. The shown latencies are reported by the *ib_read_lat* latency benchmark. The standard deviation of the latencies of the three cluster machine combinations was between $0.08\mu s$ and $0.10\mu s$. The standard deviation of the latencies for the Intel Xeon Silver 4110 cluster was $0.11\mu s$. In the trace of the Xeon Silver, we can also observe a sanity check failing at around minute three, at which point the algorithm recovers by restarting the current profiling round. To verify the correctness of the eviction set, we implemented a verification procedure that tests every eviction set against other addresses that are mapped to the same cache set, in order to check whether they are evicted. Furthermore, we test the eviction sets against each other to verify their uniqueness.

To conclude, we have shown that it is possible to create an eviction set for the DDIO cache lines in a data center topology in under 6 minutes.

VII. Covert Channel

In this section, we present two cooperative DDIO-based attacks. In the first scenario, we build a covert channel between two clients that are not on the same network but can send packets to a shared server. In the second scenario, we build a covert channel between a client and a sandboxed process on a server. We use the high-bandwidth covert channel protocol from Lui et al. [9], originally used to send data between two virtual machines running on the same physical machine. Similar to our covert channel, Maurice et al. [45] describe a cross-core covert channel between processes and Oren et al. [14] describe a covert channel built from JavaScript. Further, Maurice et al. [46] developed a robust and error-free covert channel protocol, which was used to transmit an SSH connection between two virtual machines. We present an adversarial network-based keystroke timing attack in Section VIII.

A. Covert Channel Between Network Clients

In the first scenario, the two clients send RDMA packets to the target server, but they do not share a common RDMA memory region (i.e., cannot communicate directly). Furthermore, the clients are not able to communicate with each other directly over the network. Such a scenario could be enforced by having two different physical networks or a logical separation between networks. From Section VI, we know that we can measure the cache activities of the whole DDIO portion of the LLC. This means we can also measure the activity of another client over the network in the LLC.

Thus, in a cooperative setting, two clients can communicate by sending packets to different offsets in their respective RDMA buffers, while the other client detects which offset was activated by the other client's packet. In our unidirectional covert channel, the first step in establishing communication is

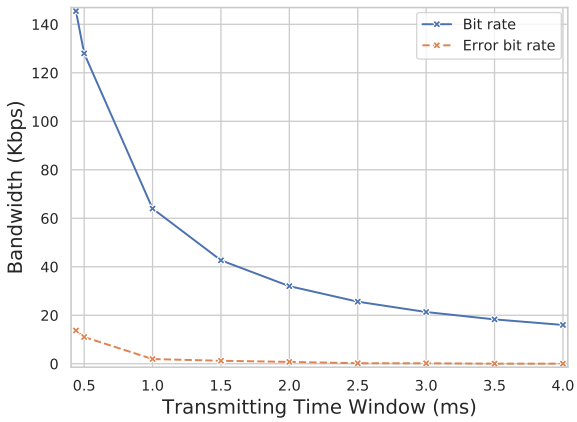

Fig. 6: Evaluation of the covert channel between two network clients. We compare the peak bit rate versus the resulting error bit rate.

to agree on which cache sets will be used for the transmission. The sender chooses a page-aligned memory location and then uses the cache sets that cover that location for communication. To synchronize, the sender then iterates over all successive cache sets in the page and sends packets (using RDMA writes) to these cache sets in a distinct pattern over a long period of time.

The receiver iterates over all profiled cache sets to detect the pattern. This allows the receiver to find the 64 cache sets (i.e., a page) that cover a page with the same color as the sender's page. The two parties have now agreed on 64 shared cache sets on the server. Therefore, in every round, the sender can transmit 64 bits by either activating or not activating each of the 64 cache sets. In order to loosely synchronize the measurements, we use the approach from Lui et al. [9]. The sender transmits the current round of information multiple times over a predefined amount of time. The receiver measures the cache activity for the same amount of time and therefore knows when a transaction round is completed. The time the receiver needs to PRIME+PROBE 64 cache sets is the minimum time window for each round.

Results The appropriate time window for the covert channel depends on the time in which the receiver can launch at least one PRIME+PROBE iteration. In our test networks [44], the smallest possible time window which reliably allowed the receiver to finish its operation within the window is 0.44 ms. This translates to a peak bandwidth of 145.45 Kb/s. Under this condition, we have an error rate of 9.43%. We evaluated multiple time windows up to a conservative choice of 4 ms (16 Kb/s). In a longer window, the receiver can launch multiple PRIME+PROBE iterations. Therefore, the receiver gains more data points, which results in a lower error rate. At the 4 ms window, we measured an error rate of 0.20%. Figure 6 illustrates our experiments with different time window sizes. Note that this simple covert channel protocol has no built-in reliability; if more reliability (i.e., redundancy) is needed, then the bandwidth of the covert channel will decrease proportionally.

B. Covert Channel to Sandboxed Process

In this scenario, we have a sandboxed process on the server that has no access to any networking capabilities. However, the sandboxed process can still write to the LLC. To build a covert channel, we observe that this scenario is similar to the previous one, except that the sandboxed process is the sender and the client is the receiver. The difference with the covert channel between the two network clients is that memory accesses by the sandboxed process do not necessarily spill into the receiver-visible LLC portion dedicated to DDIO.

In our setting, the DDIO region of the LLC consists of two cache lines (2 ways in the LLC). Thus, to ensure a successful transmission, the sandboxed process must write $n-1$ cache lines in an n-way set associative LLC, guaranteeing that the write is visible in the DDIO region. In a non-inclusive LLC, the process must also consider the L2 cache, since the L2 must be filled before data is written into the LLC. Regardless of whether the LLC is inclusive, the sandboxed process must first create an LLC eviction set, following strategies form prior work [7, 14]. Once eviction sets are found for 64 different cache sets, the covert channel can be built similarly to the case with two network clients, the main difference being that instead of one write per cache set, the sandboxed process must write the entire eviction set per targeted cache set. The receiver can then use PRIME+PROBE to monitor these evictions from the network.

