RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches

Stephan van Schaik
Vrije Universiteit Amsterdam
stephan@synkhronix.com

Kaveh Razavi
Vrije Universiteit Amsterdam
kaveh@cs.vu.nl

Ben Gras
Vrije Universiteit Amsterdam
beng@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

Recent hardware-based attacks that compromise systems with Rowhammer or bypass address-space layout randomization rely on how the processor's memory management unit (MMU) interacts with page tables. These attacks often need to reload page tables repeatedly in order to observe changes in the target system's behavior. To speed up the MMU's page table lookups, modern processors make use of multiple levels of caches such as translation lookaside buffers (TLBs), special-purpose page table caches and even general data caches. A successful attack needs to flush these caches reliably before accessing page tables. To flush these caches from an unprivileged process, the attacker needs to create specialized memory access patterns based on the internal architecture and size of these caches as well as how they interact with each other. While information about TLBs and data caches are often reported in processor manuals released by the vendors, there is typically little or no information about the properties of page table caches on different processors. In this paper, we describe RevAnC, an open-source framework for reverse engineering internal architecture, size and the behavior these page table caches by retrofitting a recently proposed `EVICT+TIME` attack on the MMU. RevAnC can automatically reverse engineer page table caches on new architectures while providing a convenient interface for flushing these caches on 23 different microarchitectures that we evaluated from Intel, ARM and AMD.

1. INTRODUCTION

As software is becoming harder to compromise due to the plethora of advanced defenses [1, 7, 11, 18, 19], attacks on hardware are instead becoming an attractive alternative. These attacks range from compromising the system using the Rowhammer vulnerability [4, 24, 25, 26] and using side channels for breaking address space layout randomization (ASLR) [10, 12, 13, 14, 17] leaking cryptographic keys [27, 6] and even tracking mouse movements [21].

Many of these attacks abuse how modern processors interact with memory. At the core of any processor today is a memory management unit (MMU) that simplifies the management of the available physical memory by virtualizing it between multiple processes. The MMU uses a data structure called the page table to perform the translation between virtual memory to physical memory. Page tables are an attractive target for hardware-based attacks. For example, a single bit flip in a page table page caused by Rowhammer could grant an attacker the control over a physical memory location that she does not own, enough to gain root privileges [25, 26]. Further, security defenses such as ASLR and others that bootstrap using ASLR [5, 8, 19, 20] rely on randomizing where code or data is placed in virtual memory. Since this (secret) information is embedded in page tables, the attackers can perform various side-channel attacks on the interaction of the MMU with page tables to leak this information [12].

The virtual to physical memory translation is slow as it requires multiple additional memory accesses to resolve the original virtual address. To improve performance, modern processors make use of multiple levels of caches such as translation lookaside buffers (TLBs), special-purpose page table caches and even general data caches. To mount a successful attack on page tables, attackers often need to repeatedly flush these caches [12, 13, 25, 26] to observe the system's behavior when manipulating page tables. The details of TLBs and data caches are easy to find by examining processor manuals [15, 16]. Information on page table caches such as their size and behavior, however, is often lacking. Without this information, attackers need to resort to trial and error and it becomes difficult to build robust attacks that work across different architectures.

In this paper, we retrofit AnC [12], an existing `EVICT+TIME` side-channel attack on the MMU to build RevAnC, a reverse engineering framework for determining the size and internal architecture of page table caches as well as how they interact with other caches on 23 microarchitectures comprising various generations from Intel, ARM and AMD processors. RevAnC relies on the fact that page table lookups by the MMU are stored in the last level cache (LLC) in order to speed up the next required translation. By flushing parts of the LLC and timing the page table lookup, RevAnC can identify which parts of the LLC store page tables. On top of

EuroSec '17 April 23, 2017, Belgrade, Serbia
© 2017 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: http://dx.doi.org/10.475/123_4

flushing the LLC, RevAnC needs to flush the TLB as well as page table caches. Since the information on the size of TLB and the LLC is available, we can use RevAnC to reverse engineer the properties of the page table caches that are of interest to attackers, like their internal architecure and size. Summarizing, we make the following contributions:

