Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR

Daniel Gruss[*] Clémentine Maurice[*] Anders Fogh[†]
Moritz Lipp[*] Stefan Mangard[*]
[*] Graz University of Technology [†] G DATA Advanced Analytics

ABSTRACT

Modern operating systems use hardware support to protect against control-flow hijacking attacks such as code-injection attacks. Typically, write access to executable pages is prevented and kernel mode execution is restricted to kernel code pages only. However, current CPUs provide no protection against code-reuse attacks like ROP. ASLR is used to prevent these attacks by making all addresses unpredictable for an attacker. Hence, the kernel security relies fundamentally on preventing access to address information.

We introduce Prefetch Side-Channel Attacks, a new class of generic attacks exploiting major weaknesses in prefetch instructions. This allows unprivileged attackers to obtain address information and thus compromise the entire system by defeating SMAP, SMEP, and kernel ASLR. Prefetch can fetch inaccessible privileged memory into various caches on Intel x86. It also leaks the translation-level for virtual addresses on both Intel x86 and ARMv8-A. We build three attacks exploiting these properties. Our first attack retrieves an exact image of the full paging hierarchy of a process, defeating both user space and kernel space ASLR. Our second attack resolves virtual to physical addresses to bypass SMAP on 64-bit Linux systems, enabling ret2dir attacks. We demonstrate this from unprivileged user programs on Linux and inside Amazon EC2 virtual machines. Finally, we demonstrate how to defeat kernel ASLR on Windows 10, enabling ROP attacks on kernel and driver binary code. We propose a new form of strong kernel isolation to protect commodity systems incuring an overhead of only 0.06–5.09%.

CCS Concepts

•Security and privacy → Side-channel analysis and countermeasures; Systems security; Operating systems security;

Keywords

ASLR; Kernel Vulnerabilities; Timing Attacks

CCS'16, October 24 - 28, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978356

1. INTRODUCTION

The exploitation of software bugs imperils the security of modern computer systems fundamentally. Especially, buffer overflows can allow an attacker to overwrite data structures that are used in the control flow of the program. These attacks are not limited to user space software but are also possible on operating system kernels [16]. Modern computer hardware provides various features to prevent exploitation of software bugs. To protect against control-flow hijacking attacks, the operating system configures the hardware such that write access to executable pages is prevented. Furthermore, the hardware is configured such that in kernel mode, the instruction pointer may not point into the user space, using a mechanism called supervisor mode execution prevention (SMEP). Data accesses from kernel mode to user space virtual addresses are prevented by operating system and hardware, using a mechanism called supervisor mode access prevention (SMAP). To close remaining attack vectors, address-space layout randomization (ASLR) is used to make all addresses unpredictable for an attacker and thus make return-oriented-programming (ROP) attacks infeasible. All major operating systems employ kernel ASLR (KASLR) [32, 39, 43]. Information on where objects are located in the kernel address space is generally not available to user programs.

Knowledge of virtual address information can be exploited by an attacker to defeat ASLR [17, 46]. Knowledge of physical address information can be exploited to bypass SMEP and SMAP [27], as well as in side-channel attacks [11, 22, 34, 35, 42] and Rowhammer attacks [10, 29, 30, 45]. Thus, the security of user programs and the kernel itself relies fundamentally on preventing access to address information. Address information is often leaked directly through system interfaces such as `procfs` [27] or indirectly through side channels such as double page faults [17]. However, operating system developers close these information leaks through security patches [30]. In this paper, we show that even if the operating system itself does not leak address information, recent Intel and ARM systems leak this information on the microarchitectural level.

We introduce Prefetch Side-Channel Attacks, a new class of generic attacks that allow an unprivileged local attacker to completely bypass access control on address information. This information can be used to compromise the entire physical system by bypassing SMAP and SMEP in ret2dir attacks or defeating KASLR and performing ROP attacks in the kernel address space. Our attacks are based on weaknesses in the hardware design of prefetch instructions. Indeed, prefetch instructions leak timing information on the exact translation level for every virtual address. More severely, they lack a privilege check and thus allow fetching inaccessible privileged memory into various CPU caches. Using these two properties, we build two attack primitives: the translation-level oracle and the address-translation oracle. Building upon these primitives, we then present three different attacks. Our first attack infers the translation level for every virtual address, effectively defeating ASLR. Our second attack resolves virtual addresses to physical addresses on 64-bit Linux systems and on Amazon EC2 PVM instances in less than one minute per gigabyte of system memory. This allows an attacker to perform ret2dir-like attacks. On modern systems, this mapping can only be accessed with root or kernel privileges to prevent attacks that rely on knowledge of physical addresses. Prefetch Side-Channel Attacks thus render existing approaches to KASLR ineffective. Our third attack is a practical KASLR exploit. We provide a proof-of-concept on a Windows 10 system that enables return-oriented programming on Windows drivers in memory. We demonstrate our attacks on recent Intel x86 and ARM Cortex-A CPUs, on Windows and Linux operating systems, and on Amazon EC2 virtual machines.

We present a countermeasure against Prefetch Side-Channel Attacks on commodity systems, that involves reorganizing the user and kernel address space to protect KASLR. Our countermeasure requires only a small number of changes to operating system kernels and comes with a performance impact of 0.06–5.09%.

Our key contributions are:

1. We present two generic attack primitives leveraging the prefetch instructions: the translation-level oracle and the address-translation oracle. We then use these primitives in three different attacks.
2. We present a generic attack to infer the translation level for every virtual address to defeat ASLR.
3. We demonstrate generic unprivileged virtual-to-physical address translation attack in the presence of a physical direct map in kernel or hypervisor, on Linux and in a PVM on Amazon EC2. This allows bypassing SMAP and SMEP, enabling ret2dir attacks.
4. We present a generic attack to circumvent KASLR, which enables ROP attacks inside the kernel. We demonstrate our attack on a Windows 10 system.
5. We propose a new form of strong kernel isolation to mitigate Prefetch Side-Channel Attacks and double page fault attacks on kernel memory.

Outline.

This paper is structured as follows. Section 2 provides background on caches and address spaces. Section 3 presents the settings and two novel attack primitives leveraging the prefetch instructions: the translation-level oracle and the address-translation oracle. The translation-level oracle is used in Section 4 to perform a translation-level recovery attack to defeat ASLR. The address-translation oracle is used in Section 5 to perform unprivileged virtual-to-physical address translation as the basis of ret2dir attacks. Both oracles are used in Section 6 to defeat KASLR. Section 7 shows how to perform cache side-channel and Rowhammer attacks on inaccessible kernel memory. Section 8 presents countermeasures against our attacks. Section 9 discusses related work, and Section 10 concludes this article.

2. BACKGROUND AND RELATED WORK

2.1 Address translation

To isolate processes from each other, CPUs support virtual address spaces. For this purpose, they typically use a multi-level translation table. Which translation table is used is determined by a value stored in a CPU register. This register value is exchanged upon a context switch. Thus, each process has its own address mappings and only access to its own address space. The kernel is typically mapped into every address space but protected via hardware-level access control. When a thread performs a system call it switches to an operating system controlled stack and executes a kernel-level system call handler. However, it still has the same translation table register value.