Results Similar to our covert channel between network clients, the transmission rounds are loosely synchronized with a predefined time window. Also similarly, the covert channel bandwidth is limited by how fast a receiving client can check the 64 cache sets. Hence, even though the sender must issue more write operations compared to the previous covert channel, these operations are done natively on the CPU, making them much faster than the receiver's network-based operations. As a result, the bandwidth for the sandboxed process covert channel is the same as for the network to network covert channel.

VIII. Network-based Keystroke Attacks

In this section, we present results from an adversarial setting. We measure keystroke timings on an SSH connection from a victim to reconstruct sensitive (typed) data. Our goal here is not to improve upon the existing body of keystroke attack literature, but rather demonstrate our cache measurements are sufficiently accurate to implement practical, adversarial timing attacks.

On a high level, our attack works as follows. The attacker controls a machine that has an RDMA link to an application server as illustrated in Figure 2. The attacker uses remote PRIME+PROBE to detect network activity in the LLC. A user then opens an interactive SSH session to the application server from a different machine. In an interactive SSH session, each keystroke is sent in a separate packet. The attacker is able to recover the inter-packet times from the cache using the ring buffer location and map them to keystrokes. As we will

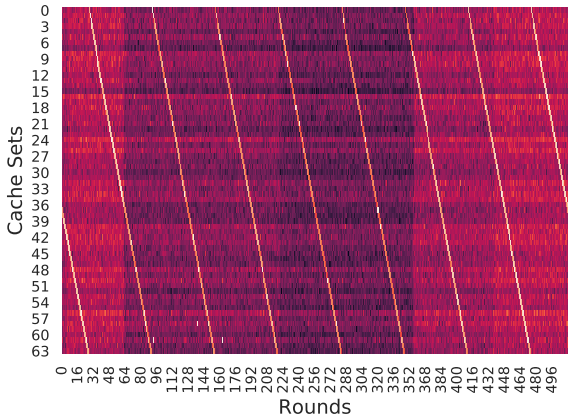

Fig. 7: Memorygram of ring buffer experiment with remote PRIME+PROBE. Darker colors imply faster and lighter colors imply slower access times. In every round we send two network packets. We can see that the ring buffer moves forward each round.

show in this section, such an attack can be launched with a single trace of sensitive data. After launching a remote PRIME+PROBE to measure LLC activity, a successful attack requires the following steps:

1) Locate the network ring buffers of the RX-Queues.
2) Track the RX head to recover incoming packet times.
3) Use machine learning to map times to keystrokes.

A. Locating Ring Buffers in the LLC

A ring buffer is a circular data structure that facilitates processes in reading and writing data asynchronously. In the case of networking, ring buffers are used as a queue between the NIC and the operating system. The ring buffer does not hold packet data directly, but rather pointers to the actual packet data structure (socket kernel buffers). Modern operating systems often have distinct queues (ring buffers) for receiving (RX) and sending (TX) packets. The network ring buffers are often allocated over multiple differently colored pages, which should prevent the ring buffer from self-evicting from the cache. Our experiments show that the ring buffer accesses leave a very distinct pattern in a memorygram. Specifically, two consecutive incoming packets activate the same eviction set, the next two packets then activate the next eviction set, and so on. As a result, for multiple consecutive packets, a staircase-like pattern becomes visible in the memorygram, as can be seen in Figure 7.

To locate the ring buffers in the remote LLC, we first build the remote eviction set as described in Section VI. Next, we launch a PRIME+PROBE variant, where we send two network packets to the server after each prime, and then immediately measure the latency with the probe step. For each of the 256 profiled colors, we execute the PRIME+PROBE 512 times, for a total of 1024 packets per color. After finishing all rounds, we find the distinct staircase pattern in one of the pages. With an RX queue length of 128, the pattern repeats eight times, as shown in Figure 7.

As most modern operating systems have a default network ring buffer size of 512–4096 entries, the staircase pattern still emerges, but covers multiple pages. As the pattern is cyclic, the attacker can reconstruct all possible locations of the ring buffers and predict where to expect the next cache activity.

Modern NICs and operating systems often support multiple RX and TX queues, and use mechanisms such as receive-side scaling (RSS) to distribute packets over different queues on the receiving side, according to a hash over the packet data. Specifically, the hash function is typically a five-tuple input hash over the source IP address, source port, destination IP address, destination port, and the protocol. By changing the source port and protocol, an attacker can map all different queues with the profiling method mentioned above. For simplicity, but without loss of generality, we illustrate the attack on a system that has one RX queue enabled and a ring buffer that resides within one page, i.e., 128 entries.

B. Tracking the Ring Buffer

Once we have determined the page containing the ring buffer, we want to track the exact movements of the ring buffer to leak incoming inter-packet times. One challenge is that when we see an activation of a cache set, we cannot be sure whether this was due to the ring buffer or due to other cache activity. Furthermore, one observed activity of the ring buffer can mean that one or two packets were received, since both subsequent packets activate the same cache set. Lastly, unlike in cooperative attacks, we cannot use multiple measurement rounds because the location of the ring buffer may change between measurements.

To overcome these challenges, we designed a two-stage pipeline to extract inter-packet times. An *online tracker* is in charge of following the Ethernet NIC ring buffer during the measurements, and sends Ethernet probing packets to continuously confirm its position in the cache, determined by sending the RDMA cache PRIME+PROBE packets. The *offline extractor* takes the data produced by the tracker and uses it to compute the likeliest occurrences of client Ethernet network packets (non-probing packets, more specifically, client SSH packets). The following two paragraphs detail how these two algorithms are designed.

Online tracking Repeatedly checking all 64 eviction sets is too slow to measure unsynchronized network packets. Thus, we reduce the number of eviction sets measured at the same time by forming a window w of measurements and shifting w according to the current position of the ring buffer pointer. One of the challenges with this approach is deciding when to shift the window in order to follow the head of the ring buffer. To solve this challenge, we send packets from the attacker machine between measurement rounds. These packets guarantee ring buffer advancement and corresponding cache miss. If the online tracker does not observe this, we know that we must adjust the position of the window w. At a high level, the online tracking algorithm works as follows. First, we have to determine the current position pos of the ring buffer. We do this by sending many network packets in

an initial PRIME+PROBE phase. We stop once we have high confidence that the algorithm detected the correct current ring buffer position.