- We describe a novel technique to reverse engineer the undocumented page table caches commonly found in modern processors.
- We evaluate our technique on 23 different microarchitectures from recent Intel, ARM and AMD processors.
- We release RevAnC as open-source software. RevAnC provides a convenient interface to flush page table caches on the microarchitectures that we tested and can automatically detect page table caches on new processors. More information about AnC and RevAnC can be found at: https://www.vusec.net/projects/anc

We provide some necessary background in Section 2. We then discuss our design and implementation on multiple architectures in Section 3 before evaluating it on 23 different processors in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. BACKGROUND AND MOTIVATION

In this section, we discuss the paging memory management scheme and its implementation on most modern processors. Furthermore, we look at how the MMU performs virtual address translation, and the caches that are used to improve the performance of this translation.

2.1 Paging and the MMU

Paging has become integral to modern processor architectures as it simplifies the management of physical memory by virtualizing it: the operating system no longer needs to relocate the entire memory of applications due to a limited address space and it no longer needs to deal with fragmentation of physical memory. Furthermore, the operating system can limit the memory to which a process has access, preventing malicious or malfunctioning code from interfering with other processes.

As a direct consequence, many modern processor architectures use an MMU, a hardware component responsible for the translation of virtual addresses to the corresponding physical addresses. The translations are stored in the page tables—a unidirectional tree of multiple levels, each of which is indexed by part of the virtual address to select the next level page table, or at the leaves, the physical page. Hence, every virtual address uniquely selects a path from the root of the tree to the leaf to find the corresponding physical address.

Figure 1 shows a more concrete example of how the MMU performs virtual address translation on `x86_64`. First, the MMU reads the `CR3` register to find the physical address of the top-level page table. Then, the top nine bits of the virtual address index into this page table to select the page table entry (PTE). This PTE contains a reference to the next-level page table, which the next nine bits of the virtual address index to select the page table entry. By repeating this operation for the next two levels, the MMU can then find the corresponding physical page for `0x644b321f4000` at the lowest-level page table.

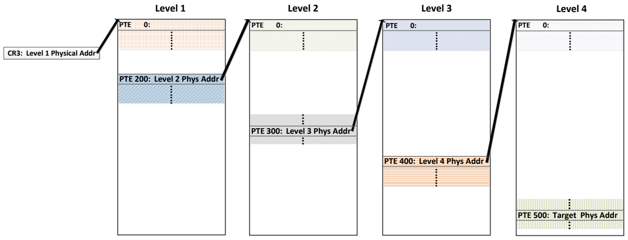
Level 1
Level 2
Level 3
Level 4
CR3: Level 1 Physical Addr
PTE 0:
PTE 200: Level 2 Phys Addr
PTE 0:
PTE 300: Level 3 Phys Addr
PTE 0:
PTE 400: Level 4 Phys Addr
PTE 0:
PTE 500: Target Phys Addr

Figure 1: MMU's page table walk to translate `0x644b321f4000` to its corresponding memory page on the `x86_64` architecture.

2.2 Caching MMU's Operations

The performance of memory accesses improves greatly if the MMU can avoid having to resolve a virtual address that it already resolved recently. For this purpose, CPUs store the resolved address mappings in the TLB cache. Hence, a hit in the TLB removes the need for an expensive page table walk. In addition, to improve the performance of a TLB miss, the processor stores the page table data in the data caches.