In the case of recent Intel CPUs, this translation table has 4 levels. On each level, translation table entries define the properties of this virtual memory region, e.g., whether the memory region is present (*i.e.*, mapped to physical memory), or whether it is accessible to user space. The upper-most level is the page map level 4 (PML4). It divides the 48-bit virtual address space into 512 memory regions of each 512 GB (PML4 entries). Each PML4 entry maps to page directory pointer table (PDPT) with 512 entries each controlling a 1 GB memory region that is either 1 GB of physical memory directly mapped (a so-called 1 GB page), or to a page directory (PD). The PD again has 512 entries, each controlling a 2 MB region that is either 2 MB of physical memory directly mapped (a so-called 2 MB page), or a page table (PT). The PT again has 512 entries, each controlling a 4 KB page. The lowest level that is involved in the address translation is called the *translation level*. The CPU has special caches and so-called translation-lookaside buffers for different translation levels, to speed up address translation and privilege checks.

A second, older mechanism that is used on x86 CPUs in virtual-to-physical address translation is segmentation. User processes can be isolated from each other and especially from the kernel by using different code and data segments. Segments can have a physical address offset and a size limit, as well as access control properties. However, these features are widely redundant with the newer translation table mechanism. Thus, most of these features are not available in 64-bit mode on x86 CPUs. In particular, all general purpose segments are required to have the offset set to physical address 0 and the limit to the maximum value. Thus, the CPU can ignore these values at runtime and does not have to perform runtime range checks for memory accesses.

2.2 Virtual address space

The virtual address space of every process is divided into user address space and kernel address space. The user address space is mapped as user-accessible, unlike the kernel space that can only be accessed when the CPU is running in kernel mode.The user address space is divided into memory regions for code, data, heap, shared libraries and stack. Depending on the operating system, the user address space may look entirely different in different processes with respect to the absolute virtual offsets of the regions and also the order of the regions. In contrast, the kernel address space looks mostly identical in all processes.

To perform context switches, the hardware requires mapping parts of the kernel in the virtual address space of every

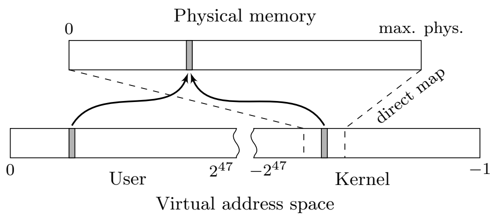

Figure 1: Direct mapping of physical memory. A physical address is mapped multiple times, once accessible for user space and once in the kernel space.

process. When a user thread performs a syscall or handles an interrupt, the hardware simply switches into kernel mode and continues operating in the same address space. The difference is that the privileged bit of the CPU is set and kernel code is executed instead of the user code. Thus, the entire user and kernel address mappings remain generally unchanged while operating in kernel mode. As sandboxed processes also use a regular virtual address space that is primarily organized by the kernel, the kernel address space is also mapped in an inaccessible way in sandboxed processes.

Many operating systems have a physical memory region or the whole physical memory directly mapped somewhere in the kernel space [28, 32]. This mapping is illustrated in Figure 1. It is used to organize paging structures and other data in physical memory. The mapping is located at a fixed and known location, even in the presence of KASLR. Some hypervisors also employ a direct map of physical memory [49]. Thus, every user page is mapped at least twice, once in the user address space and once in the kernel direct map. When performing operations on either of the two virtual addresses, the CPU translates the corresponding address to the same physical address in both cases. The CPU then performs the operation based on the physical address.

Physical direct maps have been exploited in ret2dir attacks [27]. The attacker prepares a code page to be used in the kernel in the user space. Exploiting a kernel vulnerability, code execution in the kernel is then redirected to the same page in the physical direct map. Hence, the attacker has obtained arbitrary code execution in the kernel.

2.3 Address-space layout randomization

Modern CPUs protect against code injection attacks (e.g., NX-bit, $W \oplus X$ policy), code execution in user space memory in privileged mode (e.g., SMEP, supervisor mode execution protection), and data accesses in user space memory regions in privileged mode (e.g., SMAP, supervisor mode access protection). However, by chaining return addresses on the stack it is possible to execute small code gadgets that already exist in the executable memory regions, e.g., return-to-libc and ROP attacks. In an ROP attack, the attacker injects return addresses into the stack and in some cases modifies the stack pointer to a user-controlled region, in order to chain the execution of so-called gadgets. These gadgets are fragments of code already existing in the binary, typically consisting of a few useful instructions and a return instruction.

ASLR is a countermeasure against these control flow hijacking attacks. Every time a process is started, its virtual memory layout is randomized. ASLR can be applied on a coarse-grained level or a fine-grained level. In the case of coarse-grained ASLR, only the base addresses of different memory regions are randomized, e.g., code, data, heap, libraries, stack. This is mostly performed on a page-level granularity. An attacker cannot predict addresses of code and data and thus cannot inject modified code or manipulate data accesses. In particular, an attacker cannot predict the address of gadgets to be used in an ROP attack. All modern operating systems implement coarse-grained ASLR. Fine-grained ASLR randomizes even the order of functions, variables, and constants in memory on a sub-page-level granularity. However, it incurs performance penalties, and can be bypassed [47] and thus is rarely used in practice.

User space ASLR primarily protects against remote attackers that only have restricted access to the system and thus cannot predict addresses for ROP chains. KASLR primarily protects against local attackers as they cannot predict addresses in the kernel space for ROP chains. In particular, invalid accesses cause a crash of the application under attack or the entire system. On Windows, the start offsets of the kernel image, drivers and modules, are randomized.

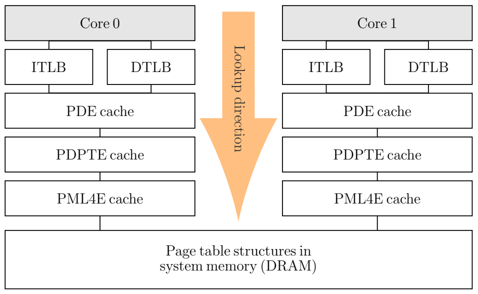

Figure 2: Paging caches are used to speed-up address translation table lookups.

2.4 CPU caches

CPU caches hide slow memory access latencies by buffering frequently used data in smaller and faster internal memory. Modern CPUs employ set-associative caches, where addresses are mapped to cache sets and each cache set consists of multiple equivalent cache lines (also called ways). The index to determine the cache set for an address can be based on the virtual or physical address. The last-level cache is typically physically indexed and shared among all cores. Thus executing code or accessing data on one core has immediate consequences for all other cores.

Address translation structures are stored in memory and thus will also be cached by the regular data caches [21]. In addition to that, address translation table entries are stored in special caches such as the translation-lookaside buffers to allow the CPU to work with them. When accessing virtual addresses these buffers are traversed to find the corresponding physical address for the requested memory area. The caches of the different table lookups are represented in Figure 2. These caches are typically fully-associative.

As CPUs are getting faster, they rely on speculative execution to perform tasks before they are needed. Data prefetching exploits this idea to speculatively load data into the cache. This can be done in two different ways: hardware prefetching, that is done transparently by the CPU itself, and software prefetching, that can be done by a programmer. Recent Intel CPUs have five instructions for software prefetching: `prefetcht0`, `prefetcht1`, `prefetch2`, `prefetchnta`, and `prefetchw`. These instructions are treated like hints to tell the processor that a specific memory location is likely to be accessed soon. The different instructions allow hinting future repeated accesses to the same location or write accesses. Similarly, recent ARMv8-A CPUs supply the prefetch instruction `PRFM`. Both on Intel and ARM CPUs, the processor may ignore prefetch hints.