Next, the online tracker uses its knowledge of pos to prime the eviction sets around pos in a window of size w. In our tests, we chose $w = 10$ for a good trade-off between measuring speed and reliability. We now probe until the algorithm detects a cache activation in the window w, at which point we save the measurements and begin another priming round. We periodically synchronize by sending a packet after priming the cache. After each synchronization, the algorithm sends a network packet to confirm that we register the cache activation as expected. For these experiments, we require a latency threshold to differentiate between packets that cause a cache hit versus the ones that cause a cache miss. We find that we need to maintain this threshold dynamically, as it can slightly drift on the victim machine for unspecified reasons (likely due to power management). In addition to our measurements of ring buffer activity, we save all confirmed and missed synchronization points to aid in the offline analysis phase described next. We provide pseudocode detailing the online tracker's behavior in Appendix C.

Offline extraction The goal of the offline extractor phase is to compute at which time steps the victim machine received packets that were not probes sent by the attacker. To this end, the offline extractor receives cacheline latency measurements, and the state of the online tracker at each measurement point. The online tracker only records a timestep when it estimates that there is a cacheline miss observed anywhere among the measurements, regardless of whether it was caused by a probe or not.

The offline extractor examines the cacheline latency measurements and reconstructs the ring buffer accesses. The extractor can rely on the known times of the probing packets which serve as a baseline score for the extractor. We compute the corresponding ring buffer progression that this arrival pattern produces. We score this guess by summing up all measurement latencies that, according to this progression, should be cache misses. We clamp latencies below the 10th percentile and above the 99th percentile to limit the effect of outliers.

We try to enhance our most basic guess by greedily inserting one extra arrived packet in any timestep, starting at 0. If any of these insertions result in a better scoring guess than the current one, we adopt this new pattern of packet arrivals. If our new packet was inserted in step N, we try to insert another packet starting at N (not 0), and only adopt the new guess if there is an improvement, and we repeat this until we cannot improve the guess further. The output of the extractor is a list of timestamps of possible packets that were sent by other clients.

In the last step, we filter network packets that are most likely SSH packets. This step is done by a heuristic, as we do not have any header packet information to distinguish an SSH packet form other network packets. The idea of the heuristic is that after a keystroke is transmitted, the client will send an *ACK* packet. This heuristic works on an idle network. However, this is also an inherent limitation of network-based attacks. If there is more network traffic, i.e., packets arriving close together, our algorithm is not able to distinguish them from SSH packets.

C. Keystroke Prediction

In the previous section, we described how an attacker could measure the cache activity related to the ring buffer and then extract possible SSH packets. The next step is to predict keystrokes from the extracted inter-packet times. Song et al. [47] pioneered the recovery of keystrokes from interactive SSH sessions. In their work, they showed the feasibility of such an attack when capturing SSH packets on a network tap. For this purpose, they used bigrams and a Hidden Markov Model (HMM) to guess typed passwords. The challenge with password datasets is that it is unethical to collect real passwords from users. This would leave the option to let users type a predetermined set of passwords. However, real passwords typing underlies a unique typing frequency which is hard to approximate when users are not trained to type these passwords frequently and over a longer time period. Furthermore, such a dataset would need hundreds of different passwords for a fair evaluation. Similar to more recent work in the area [23, 48], we decided to use word guessing to show an attacker can successfully perform keystroke prediction from the cache measurements.

In order to facilitate reproducibility and comparability, we used a publicly available dataset [49]. The dataset consists of twenty subjects typing free and transcribed text. We extracted words from the free typing sessions with only lowercase characters. The filtered dataset contained a total of 4,574 unique words, on average 228.7 unique words per subject. We split the dataset in training and test set for each user. For each word that was typed multiple times, we split the dataset in a 2:1 ratio between training and test set. We ensure, that a word trace that is in the test set has at least one other trace of the same word in the training set. Furthermore, we also kept word traces in the training set that only occurred once. On average, the training set consists of 376.25 word traces and the test set of 121 traces per user. As we will show later, it is crucial to evaluate a dataset with a sufficiently large word corpus.

To predict words, we used the k-nearest neighbor's algorithm (k-NN), similar to recent work on microarchitectural attacks [23]. The k-NN algorithm classifies an unseen sample by looking at the k nearest neighbors by using a distance metric. In our experiments we used $k = 15$ and uniform weights. This simple approach is a good match to classify keystroke sequences, as we expect users to type words similarly every time. However, users still have a certain degree of variance in their typing, which makes keystroke timing recovery challenging. Prior keystroke timing attacks [47, 48, 50, 51, 52, 53] have also experimented with more sophisticated methods like HMMs, support-vector machines, and neural networks to map keystrokes to characters, words, or users. We focus on a

TABLE II: SSH packet recovery quality for different intervals based on *tcpdump* (Network) data and cache activity (Cache) data.

Source	0.05s Interval TP	FP	FN	0.01s Interval TP	FP	FN	0.001s Interval TP	FP	FN
tcpdump	1.00	.00	.00	1.00	.00	.00	.93	.00	.06
DDIO	.85	.04	.11	.70	.04	.26	.49	.04	.46

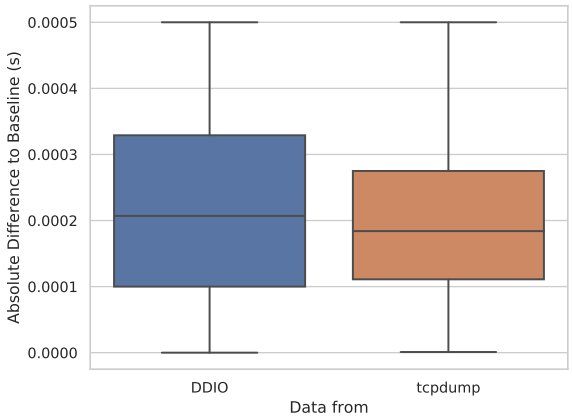

Fig. 8: Absolute difference between the SSH keystrokes emission time and all the correctly predicted SSH packets within $I = 0.001$.

simple k-NN baseline here and expect that, by applying a more sophisticated method, the accuracy of our predictions can further be increased.