Modern processors may improve the performance of a TLB miss even further by means of *page table caches* or *translation caches* that store PTEs for different page table levels [2] [3]. While *page table caches* use the physical address and the PTE index for indexing, *translation caches* use the partially resolved virtual address instead. With *translation caches* the MMU can look up the virtual address and select the page table with the longest matching prefix. Hence, translation caches skip upper levels of the page table and continue the translation with the lowest-level page table present within the cache for the given virtual address. While this allows the MMU to skip part of the page table walk, the implementation of translation caches also comes with additional complexity. Furthermore, these caches can be implemented as split dedicated caches for different page table levels, as one single unified cache for different page table levels, or even as a TLB that is also capable of caching PTEs. We discover using RevAnC that for instance, AMD's Page Walking Caches, as found in the AMD K8 and AMD K10 microarchitectures, and Intel's Page-Structure Caches are examples of *unified page table caches* and *split translation caches*, respectively. Similarly, ARM implements a *unified page table cache* (table walking cache) in the designs that are optimized for low-power consumption and silicon utilization, while they implement a *split translation cache* (intermediate table walking cache) in their designs optimized for high performance.

Figure 2 visualizes how different caches interact with each other when the MMU translates a virtual address. While the TLBs and caches have been documented thoroughly, many of the details on both page table and translation caches remain unspecified.

2.3 Motivation

Recent hardware attacks that abuse page tables need to properly flush page table caches in order to operate correctly. The `prefetch` attack [13], for example, relies on when the virtual address translation partially succeeds in one of the page table caches in order to gain knowledge about a ran-

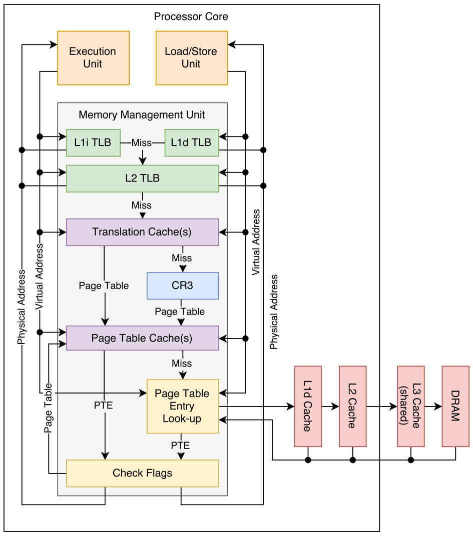

Figure 2: a generic implementation of an MMU and all the components involved to translate a virtual address into a physical address.

domized address in the kernel. Rowhammer attacks that manipulate page tables need to repeatedly flush the TLB and page table caches in order to scan the physical memory for sensitive information.

Another example where flushing page table caches is necessary is the AnC attack [12]. MMU's page table walks are cached in the LLC. AnC makes use of this fact to perform a EVICT+TIME attack [22] to determine the offsets within page table pages that the MMU accessed during a page table walk. The known offsets reveal the randomized virtual addresses, breaking ASLR. AnC needs to try many different access patterns multiples of times for a reliable attack and for each one of them it needs to flush the page table caches in order to trigger a complete page table walk. Hence, the knowledge of the internal page table caches are necessary for a correct and an efficient implementation of AnC.

In some cases, TLBs function as page table caches. In these cases, the cpuid instruction can be used to report the size of the different TLBs, and thus the size of the different page table caches. However, on some x86_64 microarchitectures the cpuid instruction does not report the sizes for all the TLBs. For instance, despite the fact that a TLB is present on Intel Sandy Bridge and Ivy Bridge processors to cache 1 GB pages, this information is not being provided by the cpuid instruction at all. Furthermore, on other CPU architectures there may be no way to report the TLB sizes, or the page table caches may have been implemented as completely independent units. We hence need a more principled approach to finding the important properties of page table caches.