2.5 Cache attacks

Cache attacks are side-channel attacks exploiting timing differences introduced by CPU caches. Cache attacks have first been studied theoretically [26,31], but practical attacks on cryptographic algorithms followed since 2002 [3,38,48].

In the last ten years, fine-grained cache attacks have been proposed, targeting single cache sets. In an *Evict+Time* attack [37], the attacker measures the average execution time of a victim process, e.g., running an encryption. The attacker then measures how the average execution time changes when evicting one specific cache set before the victim starts its computation. If the average execution time is higher, then this cache set is probably accessed by the victim.

A *Prime+Probe* attack [37,41] consists of two steps. In the Prime step, the attacker occupies one specific cache set. After the victim program has been scheduled, the Probe step is used to determine whether the cache set is still occupied. A new generation of *Prime+Probe* attacks have recently been used to perform attacks across cores and virtual machine borders [22,34,35] as well as from within sandboxes [36].

Gullasch et al. [13] built a significantly more accurate attack that exploits the fact that shared memory, e.g., shared libraries, is loaded into the same cache set for different processes running on the same CPU core. Yarom and Falkner [50] presented an improvement over this attack, called *Flush+Reload* that targets the last-level cache and thus works across cores. *Flush+Reload* attacks work on a single cache line granularity. These attacks exploit shared inclusive last-level caches. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime. Applications of *Flush+Reload* are more reliable and powerful in a wide range of attacks [12,14,23,24,51].

Flush+Reload causes a high number of cache misses due to the frequent cache flushes. This has recently also been used to perform a memory corruption attack called Rowhammer [29]. In a Rowhammer attack, an attacker causes random bit flips in inaccessible and higher privileged memory regions. These random bit flips occur in DRAM memory and the *Flush+Reload* loop is only used to bypass all levels of caches to reach DRAM in a high frequency. Proof-of-concept exploits to gain root privileges and to evade a sandbox have been demonstrated [44]. For the attack to succeed, an attacker must hammer memory locations that map to different rows in the same bank. However, the mapping from addresses to rows and banks is based on physical addresses. Thus, Rowhammer attacks are substantially faster and easier if physical address information is available as an attacker can directly target the comparably small set of addresses that map to different rows in the same bank. As a countermeasure, operating systems have recently restricted access to physical address information to privileged processes [30].

3. SETTING AND ATTACK PRIMITIVES

In this section, we describe the prefetch side channel and two primitives that exploit this side channel. We build a translation-level oracle, that determines whether a page is present and which translation table level is used for the mapping. This primitive is the basis for our translation-level recovery attack described in Section 4 to defeat ASLR. We build an address-translation oracle that allows verifying whether a specific virtual address maps to a specific physical address. We use this to resolve the mapping of arbitrary virtual addresses to physical addresses to mount ret2dir attacks, defeating SMAP and SMEP, in Section 5. We use both attack primitives in our the KASLR exploit described in Section 6.

3.1 Attack setting and attack vector

Attack setting.

In our attacks, we consider a local attack scenario where user space and KASLR are in place. The attacker can run arbitrary code on the system under attack, but does not have access to the kernel or any privileged interfaces such as `/proc/self/pagemap` providing user space address information. This includes settings such as an unprivileged process in a native environment, an unprivileged process in a virtual machine, and a sandboxed process.

To exploit a potential vulnerability in kernel code, an attacker cannot inject code into a writable memory region in the kernel, or directly jump into code located in the user address space as this is prevented by modern CPUs with features like the NX-bit, SMEP, and SMAP. Thus, an attacker can only reuse existing code in a so-called code reuse attack, e.g., ROP attacks. However, building an ROP payload requires exact knowledge of the addresses space layout. Even if the operating system does not leak any address space information and ASLR is employed and effective, we show that the hardware leaks a significant amount of address space information.

The information gained allows an attacker to conduct cache side-channel attacks and Rowhammer attacks, as well as to defeat KASLR and bypass SMAP and SMEP in a ret2dir-like attack.

Attack vector.

Prefetch Side-Channel Attacks are novel and generic side-channel attacks. We exploit the following two properties:

Property 1 The execution time of prefetch instructions varies depending on the state of various CPU internal caches.

Property 2 Prefetch instructions do not perform any privilege checks.

The execution time (Property 1) of a prefetch instruction can be directly measured. It is independent of privilege levels and access permissions. We exploit this property in our translation-level oracle. Intel states that prefetching "addresses that are not mapped to physical pages" can introduce non-deterministic performance penalties [21]. ARM states that the prefetch instructions are guaranteed not to cause any effect that is not equivalent to loading the address directly from the same user process [1]. In both cases, we found timing differences to be deterministic enough to be exploitable. That is, Property 1 can be observed on all our test

Table 1: Experimental setups.

CPU / SoC	Microarchitecture	System type
i5-2530M, i5-2540M	Sandy Bridge	Laptop
i5-3230M	Ivy Bridge	Laptop
i7-4790	Haswell	Desktop
i3-5005U, i5-5200U	Broadwell	Laptop
i7-6700K	Skylake	Desktop
Xeon E5-2650	Sandy Bridge	Amazon EC2 VM
Exynos 7420	ARMv8-A	Smartphone

platforms shown in Table 1, *i.e.*, all Intel microarchitectures since Sandy Bridge as well as the ARMv8-A microarchitecture. Thus, attacks based on Property 1 are applicable to the vast majority of systems used in practice. We demonstrate our translation-level recovery attack on all platforms.

The timing difference caused by the lack of privilege checks (Property 2) can only be measured indirectly using one of the existing cache attack techniques. The combination of the prefetch side channel with different cache attack techniques yields different properties. Intel states that software prefetches should not be used on addresses that are not "managed or owned" by the user process [19], but in practice does not prevent it, thus letting us do this in our attack. Property 2 can be observed on all Intel test platforms shown in Table 1, *i.e.*, all microarchitectures since Sandy Bridge. Thus, attacks based on Property 2 are applicable to the vast majority of desktop, server, and cloud systems.

Measurements.

Measuring the execution time of instructions or memory accesses is typically necessary to perform micro-benchmarks. On ARM CPUs we experienced no difficulties with out-of-order execution. We used `clock_gettime()` to measure time in nanoseconds, as in previous work [33], and surrounded the *target instr.* with a memory and instruction barrier consisting of `DSB SY; ISB`. Depending on the attack we used a memory access or the `PRFM` instruction as *target instruction*.

On Intel CPUs micro-benchmark measurements are significantly harder, due to out-of-order execution. The instructions `rdtsc` and `rdtscp` both provide a sub-nanosecond timestamp. `rdtscp` also waits for all memory load operations to be processed before retrieving the timestamp. The `cpuid` instruction can be used to serialize the instruction stream. To perform accurate measurements, Intel recommends using a sequence of `cpuid; rdtsc` before executing the code to measure and `rdtscp; cpuid` afterward [18]. In cache side-channel attacks memory fences like `mfence` can be used to ensure that memory store operations are also serialized. However, the `prefetch` instruction is not serialized by `rdtscp` or any memory fence, but only by `cpuid` [20]. Due to these serialization issues we crafted instruction sequences to measure exactly a *target instruction* in different scenarios:

1. In cases of long measurements and measurements of memory access times, the target instruction is unlikely to be reordered before a preceding `rdtscp` instruction. We thus use:
   `mfence cpuid rdtscp` *target instr.* `rdtscp cpuid mfence`.
2. When measuring prefetch instructions repeatedly, correct values for minimum and median latency are important. Thus, noise introduced by `cpuid` is tolerable but reordering the target instruction is not, because it could lead to a lower measurement for the minimum latency. In this case we use:
   `mfence rdtscp cpuid` *target instr.* `cpuid rdtscp mfence`.