D. Evaluation

We evaluated NetCAT on the Intel Xeon Silver 4110 cluster with three machines shown in Figure 2. The attacker can send packets to the NIC to which the victim machine is connected. This allows the attacker to send synchronization packets, as previously described. The victim starts an SSH connection to the application server and then starts typing the words from the test set. We use *expect*, which is a programmable interface to interact with interactive programs, to replay the words in the SSH session by using the key-down-to-key-down times from the dataset [49]. This approach allows us to replicate our experiments with different settings and environmental factors. The online tracker measures the cache activity over 7 seconds. The *expect* program starts to replay the words within such capturing window. Note that the exact start time is not fed into the tracking or extraction algorithms. Besides measuring the cache activity, we also capture the incoming network traffic on the application server with *tcpdump*. While the *tcpdump* baseline assumes a strong attacker model which requires physical access to the server (i.e., a network tap), these traces allow us to make a side-by-side comparison of the classifier on keyboard key-down-to-key-down times, the actual network packet interarrival times (through *tcpdump*), and data recovered from the cache activity.

We have a total of 2,420 test word traces. The total capturing of the training data takes ~6h. This time includes measuring the cache for each word during 7 seconds plus some time to set up *tcpdump*. It is important to note that we only trace the words once and use the resulting data in our classification.

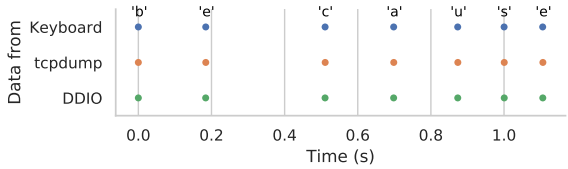

Fig. 9: Successful extraction of the packet times from the cache measurements of the word "because" typed by subject *s033*.

Evaluation of SSH packet recovery We evaluated the work of the online tracker and offline extractor over the entire test set of twenty subjects. We define a predicted packet as True Positive (TP) if the packet is within an interval I of the SSH keystroke packet. A False negative is registered if no predicted packet is within interval I of a SSH keystroke. If the signal extraction predicts more packets than there were emitted, these are counted as False Positive (FP). Similarly, if multiple packets are predicted for one keystroke only, one results in a TP and the rest are FPs. We evaluated the extraction on three different intervals I. Table II presents our results. For $I = 0.05s$, we can extract the SSH keystroke packets with a TP rate of 84.72% (and 11% FN rate). When reducing the interval I, the number of FNs increases. With $I = 0.001s$, the TP rate is still at nearly 50%. Zhang et al. [48] established the ballpark figure of $I = 0.001s$ as sufficient for successful keystroke attacks. For comparison, the table also shows the results of extracting the SSH packets and their times from *tcpdump*. These (ideal) results serve as a baseline for packets delayed over the network and thus no longer within the interval I.

Figure 8 shows the absolute difference between the SSH keystrokes emission time and all the correctly predicted SSH packets within $I = 0.001$. As we can see, the correctly classified packets from the cache have a higher inner quartile range compared to the packets captured with *tcpdump*. In general, this shows that we can extract incoming packet times with only slight time differences compared to the baseline and *tcpdump*. However, the challenge is to correctly extract the packets from the cache measurements in the first place. To give an intuition about a successful SSH packet recovery, we show a trace for the word "because" in Figure 9. In this case, the recovered SSH packets are almost perfectly aligned with the original emission of the keystrokes. Such alignment is possible in low-latency networks. Otherwise, the network and cache data would be shifted according to the transmission time. As we are displaying the data points over a resolution of 1.2 seconds, the small perturbations of the measurements cannot be seen.

End-to-End Evaluation To perform an end-to-end accuracy evaluation, we now take the predicted packets from cache activity and feed them into the k-NN model that was trained on the keyboard training data. We chose this setting as an attacker might have access to a dataset of keystrokes, but

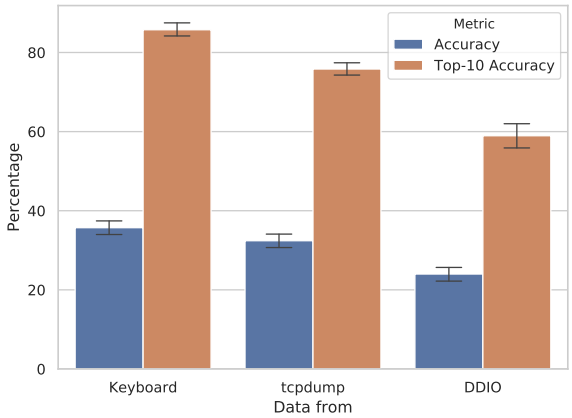

Fig. 10: Classifier accuracy and Top-10 accuracy for the raw keyboard data, the data extracted from *tcpdump*, and the data from the cache measurements over all subjects.

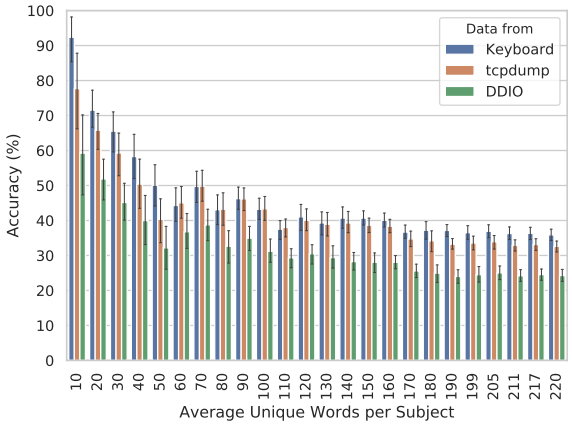

Fig. 11: Variance of the classifier's accuracy when changing the number of unique words for test and training set over all subjects.

cannot replay them on the target network topology. To put the evaluation results of the classifier into perspective, we summarize the accuracy and Top-10 accuracy for the keyboard data, the data extracted from *tcpdump*, and the data from the cache measurements in Figure 10. We can see that, even on the keyboard data, the accuracy of the k-NN model is below 40% which tells us that accurately predicting the true words in this dataset is challenging. Therefore, we used the common Top-10 accuracy metric to show that predicting the right word within a limited number of guesses (10) can be achieved accurately with 85.75%.

When comparing the Top-10 accuracy based on the network data with the raw keyboard data, we can see a significant accuracy drop. Comparing these results with the 93.48% true positive rate on a 0.001s interval in Table II, we can see that even slight perturbations in the interleaving times can mislead the classifier trained on raw keyboard data. This makes predicting the right words more challenging for imperfect SSH packet recovery as in the case of the cache measurements. However, on average over all users, the classifier predicts the right word within its first ten guesses (Top-10 accuracy) in 58.95% of the words. Encouragingly, this is only roughly 15% lower than the performance of *tcpdump* classification. For 50% of the words, the attacker is able to guess the word with 7.85 guesses (median distance). On average over all users and words, the guessing distance is 20.89. On average we have 228.7 words per user. Therefore, a random guesser would have an average distance of 114.35 words. We conclude that the signal of our cache measurement is strong enough to launch a successful keystroke timing attack via remote PRIME+PROBE. The full test scores for each test data source and subject can be found in Appendix D.

To analyze the impact of the word corpus on the classifier, we changed the number of unique words used for training and testing. On every round we choose x words at random from the user specific corpus and then increases x by ten for the next round. The unique words do not necessary increase by ten as we have different word corpus sizes per subject. As shown in Figure 11, with less unique words the classifier has a much higher accuracy than with the total number of unique words in the original dataset [49]. Furthermore, the variance of accuracy is quite significant over a lower number of words and flattens around 170 average unique words per subject. These observations can be made for all the three different testing data sources. One drawback of using a high number of unique words is that our training data set is relatively small, i.e., most of the words in the training set only have one user trace. A dataset with many repetitions per word, a large enough word corpus, and a sufficient number of test subjects would naturally improve the accuracy of predictions.

IX. Generalization

We now discuss how, in future work, NetCAT-like attacks may be generalized beyond our proof-of-concept scenarios.

PCIe to CPU attacks As discussed, DDIO's write allocation limitation prevents an attacker from building eviction sets for the full LLC, making it challenging to directly leak information from the host CPU. To accomplish such an information leak, we believe we can leverage server software running on the victim. For example, the AnC [13] attack can potentially be launched over the network. Given a read primitive with arbitrary offset, such as Redis [54] would give, an attacker might generate an access pattern that would flush a TLB set and translation caches as reverse engineered by [3, 55]. The attacker would then dereference a target virtual address, guaranteed to generate a pagetable walk with virtual address-dependent offsets. If this experiment is repeated often enough with the same offsets (modulo page size), the evictions will eventually reach the DDIO level and the signal will become observable by NetCAT. Similarly, we expect that when certain Spectre [16] gadgets are dereferenced repeatedly with the same offsets (modulo page size), they will cause visible evictions at the DDIO level and allow secrets to be revealed remotely, just as local spectre attacks can.

A further challenge is the resolution time by which we can measure changes in the cache. The eviction set of one cache set in the DDIO context contains two addresses. Therefore continuous probing of one cache set requires two one-sided RDMA reads. The *ib_read_lat* latency benchmark measured an average latency between our Intel Xeon Silver 4110 cluster of 1,550 ns for single a read. In our experiments, we time both read operations together, which results in less overhead than single timed operations. On average we can profile one eviction set with a resolution of 2892 ns on the Intel Xeon Silver 4110 cluster (99th percentile: 3066 ns, SD: 115 ns). The resolution time is bounded by the network round trip time and will differ depending on the setup. Compared to a local cache timing attack, the reduced resolution time in network-based attacks means that for cryptographic key recovery more measurements will likely be necessary. This is an interesting avenue for further research.

Other PCIe Devices as Targets While this paper focuses on snooping on NIC activity through DDIO, in general we can snoop on other PCIe devices. For example, a USB keyboard may send user keystroke events to the LLC via DDIO. This opens the possibility for JavaScript attacks that measure the LLC activity and can obtain sensitive keystroke data or network activity as shown by previous attacks [14, 23]. Unlike previous attacks, the attack through DDIO would be able to monitor cache access patterns and discern hardware-specific behavior, as demonstrated in this paper with the NIC receive buffer access pattern. This potentially allows DDIO-enabled attacks to reach a higher precision.

X. Mitigation

This section discusses potential mitigations against last-level cache side-channel attacks from PCIe devices, such as the attack presented in this paper.

Disabling DDIO The most obvious and straightforward mitigation against DDIO-based attacks such as ours is to disable DDIO. This can be done by adjusting the Integrated I/O (IIO) configuration registers. There are two possibilities, changing it globally (*Disable_All_Allocating_Flows* bit) or per root PCIe port (*NoSnoopOpWrEn* and *Use_Allocating_Flow_Wr* bit). We successfully mitigated NetCAT by setting these bits on our Intel Xeon E5 cluster. For the Intel Xeon Silver 4110 the offsets of these bits are not (yet) publicly documented. While this approach mitigates our attack by preventing us from building a cache eviction set, it comes at a significant performance cost. For example, even for 10 GB/s NICs, disabling DDIO presents a performance bottleneck [19]. Applications which are latency sensitive could suffer an increase of latency by 11% to 18% [26]. Furthermore, power consumption could increase by seven watts per two-port NIC [27].

LLC Partitioning Another possible defense is to use CAT to partition the LLC in hardware or software [56], to limit eviction to a number of ways per set. However, note that this does not solve the problem of inter-device DDIO snooping, as all DDIO-enabled devices still share the same cache ways.

This defense can be implemented in software through page coloring, which allows security domains to be isolated by the kernel by organizing physical memory by color (each color being a partition in the LLC), and ensuring that separate security domains never share a color. Unfortunately, given that domains frequently share devices, this defense might be hard to apply in practice. Software-based LLC partitioning is explored in detail in [57].

Another existing software cache defense is based on TSX [58]. However, this defense does not help against our attack because TSX protects only cache activity generated by the CPU, not devices. Other software defenses [59, 60] similarly fail to address the inter-device snooping possibility. Using CAT can have also negative side effects as it can be abused to accelerate rowhammer attacks [61].