Depending on the attack we used a memory access, or the prefetch instructions `prefetchnta` and `prefetcht2` as *target instruction*.

3.2 Translation-level oracle

In the translation-level oracle, we exploit differences in the execution time of prefetch instructions (Property 1). Prefetch instructions resolve virtual addresses to physical addresses to enqueue the prefetching request. Intel CPUs follow a defined procedure to find a cache entry or a physical address for a specific virtual address (cf. Section 4.10.3.2 of Intel Manual Vol. 3A [21]):

1. Cache lookup (requires TLB lookup)
2. TLB lookup
3. PDE cache lookup
4. PDPTE cache lookup
5. PML4E cache lookup

The procedure aborts as early as possible omitting all subsequent steps. Step 1 and 2 can be executed in parallel for the L1 cache and thus the latency of step 2 is hidden in this case. However, in case of the L2 or L3 cache, step 2 needs to be executed before to complete the cache lookup in step 1. If no entry is found in any TLB, step 3 needs to be executed to complete step 2 and the same applies for steps 4 and 5. Depending on the specific CPU, some caches may not be present and the corresponding steps are omitted. Every step of the lookup procedure introduces a timing differences that can be measured. For ARM CPUs, the same mechanism applies to the corresponding translation tables. However, on all CPUs tested, we found at least 4 distinct average execution times for different cases.

The translation-level oracle works in two steps. First, we calibrate the execution time of a prefetch instruction on the system under attack. Second, we measure the execution time of a prefetch instruction on an arbitrary virtual address. Based on the execution time, we can now derive on which level a prefetch instruction finished the search for a cache entry or a physical address.

Prefetch instructions on Intel CPUs ignore privilege levels and access permissions (Property 2). Thus, it is possible to prefetch execute-only pages, as well as inaccessible kernel memory. When running the procedure over the whole address space, we now also obtain information on all kernel pages. Note that even a process with root privileges could not obtain this information without loading a kernel module on modern operating systems.

3.3 Address-translation oracle

The lack of privilege checks (Property 2) is the basis for our second oracle, as well as other privilege checks that are not active in 64-bit mode on x86 CPUs (cf. Section 2.1). An attacker can execute prefetch on any virtual address including kernel addresses and non-mapped addresses. Thus, we can use prefetch as an oracle to verify whether two virtual addresses p and $\bar{p}$ map to the same physical address. The address-translation oracle works in three steps:

1. Flush address p
2. Prefetch (inaccessible) address $\bar{p}$
3. Reload p

If the two addresses map to the same physical address, the prefetch of $\bar{p}$ in step 2 leads to a cache hit in step 3 with a high probability. Thus, the access time in step 3 is lower than for a cache miss. By repeating this measurement, the

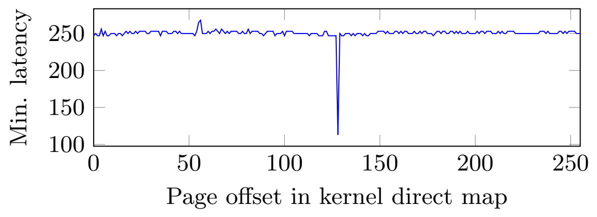

Figure 3: Minimum memory access time for an address p after prefetching different inaccessible addresses, on an i5-3320M. Peak shows the single address $\bar{p}$ mapping to the same physical address as p.

confidence level can be increased to the desired value. One measurement round takes 100–200 nanoseconds on our test systems. Thus, an attacker can run up to 10 million such measurements per second. Figure 3 shows the minimum access time from step 3, over a set of inaccessible addresses $\bar{p}$ measured on an i5-3320M. The peak shows the single address $\bar{p}$ that maps to the same physical address as p.

Similarly, we can also perform a microarchitectural timing attack on prefetch instructions directly. Based on the execution time of prefetch instructions (Property 1), we can measure whether a targeted address p is in the cache. In this *Evict+Prefetch*-variant of the address-translation oracle, we exploit both properties of prefetch instructions (cf. Section 3.1). As the target address might be inaccessible, we evict the address instead of flushing it in the first step. The prefetching replaces the reload step and checks whether the inaccessible address is already in the cache:

1. Evict address p
2. Execute function or system call
3. Prefetch p

If the function or system call in step 2 accesses any address $\bar{p}$ that maps to the same physical address as address p, we will observe a lower timing in step 3 with a high probability. Thus, as in the regular address-translation oracle, we determine whether an address $\bar{p}$ and an address p map to the same physical address. The difference is that in the *Evict+ Prefetch*-variant the address $\bar{p}$ is unknown to the attacker. Instead, the attacker learns that a targeted address p is used by the function or system call.

4. TRANSLATION-LEVEL RECOVERY ATTACK

In this section, we describe how to determine the translation level from an unprivileged user space process based on the translation-level oracle described in Section 3.2. Processes with root privileges can normally obtain system information to derive the translation level, for instance on Linux using the `pagemap` file in `procfs`. However, even here the information provided by the operating system is incomplete and to obtain the translation-level information for kernel memory, it is necessary to install a kernel module for this purpose. We instead only rely on the timing difference observed when running a prefetch attack on an address.

Figure 4 shows the median prefetch execution time for 5 different cases measured on an i5-2540M compared to the actual mapping level. We measured the execution time of `prefetchnta` and `prefetcht2` in 0.5 million tests. The lowest median timing can be observed when the address is valid

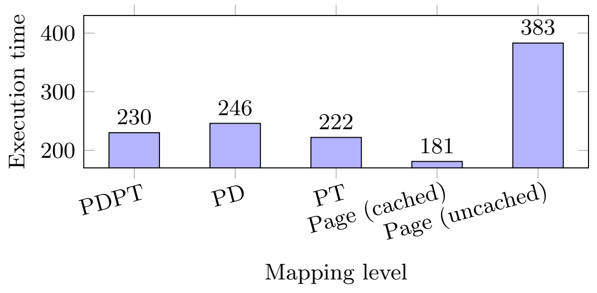

Figure 4: Median prefetch execution time in cycles compared to the actual address mapping level, measured on an i5-2540M.

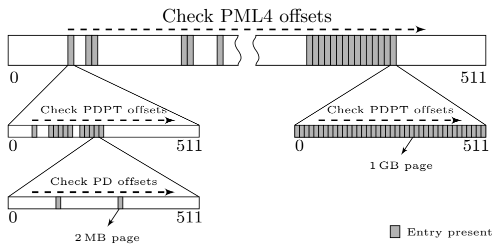

Figure 5: Breadth-first search through the page translation entries that are present. The attacker obtains an accurate map of the page translation level for user and kernel address space.

and cached, with a median of 181 cycles. It is interesting to observe that prefetching a non-cached address when all TLB entries are present has a median execution time of 383 cycles. Thus, we can use prefetch instructions to distinguish cache hits and misses for valid addresses. In case the targeted address is not valid, we observe different execution times depending the mapping level where the address resolution ends. If the memory region has a page directory but the page table is not valid, the median execution time is 222 cycles. If the memory region does not have a page directory but a PDPT, the median execution time is 246 cycles. If the memory region does not have a PDPT, the median execution time is 230 cycles. Note that these timings strongly depend on the measurement techniques in Section 3.1.

We perform a breadth-first search starting with the PML4 and going down to single pages as illustrated in Figure 5. We start the recovery attack with the top-level PML4 recursively going down to the lowest level (cf. Section 2.1). We eliminate measurement noise by checking multiple addresses in each of the 512 regions on each layer. The median execution time of a prefetch instruction sequence is used to decide whether a PDPT is mapped or not. On the PDPT level we thus obtain information on 1 GB pages, on the PD level we obtain information on 2 MB pages and on the lowest level (PT) we obtain information on 4 KB pages. On each level we learn whether the address is mapped directly from this level, or a lower level, or whether it is marked as invalid.

For a single check, we perform 2^8 tests that in total take less than 4ms on the i5-2540M. We check 4 addresses per region and thus require less than 16ms per memory region. Thus, for every translation table that is present, our attack has a runtime of approximately 8 seconds. Programs on

Linux typically use at least 8 translations tables (1 PML4, 2 PDPTs, 2 page directories, 3 page tables). The total runtime to only recover the translation levels for the user space here is approximately 1 minute. However, recovering the translation levels for the kernel can take several minutes if 1 GB pages are used by the kernel, or even several hours if 2 MB pages are used for the physical direct map. If more addresses are mapped in the user space, the execution time can increase to several minutes or hours, depending on the target process.

In either case, our attack successfully recovers the translation level which is normally only accessible for processes with root privileges. Obtaining this information effectively defeats ASLR, as the attacker can now accurately determine which addresses are mapped to physical memory by locating libraries or drivers in inaccessible memory regions. Finally, our attack defeats recently proposed countermeasures [7] that employ execute-only mappings, as prefetch instructions ignore access permissions.

Translation-level recovery from Android apps.

Similarly to 64-bit x86, 64-bit ARMv8-A has a 4-level page translation mechanism. This is for instance the case on our Samsung Galaxy S6 with an Exynos 7420 system-on-chip with a non-rooted stock Android system. On this system, we use the unprivileged `PRFM PLDL1KEEP` instruction to prefetch memory and the unprivileged `DC CIVAC` instruction to flush memory from user space.

The basic translation-level recovery attack on our ARMv8-A CPU is the same as on Intel x86 CPUs. The timing measurement by `clock_gettime` provides a measurement on a nanosecond scale. The timing measurement is significantly faster than on Intel x86 CPUs as there is no `cpuid` instruction consuming a significant amount of cycles. We performed 2^8 tests per address from an Android app. In total this takes less than $50\mu s$ on our ARMv8-A system. Thus, the translation-level recovery attack runs on ARM-based devices successfully.

5. ADDRESS-TRANSLATION ATTACK

In this section, we describe how to mount ret2dir-like attacks without knowledge of physical addresses based on our address-translation oracle. We also build an efficient attack to resolve virtual addresses to physical addresses. This attack exploits the physical direct map in the kernel. Many operating systems and hypervisors use such a mapping to read and write on physical memory [28, 32, 49]. The physical direct map has been exploited by Kemerlis et al. [27] to bypass SMEP in their attack called ret2dir. Similarly, this map can also be used for return stacks in an ROP attack if the attacker has knowledge of the physical address of user-accessible memory. However, our address-translation attack is not restricted to ret2dir-like attacks, it also provides physical address information that is necessary in many side-channel attacks and fault attacks [11,22,29,30,34,42,45].

The attack does not require prior knowledge of the virtual offset of the physical direct map in the kernel. This offset is typically fixed, but it can also be determined by using a translation-level recovery attack. We demonstrate our attack on a native Ubuntu Linux system, and from within an Amazon EC2 instance.

The attacker runs the address-translation oracle on one address p and one address $\bar{p}$ in the virtual memory area of

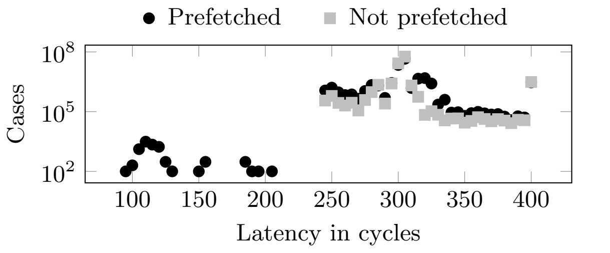

Figure 6: Access latency that has (or has not) been prefetched through a kernel address. Measurements performed on Linux on an Intel i5-3320M.

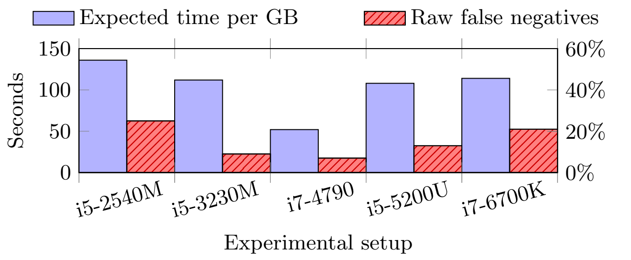

Figure 7: Expected brute-force search time per GB of physical memory, searching for a 2 MB page. Raw false negative rate after a single run of the attack. On all platforms the whole system memory can be searched exhaustively within minutes to hours.

the kernel that is directly mapped to physical memory. The timing difference resulting from prefetching $\bar{p}$ is shown in Figure 6. Note that p and $\bar{p}$ have the same page offset, as the page offset is identical for physical and virtual pages. By only checking the possible offsets based on the known page size, the attacker reduces the number of addresses to check to a minimum. The search space for 2 MB pages is only 512 possibilities per 1 GB of physical memory and for 4 KB pages only 262 144 possibilities per 1 GB of physical memory. The attacker performs a brute-force search for the correct kernel physical direct-map address by trying all possible values for $\bar{p}$. Finding an address $\bar{p}$ means that the attacker has obtained an address that maps user space memory in the kernel address space, thus providing a bypass for SMAP and SMEP. Thus it is possible to mount ret2dir-like attacks using this address. However, the correct physical address can be obtained by subtracting the virtual address of the start of the physical direct map. On Linux, the physical direct map is located at virtual address `0xffff 8800 0000 0000`.

Figure 7 shows the average brute-force search time per gigabyte of physical memory, when searching for a 2 MB page. In this search, we ran our address-translation oracle 2^{14} to 2^{17} times, depending on the architecture. Increasing the number of runs of the address-translation oracle decreases the false negative rate but at the same time increases the execution time. We did not find any false positives in any of the native attack settings. Depending on the architecture, delays were introduced to lower the pressure on the prefetcher. The highest accuracy was achieved on the i5-3230M (Ivy Bridge), the i7-4790 (Haswell), and the i5-5200U (Broadwell) systems where the false negative rate was between 7% and 13%. The accuracy was significantly lower on the i5-2540M (Sandy Bridge) test system. However, the false-negative rate remained at $\geq$ 25% even with a higher number of address-translation oracle runs. For the expected execution time per gigabyte of physical memory, we computed how long the attacks have to be repeated until the physical address is found with a probability of more than 99%. The i5-2540M (Sandy Bridge) test system had the highest false negative rate and thus the expected search time is the highest here. Similarly, on the Skylake system, the attack needs to be executed 3 times to find a physical address with a probability of more than 99%. However, as the execution time per round of the attack on the Skylake system was much lower than on the other systems, the expected execution time is close to the other systems.

Getting host physical addresses on Amazon EC2.