DDIO Improvement The most principled alternative is to change the current design of DDIO. In an ideal design, every user (e.g., network client) would receive their own portion of the cache. Partitioning by ways seems attractive, but is not scalable because of the limited number of ways in the LLC. Ultimately, we believe the optimal solution is a flexible hardware mechanism that allows systems software (such as the OS) to selectively whitelist regions of the LLC for use by DDIO-enabled devices.

XI. Related Work

A. Local Microarchitectural attacks

Local microarchitectural attacks have been widely studied in the context of leaking and corrupting information. These attacks typically either spy on a victim process [1, 2, 3, 4, 5, 6, 7] or co-located VMs [8, 9, 10, 11].

Osvik et al. [1] pioneered the PRIME+PROBE attack on the L1 cache, while Ristenpart et al. [22] developed PRIME+TRIGGER+PROBE to measure L1 and L2 activity on VMs that share a core. Liu et al. [9] extended PRIME+PROBE to the LLC under the assumption of large memory pages, allowing the attacker to extract secrets from co-hosted VMs. Later work extended the threat model to JavaScript [12, 13, 14, 15, 23], allowing attack code to be delivered from a web server.

Our remote PRIME+PROBE is based on the method of Oren et al. [14] to build a non-canonical eviction set. Moreover, our attack requires no attack code execution on the victim machine at all.

B. Network Side-channel & Microarchitectural Attacks

Network-based side-channel attacks typically trigger code execution on a victim machine and then observe the execution time to leak information. For instance, Bernstein [17] recovered an AES key by monitoring request times in a web server encrypting a known-plaintext message. Monitoring was supported by a local machine that was a clone of the victim web server. Cock et al. [62] used an OpenSSL vulnerability to launch a distinguishing attack against Datagram TLS, exploiting the non-constant execution time of the MAC check.

Schwarz et al. [16] remotely exploited a web server containing a Spectre v1 gadget triggered over the network, showing that is is possible to break ASLR over the network.

Kim et al. [63] shows that the Rowhammer flaw can be triggered from software, and later found to be exploitable by increasingly sophisticated means [64, 65, 66, 67], all of which local. Recent work has shown that Rowhammer can also be triggered from the network. Tatar et al. [68] shows how RDMA can be leverage to build an end-to-end Rowhammer exploit in data center settings. Lipp et al. [69] show that under certain cache-limited conditions, Rowhammer can also be triggered in Ethernet networks as well.

Many network-based attacks require repeating operations to filter out noise factors like network variance. In contrast, our attack leaks sensitive information using only a single trace of operation. This is possible because we can measure cache activity precisely by pinpointing the exact cache sets to measure, providing us with more accurate activity measurement than prior work. Moreover, NetCAT can spy even on other PCIe peripherals (not just the CPU), making NetCAT the first network-based attack of its kind.

C. Keystroke attacks

Prior keystroke recovery attacks have targeted *procf* [48], audio [70], CPU scheduling [71], Wi-Fi Signals [72], interrupts [73] and graphic renderings [74]. Song et al. [47] were the first to use SSH network packets to exploit interleaving times for password recovery, using a Hidden Markov Model (HMM) to model character pairs. Hogye et al. [75] argued that network timing variance would disguise such interleaving times in a real-world network. Lipp et al. [23] used JavaScript to spy on URLs typed into browser address bars, using a closed-world dictionary and using k-nearest neighbors to map their signal to URLs. We use the same basic technique to demonstrate the signal strength of our attack. However, we use a publicly available dataset [49] which provides a large set of words and subjects to show that our keystroke attack is practical in real-world settings. As discussed in Section VIII, a large word corpus is key to validating classifier results.

In our prototype setup, we were able to retrieve cacheline information with a polling frequency of between 10kHz and 20kHz. Our offline extraction logic is then reliable enough so that the word prediction accuracy only reduces on average by 11.7% compared to predicting the words from raw keyboard data.

XII. Conclusion

In the last decade, increased peripheral performance has forced Intel to place the LLC on the fast I/O path in its processors. This paper explored the security implications of this design choice and showed that the DDIO feature on modern Intel CPUs exposes the system to cache attacks over the network. Our proof of concept exploit, NetCAT, can leak secret keystrokes of a victim client of a target OpenSSH server through nothing more than timing the duration of network requests. Our implementation of NetCAT required us to reverse engineer the details of the DDIO technology on Intel processors in order to measure the timing differences between packets served from the LLC or memory, respectively. Using only this basic timing primitive, NetCAT is able to build eviction sets and use these as the first stage of a network-based LLC PRIME+PROBE attack, ultimately leading to our keystroke timing attack. While NetCAT is powerful even with only minimal assumptions, we believe that we have merely scratched the surface of possibilities for network-based cache attacks, and we expect similar attacks based on NetCAT in the future. We hope that our efforts caution processor vendors against exposing microarchitectural elements to peripherals without a thorough security design to prevent abuse.

Responsible Disclosure

We initiated a coordinated disclosure process with Intel and NCSC (the Dutch national CERT) on June 23, 2019. The vulnerability was acknowledged by Intel with a bounty and CVE-2019-11184 was assigned to track this issue. The public disclosure was on September 10, 2019.

Acknowledgements

We would like to thank our shepherd, Clémentine Maurice, and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] 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.

[2] 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.

[3] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[4] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: why stopping cache attacks in software is harder than you think," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 937–954.

[5] 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.

[6] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[7] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World," in *IEEE Symposium on Security and Privacy*, 2019.

[8] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion: https://www. vusec. net/projects/VUsion," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 531–545.

[9] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level Cache Side-channel Attacks are Practical," in *IEEE Symposium on Security and Privacy*, 2015.

[10] 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, no. 1-15, 2015.

[11] 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.

[12] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 195–210.

[13] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, vol. 17, 2017, p. 13.

[14] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "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*. ACM, 2015, pp. 1406–1418.

[15] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *arXiv preprint arXiv:1801.01203*, 2018.

[16] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," *arXiv preprint arXiv:1807.10535*, 2018.