On the Amazon EC2 instance running Linux, we exploit the Xen PVM physical direct map, located at virtual address `0xffff 8300 0000 0000`. Apart from this, the basic attack remains the same. To perform the attack on an Amazon EC2 instance, we compensated the noise by checking multiple 4 KB offsets per 2 MB page. Our machine was scheduled on an Intel Xeon E5-2650 (Sandy Bridge). In a dual CPU configuration, it can manage up to 768 GB of physical memory. To compensate for this huge amount of potentially addressable physical memory, we reduced the number of address-translation oracle runs from 2^{15} to 2^{13}. Our attack speed is thus reduced from 154 seconds to 46 seconds per gigabyte on average, limiting the total attack time to less than 10 hours. While this is significantly more than in a native environment with a smaller amount of physical memory, it is still practical to use this attack to translate a small number of virtual addresses to physical addresses.

As we had no direct access to the real address translation information, we verified our results based on the technique from Section 3.3. Translations are considered correct if multiple consecutive verification loops confirm that the hypervisor physical direct-map addresses indeed allow prefetching the targeted user virtual address, and if the mappings of multiple addresses from the same virtual page can be confirmed as well using the address-translation oracle. We obtained an accuracy of the attack in the cloud that is comparable to the accuracy of the attack in a native environment. The presumably correct physical address is always found, *i.e.*, no false negatives. When searching through the maximum 768 GB of address space, we consistently found 1 false positive match (*i.e.*, a 2 MB page) that was later eliminated in the verification loop.

Other operating systems.

Many other operating systems, such as BSD or OSX, maintain a physical direct map. However, we found no such mapping on Windows. Thus, our address-translation oracle can not directly be applied to Windows systems.

Although 64-bit Android has a physical direct map located at virtual address `0xffff ffc0 0000 0000` and 32-bit Android at virtual address `0xc000 0000`, we were not able to build an address-translation oracle on Android. As the prefetch instructions do not prefetch kernel addresses mapped through the second translation-table base register, the attack is mitigated. However, an attack could be possible on systems where user space and kernel space share a translation-table base register, while the kernel would still be inaccessible. Similarly, the attack does not work on today's Google NaCl sandbox as it uses a 32-bit address space using 32-bit segmentation. The sandboxed process therefore only partially shares an address space with the non-sandboxed code and thus the attack is mitigated. However, we verified that a Prefetch Side-Channel Attack using cache eviction instead of `clflush` within the Google NaCl sandbox works on the lowest 4 GB of virtual memory. Thus, when Google NaCl introduces support for 64-bit address spaces in the NaCl sandbox, 32-bit segmentation cannot be used anymore and our attack is likely to succeed on all virtual addresses and thus to leak physical addresses to sandboxed processes.

6. KERNEL ASLR EXPLOIT

In this section, we demonstrate how to defeat KASLR by using prefetch instructions. We demonstrate our attack on Windows 10 and Windows 7 systems. Similarly as in the previous attack, we try to locate mapped memory regions in address space regions that are not accessible from user space. Again, we exploit the omission of privilege checks by prefetch instructions (Property 2). As described in Section 3, we use prefetch instructions in combination with code execution to identify the load address of drivers in kernel mode in this first stage of the attack. In the second stage of the attack, we determine addresses used by a specific driver. By locating the driver, we effectively defeat KASLR.

Similarly, on Windows 7, kernel and hardware-abstraction layer are located between virtual address `0xffff f800 0000 0000` and `0xffff f87f ffff ffff` and system drivers are located between virtual address `0xffff f880 0000 0000` and `0xffff f88f ffff ffff`. On Windows 10, the address range is extended to the region from `0xffff 8000 0000 0000` to `0xffff 9fff ffff ffff`. Which drivers are present in the driver area depends on the system configuration. Furthermore, the order in virtual address space directly depends on the order the drivers are loaded, which again depends on the system configuration. To exploit a kernel vulnerability and build an ROP chain in the code of a known driver, an attacker has to know the exact address offset of the driver. However, the exact address offsets are randomized and can normally not be retrieved from user processes. Our attack exploits that KASLR does not randomize the offset of drivers on a sub-page level. Kernel and hardware-abstraction layer are loaded on consecutive 2 MB pages with a random 4 KB start offset on Windows 7. Thus, we cannot attack this memory region directly using a translation-level recovery attack. However, an *Evict+Prefetch* attack is possible on any kernel memory region. To build the most efficient attack, we target the driver memory area where we can first perform a translation-level recovery attack and an *Evict+Prefetch* attack afterward. Windows 10 uses 4 KB pages instead, adding entropy to the randomized driver location.

In the first stage of our attack, we locate addresses mapped to physical memory in the driver memory area using our translation-level recovery attack. Figure 8 illustrates the timing difference between valid and invalid addresses in the driver region on Windows 7 on an Intel i3-5005U. As drivers are loaded consecutively in the virtual address space, we found it to be sufficient for our attack to search through the address space in 2 MB steps and measure where pages are mapped to physical memory. On Windows 7, the average

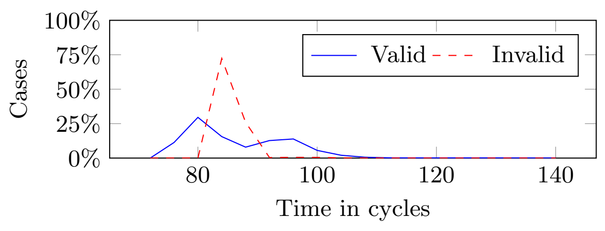

Figure 8: Timing difference of a prefetch sequence on valid and invalid addresses in kernel space, from unprivileged user space process. Measurements performed on Windows 7 on an Intel i3-5005U.

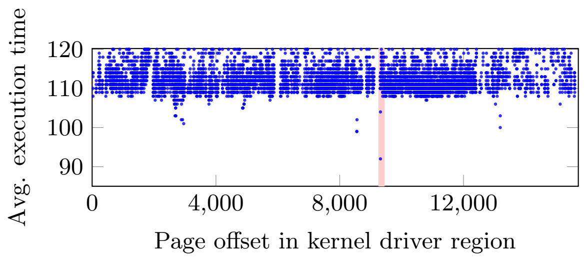

Figure 9: Second stage: driver region is searched by measuring average execution times of prefetching addresses in the driver memory area from the first stage. Lowest average execution time is measured on an address in the memory of the targeted driver.

runtime for the first stage of the attack, mapping both the kernel region and the driver region, is 7 ms on an idle system. On Windows 10, the first stage runs in 64 KB steps and takes 101 ms on average. As Windows 10 maps 4 KB pages we scan the address range in 4 KB steps in an intermediate step taking 180 ms on average. At a high system load, the attack requires several hundred repetitions to perform the first stage of the attack reliably, having an average runtime below 2 seconds on Windows 7.

In the second stage of our attack, we use the *Evict+Prefetch* variant of the address-translation oracle. Instead of searching for pages that are mapped to physical memory, we now determine whether a target address p is used by a syscall. Therefore, we perform the *Evict+Prefetch* attack over all potentially used addresses in a random order. We run the following three steps:

1. *We evict all caches.* For this purpose, we access a buffer large enough to evict all driver addresses from all TLBs, page translation caches, code and data caches.
2. *We perform a syscall to the targeted driver.* If the target address p is used by the targeted driver, the CPU fetches it into the caches while executing the syscall.
3. *We measure the timing of a prefetch instruction sequence.* This reveals whether the target address p was loaded into the cache by the driver in the second step.