[17] D. J. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.

[18] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019.

[19] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 50–59.

[20] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 327–341.

[21] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 476–488.

[22] 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.

[23] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 191–209.

[24] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[25] M. Neve, J.-P. Seifert, and Z. Wang, "Cache time-behavior analysis on aes," *Selected Area of Cryptology*, 2006.

[26] Intel, "Intel Data Direct I/O Technology (Intel DDIO): A Primer," https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012, [Accessed: 24.03.2019].

[27] Intel., "Intel Data Direct I/O Technology Overview," https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2012, [Accessed: 24.03.2019].

[28] Microsoft, "Azure High performance compute VM sizes," https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc, [Accessed: 24.03.2019].

[29] Oracle, "HPC on Oracle Cloud Infrastructure," https://cloud.oracle.com/iaas/hpc, [Accessed: 24.03.2019].

[30] Huawei, "Huawei Cloud Service Combination," https://www.huaweicloud.com/en-us/solution/solution-high-mb/mb1.html, [Accessed: 24.03.2019].

[31] Alibaba, "Alibaba Cloud Super Computing Cluster," https://www.alibabacloud.com/product/scc, [Accessed: 24.03.2019].

[32] Microsoft, "SMB-Direct," https://docs.microsoft.com/en-us/windows-server/storage/file-server/smb-direct, [Accessed: 24.03.2019].

[33] RedHat, "NFS OVER RDMA," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/nfs-rdma, [Accessed: 24.03.2019].

[34] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 14), 2014, pp. 401–414.

[35] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with RDMA for big data processing: Early experiences," in *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*. IEEE, 2014, pp. 9–16.

[36] A. Bhat, N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. K. D. Panda, "A Plugin-Based Approach to Exploit RDMA Benefits for Apache and Enterprise HDFS," in *BPOE*. Springer, 2015, pp. 119–132.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[38] Mellanox, "Accelerate Microsoft SQL Server Performance," http://www.mellanox.com/related-docs/solutions/SB_MFSFT_SQL.PDF, [Accessed: 24.03.2019].

[39] ——, "Mellanox Mitigates Meltdown Mess, Stops Spectre Security Slowdown," http://www.mellanox.com/blog/2018/02/spectre-meltdown-restore-security-performance-patches-mellanox-offload-technologies/, [Accessed: 24.03.2019].

[40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[41] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 48–65.

[42] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance rdma capable interconnects," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[43] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE Symposium on Security and Privacy*, 2019.

[44] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[45] 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.

[46] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." in *NDSS*, vol. 17, 2017, pp. 8–11.

[47] D. X. Song, D. A. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security Symposium*, vol. 2001, 2001.

[48] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security Symposium*, vol. 20, 2009, p. 23.

[49] K. S. Killourhy and R. A. Maxion, "Free vs. Transcribed Text for Keystroke-Dynamics Evaluations," in *Proceedings of the 2012 Workshop on Learning from Authoritative Security Experiment Results*. ACM, 2012, pp. 1–8.

[50] W. Chen and W. Chang, "Applying hidden Markov models to keystroke pattern analysis for password verification," in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.* IEEE, 2004, pp. 467–474.

[51] V. Shanmugapriya and G. Padmavathi, "Keystroke dynamics authentication using neural network approaches," in *International Conference on Advances in Information and Communication Technologies*. Springer, 2010, pp. 686–690.

[52] S. Z. S. Idrus, E. Cherrier, C. Rosenberger, and P. Bours, "Soft Biometrics for Keystroke Dynamics: Profiling Individuals while typing Passwords," *Computers & Security*, vol. 45, pp. 147–155, 2014.

[53] P. Kobojek and K. Saeed, "Application of recurrent neural networks for user verification based on keystroke dynamics," *Journal of telecommunications and information technology*, no. 3, pp. 80–90, 2016.

[54] Redis, "Redis," https://redis.io/, [Accessed: 24.03.2019].

[55] S. Van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, "RevAnC: A framework for reverse engineering hardware page table caches," in *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017, p. 3.

[56] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[57] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 871–882.

[58] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[59] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud com-*

puting security. ACM, 2009, pp. 77–84.

[60] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. H. Campbell, "A Novel Scheduling Framework Leveraging Hardware Cache Partitioning for Cache-Side-Channel Elimination in Clouds," *arXiv preprint arXiv:1708.09538*, 2017.

[61] M. T. Aga, Z. B. Aweke, and T. Austin, "When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017, pp. 8–13.

[62] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The Last Mile: An Empirical Study of Timing Channels on seL4," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 570–581.

[63] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[64] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, 2015.

[65] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 987–1004.

[66] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1–18.

[67] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 1675–1689.

[68] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 213–226.

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

[70] D. Foo Kune and Y. Kim, "Timing attacks on pin input devices," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 678–680.

[71] S. Jana and V. Shmatikov, "Memento: Learning Secrets from Process Footprints," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 143–157.

[72] K. Ali, A. X. Liu, W. Wang, and M. Shahzad, "Keystroke Recognition Using WiFi Signals," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015, pp. 90–102.

[73] W. Diao, X. Liu, Z. Li, and K. Zhang, "No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 414–432.

[74] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries." in *NDSS*, 2019.

[75] M. A. Hogye, C. T. Hughes, J. M. Sarfaty, and J. D. Wolf, "Analysis of the Feasibility of Keystroke Timing Attacks Over SSH Connections," *Research Project at University of Virginia*, 2001.