In order to verify the measurement, we perform a control run where we omit the system call to the targeted driver. If the execution time of a prefetch instruction sequence on the target address p is higher without the system call, we learn that p is in fact used by the driver and not loaded into the cache by other driver activity on the system. The attack can be repeated multiple times to increase the accuracy.

By determining the lowest virtual address in the driver region that is used by the targeted driver, we learn where the driver starts. As we know the driver version we can now use the virtual addresses from this kernel driver in return-oriented-programming attacks.

The average runtime for the second stage of the attack is 490 seconds on Windows 7. Thus, the total average runtime is below 500 seconds on Windows 7 on our i3-5005U. On Windows 10 we narrowed down the potential addresses in the first stage more than in Windows 7. Thus, the average runtime of the second stage is also lower on Windows 10, requiring only 12 seconds on average to locate a driver.

7. OTHER APPLICATIONS

In this section, we discuss how prefetch instructions can be used in other cache attacks. First, we implemented modified variant of *Flush+Reload* called *Flush+Prefetch*. The measurement accuracy of this cache attack is comparable to *Prime+Probe* while the spatial accuracy is the same as in a *Flush+Reload* attack. We verified the feasibility of this attack by implementing a cross-core covert channel. On a Haswell i7-4790 we achieved a performance of 146 KB/s at an error rate of $< 1\%$. This is in the same order of magnitude as the fastest state-of-the-art cache covert channels [11].

Second, the *Evict+Prefetch* variant of the address-translation oracle can be used to perform a *Flush+Reload*-style attack on privileged kernel addresses. Indeed, we demonstrated such an attack in Section 6 to detect whether specific virtual addresses are used by a driver. However, an attacker could also spy on the usage of known virtual addresses in kernel code and drivers. This would allow monitoring activity on system and specific hardware interfaces.

Third, the *Evict+Prefetch* attack also allows performing Rowhammer attacks on privileged addresses. An attacker could directly target kernel page tables or any other kernel data structure. As the execution time is lower than that of *Evict+Reload*, an attack is likely possible. We verified that bit flips can be induced by this attack on a system running at a refresh rate reduced to 25%. However, we leave examinations on the prevalence of this problem on default configured systems and the study of practical Rowhammer exploits using *Evict+Prefetch* open to future work.

8. COUNTERMEASURES

In this section, we discuss countermeasures against Prefetch Side-Channel Attacks. First, we propose a new form of strong kernel isolation, that effectively prevents all Prefetch Side-Channel Attacks on the kernel address space. Second, we will discuss countermeasures that have been proposed against other side-channel attacks and hardware modifications to mitigate Prefetch Side-Channel Attacks.

Stronger kernel isolation.

Removing the identity mapping would help against our virtual-to-physical address translation attack and completely prevent ret2dir-like attacks, however, it would not protect against our KASLR or translation-level recovery attacks.

We propose *stronger kernel isolation*, a new form of strong kernel isolation, to provide security against a wide range of attacks. Strong kernel isolation ensures that no address is mapped in both user space and kernel space. This mechanism has initially been proposed by Kemerlis et al. [27].

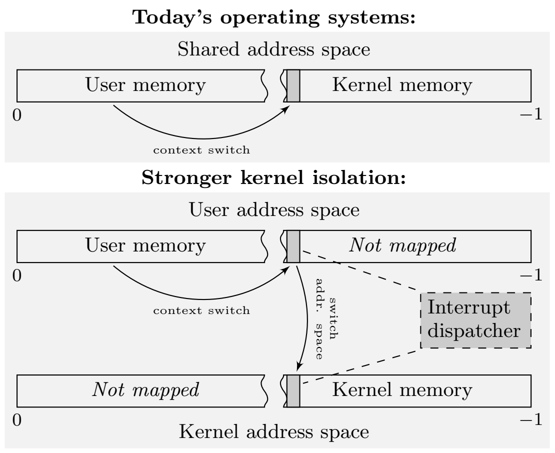

Figure 10: Currently kernel and user memory are only separated through privilege levels. With stronger kernel isolation, the kernel switches from the user space to a dedicated kernel space, directly after a context switch into privileged mode. Thus, only a negligible portion of interrupt dispatcher code is mapped in both address spaces.

Their approach unmaps pages from the kernel physical direct map when they are mapped in user space. This only introduces a performance penalty of 0.18–2.91%. However, this is not sufficient to protect against our attacks. Instead, *stronger kernel isolation* does not run syscalls and unrelated kernel threads in the same address space as user threads. We propose to switch the address translation tables immediately after the context switch into the kernel. Thus, only short and generic interrupt dispatching code would need to be mapped in the same address space used by the user program. The remainder of the kernel and also the direct mapping of physical memory would thus not be mapped in the address translation tables of the user program. This layout is illustrated in Figure 10.

Stronger kernel isolation also eliminates the double page fault side channel [17], as no virtual address in the user program is valid in both user space and kernel space. This countermeasure can be implemented on commodity hardware and existing operating systems and it only requires a few modifications in operating system kernels. The performance impact is comparably small as switching the address translation tables has to be done once per context switch into the kernel and once per context switch from the kernel back to the user space. This is done by replacing the value in the `cr3` register on Intel x86 CPUs once per context switch. We implemented a proof-of-concept to measure the overhead of updating the `cr3` as an estimate for the performance penalty of stronger kernel isolation. Table 2 shows the overhead in different benchmarks. We observe that for benchmarks that perform a small number of syscalls, the performance overhead is negligible, e.g., 0.06%. For other benchmarks the overhead can be higher, e.g., up to 5.09% in the case of pgbench.

State-of-the-art countermeasures.

While there have been recent advances in detecting cache attacks using performance counters [6, 11, 15, 40] it is less clear whether this is also applicable to Prefetch Side-Channel Attacks. Prefetch Side-Channel Attacks can indeed cause an increased number of DTLB misses and thus could be detected using hardware performance counters. We observe approximatively 4 billion DTLB hits/minute while browsing in Firefox, and approximatively 47 billion while running our virtual-to-physical attack. A more thorough evaluation is needed to assess false positives. While there are numerous events related to prefetching that can be monitored with performance counters, to the best of our knowledge, since Nehalem micro-architecture it is not possible anymore to monitor software prefetching but only hardware prefetching [21]. Future work has to show whether performance counters can indeed be used for a reliable detection mechanism. We also note that while it is possible to disable hardware prefetching, it is not possible to disable software prefetching.

Table 2: Estimation of overhead.

Benchmark	Baseline	Stronger kernel isolation	Overhead
apache	37578.83 req./s	37205.16 req./s	+1.00%
pgbench	146.81 trans./s	139.70 trans./s	+5.09%
pybench	1552 ms	1553 ms	+0, 06%
x264	96.20 fps	96.14 fps	+0.06%

Hardware modifications.

Complete protection against Prefetch Side-Channel Attacks could also be achieved through microarchitectural modifications. We think that prefetch instructions need to be modified in two ways to completely close this attack vector. First, if prefetch instructions performed privilege checks just as other memory referencing instructions, prefetching kernel addresses would trigger a segmentation fault and the process would be killed. It would also prevent measuring the translation table levels over the whole address space as the process would be killed after accessing the first invalid address. Second, prefetch instructions leak timing information on the cache state. The timing difference on our ARM-based smartphones was even higher than on our Intel x86 test system. Eliminating this timing difference would only introduce a small performance overhead, as prefetch instruction are not used by most software. This would prevent cache attacks based on prefetch instructions completely.

9. RELATED WORK

Hund et al. [17] demonstrated three timing side channel attacks to obtain address information. The first is a cache attack searching for cache collisions with kernel addresses. The second performs double page faults to measure timing differences for valid and invalid memory regions introduced by the TLB. The third exploits page fault timing differences due to the TLB and address translation caches. The first attack is mitigated on current operating systems by preventing access to physical addresses, and the second and third attacks can be prevented at the operating system level by preventing excessive use of page faults leading to segmentation faults. In contrast, our attack exploits the TLB and address translation caches without triggering any page faults. Furthermore, as our approach leaks the timing more directly through prefetch instructions, it is faster and retrieves information on a finer granularity, *i.e.*, we can obtain the exact virtual-to-physical address translation. Our approach is also more generic as it bypasses the operating system.

Kemerlis et al. [27] presented two methods providing a basis of ret2dir attacks. First, they use the `procfs` interface to obtain physical addresses, now mitigated on current operating systems by preventing access to this interface. Second, they perform a memory spraying attack where they can use any address in the physical direct map for their ret2dir attack. Our attack enables ret2dir-like attacks without knowledge of physical addresses and recovery of physical addresses from unprivileged user space applications, enabling ret2dir attacks. As a countermeasure, they proposed strong kernel isolation, which we extended in this paper.

Barresi et al. [2] focused on a cross-VM scenario to break ASLR in the cloud with CAIN, while our work mostly focuses on a local attack, that can also be performed on a guest VM. However, CAIN attacks assume a cloud environment that enables memory deduplication, which is already known to be nefarious and is not deployed on e.g., Amazon EC2. In contrast, our attacks do not require memory deduplication and have been performed on Amazon EC2.

Bhattacharya et al. [4] showed that hardware prefetching, performed automatically by the CPU, leaks information. In contrast to this work, we exploit software prefetching which can be triggered at any time by an attacker, from user space. The hardware prefetcher has also been used by Fuchs and Lee [9], as a countermeasure against cache side channels.

Concurrent to our work, Jang et al. [25] exploited Intel TSX transaction to defeat KASLR. TSX transactions prevent pagefaults by jumping to an alternative code path. When accessing or executing on kernel address the timing difference until reaching the alternative code path leaks information on the address translation caches. Evtyushkin et al. [8] exploit the branch-target buffer to break KASLR. Finally, Chen et al. [5] proposed dynamic fine-grained ASLR during runtime to defeat KASLR attacks.

10. CONCLUSION

Prefetch Side-Channel Attacks are a new class of generic attacks exploiting fundamental weaknesses in the hardware design of prefetch instructions. These new attacks allow unprivileged local attackers to completely bypass access control on address information and thus to compromise an entire physical system by defeating SMAP, SMEP, and kernel ASLR. Our attacks work in native and virtualized environments alike. We introduced two primitives that build the basis of our attacks. First, the translation-level oracle, exploiting that prefetch leaks timing information on address translation. Second, the address-translation oracle, exploiting that prefetch does not perform any privilege checks and can be used to fetch inaccessible privileged memory into various caches. The translation-level oracle allowed us to defeat ASLR and locate libraries and drivers in inaccessible memory regions. Using the address-translation oracle, we were able to resolve virtual to physical addresses on 64-bit Linux systems and from unprivileged user programs inside an Amazon EC2 virtual machine. This is the basis for ret2dir-like attacks that bypass SMEP and SMAP. Based on both oracles, we demonstrated how to defeat kernel ASLR on Windows 10, providing the basis for ROP attacks on kernel and driver binary code. As a countermeasure against this new class of attacks, we proposed stronger kernel isolation, such that syscalls and unrelated kernel threads do not run in the same address space as user threads. This countermeasure only requires a few modifications in operating system kernels and that the performance penalty is as low as 0.06–5.09%. Therefore, we recommend that it is deployed in all commodity operating systems.

11. ACKNOWLEDGMENTS

We would like to thank Klaus Wagner for help with some experiments and our anonymous reviewers for their valuable comments and suggestions.

Supported by EU Horizon 2020 programme GA No. 644052 (HECTOR) and EU FP7 programme GA No. 610436 (MATTHEW).

12. REFERENCES

[1] ARM Limited. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[2] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: silently breaking ASLR in the cloud. In *WOOT'15*, 2015.

[3] D. J. Bernstein. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2004.

[4] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay. Hardware prefetchers leak : A revisit of SVF for cache-timing attacks. In *45th International Symposium on Microarchitecture Workshops (MICRO'12)*, 2012.

[5] Y. Chen, Z. Wang, D. Whalley, and L. Lu. Remix: On-demand live randomization. In *6th ACM Conference on Data and Application Security and Privacy*, 2016.

[6] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. Cryptology ePrint Archive, Report 2015/1034, 2015.

[7] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P'15*, pages 763–780, 2015.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016 (to appear).

[9] A. Fuchs and R. B. Lee. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*, 2015.

[10] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA'16*, 2016.

[11] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA'16*, 2016.

[12] D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[13] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P'11*, 2011.

[14] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar. A Faster and More Realistic Flush+Reload

Attack on AES. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2015.

[15] N. Herath and A. Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat 2015 Briefings*, 2015.

[16] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

[17] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P'13*, 2013.

[18] Intel. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper, 2010.

[19] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2014.

[20] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. 253665, 2014.

[21] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 253665, 2014.

[22] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P'15*, 2015.

[23] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Know thy neighbor: Crypto library detection in cloud. *Proceedings on Privacy Enhancing Technologies*, 1(1):25–40, 2015.

[24] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Lucky 13 strikes back. In *AsiaCCS'15*, 2015.

[25] Y. Jang, S. Lee, , and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS'16*, 2016 (to appear).

[26] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2/3):141–158, 2000.

[27] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, pages 957–972, 2014.

[28] kernel.org. Virtual memory map with 4 level page tables (x86_64). https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt, May 2009.

[29] 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 *ISCA'14*, 2014.

[30] Kirill A. Shutemov. Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce, Mar. 2015. Retrieved on November 10, 2015.

[31] P. C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *Crypto'96*, pages 104–113, 1996.

[32] J. Levin. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[33] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Last-Level Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[34] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P'15*, 2015.

[35] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA'15*, 2015.

[36] 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 *CCS'15*, 2015.

[37] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA 2006*, 2006.

[38] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. *Cryptology ePrint Archive, Report 2002/169*, 2002.

[39] PaX Team. Address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[40] M. Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS'16*, 2016.

[41] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.

[42] P. Pessl, D. Gruss, C. Maurice, and S. Mangard. Reverse engineering intel DRAM addressing and exploitation. In *USENIX Security Symposium*, 2016.

[43] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows internals*. Pearson Education, 2012.

[44] M. Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, March 2015. Retrieved on June 26, 2015.

[45] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat 2015 Briefings*, 2015.

[46] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS'04*, 2004.

[47] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P'13*, 2013.

[48] Y. Tsunoo, T. Saito, and T. Suzaki. Cryptanalysis of DES implemented on computers with cache. In *CHES'03*, pages 62–76, 2003.

[49] xenbits.xen.org. page.h source code. http://xenbits.xen.org/gitweb/?p=xen.git;a=blob;hb=refs/heads/stable-4.3;f=xen/include/asm-x86/x86_64/page.h, Mar. 2009.

[50] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[51] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS'14*, 2014.