APPENDIX A
FORWARD SELECTION ALGORITHM

Algorithm 1 Forward Selection

//Let M be a pool of page-aligned addresses
while No Eviction Detected **do**
  //Select n addresses from address pool M
  $S \leftarrow$ selectFromAddressPoolM(n)

  //Write S and do a timed access on all elements of S
  rdmaWrite(S)
  for i in S.size() **do**
    $latencies[i] \leftarrow$ rdmaTimedRead($S[i]$)
  //Take address with highest time as potential x
  $x \leftarrow$ argmax($latencies$)

  //Cache hit time for x
  rdmaWrite(x)
  $t1 \leftarrow$ rdmaTimedRead(x)

  //Potential read from main memory for x
  rdmaWrite($S \setminus x$)
  $t2 \leftarrow$ rdmaTimedRead(x)

  //Determine if x got evicted from $S \setminus x$
  if $t2 - t1 > threshold$ **then**
    $S \leftarrow S \setminus x$
    $break$
  else
    $n \leftarrow n + 1$

APPENDIX B
BACKWARD SELECTION ALGORITHM

Algorithm 2 Backward Selection

//Let S be the set of address from the forward selection which evict address x
for k **do**
  //Select n addresses to be potentially removed from S
  $n \leftarrow$ min(n, S.size()/2)
  $S_{rm} \leftarrow$ selectFromS(n)

  //Cache hit time for x
  rdmaWrite(x)
  $t1 \leftarrow$ rdmaTimedRead(x)

  //Potential read from main memory for x
  rdmaWrite($S \setminus S_{rm}$)
  $t2 \leftarrow$ rdmaTimedRead(x)

  //Determine if x got evicted from $S \setminus S_{rm}$
  if $t2 - t1 > threshold$ **then**
    $S \leftarrow S \setminus S_{rm}$
    $n \leftarrow n + 10$
  else
    $n \leftarrow n - 1$

APPENDIX C
ONLINE TRACKING ALGORITHM

Algorithm 3 Online Tracking

//Let pos be the start position of the ring buffer pointer
while Measurement **do**
  //Eviction sets around pos with window size w
  $es \leftarrow$ getEvictionSets(pos, w)
  Prime(es)
  while True **do**
    //Send network packet if many unsynchronized
    //measurements or if synchronization point failed
    if $unsynced$ > 2 || $send$ == 1 **then**
      SendPacketToServer()
      $injected = 1$
    $latencies \leftarrow$ Probe(es)
    if $latencies[pos] > threshold$ || $injected$ == 1 **then**
      $break$

  if $latencies[pos] > threshold$ || $injected$ == 1 **then**
    //Reached Synchronization state
    $pos \leftarrow$ ExtractNextPos($latencies$)
    $unsynced \leftarrow 0$
    $send \leftarrow 0$
    $syncStatus \leftarrow 1$
  else
    if $injected$ == 1 **then**
      //Missed Synchronization State
      $pos \leftarrow$ RecoverPos($latencies$)
      $send \leftarrow 1$
      $syncStatus \leftarrow 2$
    else
      //Unsynchronized Measurement
      $unsynced \leftarrow unsynced + 1$
      $send \leftarrow 0$
      $syncStatus \leftarrow 0$
  //Export current measurements
  Save($latencies$, $syncStatus$)

APPENDIX D
FULL EVALUATION RESULTS

TABLE III: Full end-to-end evaluations of the word classification for the keyboard data, the network data from *tcpdump* and from the cache measurement.

				Keyboard				tcpdump				DDIO			
Subject	**Traces Training Set**	**Traces Test Set**	**Total Unique Words**	**Accuracy %**	**Top-10 Accuracy %**	**AVG Distance**	**Median Distance**	**Accuracy %**	**Top-10 Accuracy %**	**AVG Distance**	**Median Distance**	**Accuracy %**	**Top-10 Accuracy %**	**AVG Distance**	**Median Distance**
s019	420	146	259	31.51	82.19	6.84	2.0	30.82	76.03	8.45	4.0	21.23	54.79	25.16	9.0
s021	298	80	191	36.25	85.0	5.59	3.0	30.0	76.25	7.67	4.5	26.25	67.5	16.09	7.0
s027	314	103	187	33.98	84.47	6.31	3	33.98	83.5	6.77	4	31.07	69.9	15.32	5
s033	509	166	305	33.73	86.75	6.73	3.0	26.51	71.69	9.83	5.0	21.69	60.24	22.28	7.5
s039	363	110	248	35.45	82.73	5.86	3.5	31.82	72.73	7.72	5.0	25.45	60.0	21.09	8.0
s040	319	86	208	40.7	89.53	4.94	2.5	36.05	77.91	6.84	3.5	24.42	68.6	15.49	5.5
s043	364	119	230	41.18	89.08	4.57	2	36.97	77.31	6.77	4	31.93	65.55	14.24	4
s046	474	171	248	40.35	88.89	5.06	2	36.84	73.68	7.77	4	23.98	54.97	19.94	8
s062	344	103	224	33.98	80.58	6.5	4	30.1	67.96	8.69	6	24.27	55.34	19.7	9
s063	369	116	229	28.45	78.45	6.82	3.0	28.45	76.72	7.63	4.0	16.38	55.17	20.88	9.0
s067	309	96	198	38.54	92.71	4.53	2.0	35.42	78.12	11.27	4.0	26.04	57.29	23.09	7.5
s070	313	89	199	42.7	94.38	3.34	2	40.45	82.02	5.89	4	26.97	65.17	15.89	5
s071	616	227	304	35.24	87.67	6.11	3	29.96	71.37	9.78	6	16.74	42.29	41.04	15
s085	386	142	207	31.69	83.1	5.58	2.0	30.99	78.17	7.51	3.0	22.54	60.56	19.09	6.0
s087	366	117	234	34.19	86.32	6.25	3	30.77	73.5	8.5	5	26.5	64.1	15.86	7
s089	377	112	244	37.5	85.71	5.36	3.0	38.39	74.11	7.65	5.0	25.0	49.11	30.85	11.0
s091	315	87	202	29.89	82.76	6.37	4	25.29	77.01	7.43	5	18.39	62.07	20.01	7
s092	301	87	196	33.33	83.91	5.64	3	28.74	77.01	7.45	5	25.29	60.92	17.79	7
s093	352	108	231	40.74	84.26	5.59	3.0	35.19	77.78	7.07	5.0	21.3	50.0	23.63	10.5
s094	416	155	230	34.84	86.45	5.15	3	31.61	73.55	7.75	5	24.52	55.48	20.41	9
Average	**376.25**	**121.0**	**228.7**	**35.71**	**85.75**	**5.66**	**2.8**	**32.42**	**75.82**	**7.92**	**4.55**	**24.0**	**58.95**	**20.89**	**7.85**