3. RevAnC

Using cpuid on x86_64 and the *flat device tree* (FDT) on ARMv7-A and ARMv8-A, we can detect the processor topology including its properties such as the TLBs, the caches, the names of the processor and the vendor and the microarchitecture. With this information, we can build an appropriate eviction set to flush the TLB and the data caches. We can then use these eviction sets in RevAnC to learn about the page table and translation caches by performing the AnC attack incrementally as we described in this section. We needed to overcome a number of challenges to make RevAnC work across different architectures which we will discuss hereafter.

```
def detect_caches(n, threshold):
    sizes = []

    for level in page_levels:
        for size in [1, 2, 3, 4, 6, 8, 12, 16, 24, ...]:
            count = 0
            evict_set = build_evict_set(sizes + [size])

            for attempt in [1 .. n]:
                timings = profile_page_table(evict_set)
                slot = solve(timings)

                if slot == expected_slot:
                    count += 1

            if count / n > threshold:
                break

        sizes.append(size)

    return sizes
```

Listing 1: high-level overview of the code used to detect the sizes of the page table cache where n is the amount of attempts and threshold is the required accuracy.

3.1 Using the MMU's Cache Signal

We rely on the fact that the MMU's page table walk ends up in the target processor's data caches for learning about page table caches. Assuming an Intel x86_64 with four page table levels as a running example, the MMU's page table walk of a given virtual address v brings four cache lines from the four page table pages into the L1 data cache as well as L3, given that L3 is inclusive of L1. As a result, the next page table walk for v will be considerably faster if the cache lines are still in one of the data caches.

The CPU data caches are partitioned into cache sets. Each cache set can store up to N cache lines which is referred to as N-way set-associative cache. Oren et al. [21] realised that given two different (physical) memory pages, if their first cache lines belong to the same cache set, then the other cache lines in the page share (different) cache sets as well, i.e. if we select an offset t within one page that is aligned on a cache line boundary, then the cache line at offset t in the other page will share the same cache set. This is due to the fact that for the first cache lines to be in the same cache set, all the bits of the physical address of both pages that decide the cache set and slice have to be the same and that an offset within both pages will share the same lower bits.

This property of the caches allows us to simply use many pages as an eviction buffer. Let's assume that one of the four page table entries that translate v is at offset zero of the page table page. If we access the first cache line of all the pages in our eviction buffer, we will evict an MMU's recent page table walk of v from the cache. Hence, the next page table walk of v will be slightly slower since it needs to fetch the aforementioned page table entry from memory. This is an example of an EVICT+TIME attack which RevAnC uses to find up to four cache lines out of the possible 64 cache lines in a memory page that store page table entries. Note that by trying various offsets apart from v, we can distinguish which cache line hosts the page table entry at each level. For example, if we perform the EVICT+TIME on v + 32 KB, the cache line that changes compared to performing EVICT+TIME on v is the cache line that is hosting the level 1 page table. This is because on x86_64, each cache line can store 8 page table entries that map 32 KB of virtual memory.

Assuming a page table cache for one of the page table levels, we will not observe the MMU's activity on that level without flushing the page table cache for that level. As an example, assume that there is a page table cache for level 2 page tables with 32 entries. Given that each entry in the level 2 page table maps 2 MB of virtual memory, if we access a virtually contiguous 64 MB buffer (at every 2 MB boundary), we are going to flush this page table cache. We hence can easily bruteforce the size of the potential page table cache at each level. For example, if we could not observe the signal with RevAnC for the upper three levels of the page tables on an x86_64 Intel, it is due to the page table (translation) cache at the level 2 page table. We then can bruteforce the size of this cache, before moving to the upper level. Listing 1 shows how this is possible. Note that unlike the original AnC attack, in RevAnC we assume a known virtual address, so we know exactly where the MMU signal should appear in the cache. To make Listing 1 robust across architectures we needed to address a number of issues which we discuss next.

3.2 Ensuring Memory Order

Many modern CPU architectures implement out-of-order execution, where instructions are being executed in an order governed by the availability of input data, rather than by their original program order. With out-of-order execution instructions are decoded and stalled in a queue until their input operands are available. Once the input operands are available, the instruction may be issued to the appropriate execution unit and executed by that unit before earlier instructions. In addition such CPU architectures are often superscalar, as they have multiple execution units allowing multiple instructions to be scheduled to these different execution units, allowing for the simultaneous execution of these instructions. After the completion of the instructions their results are written in another queue that is being retired in the order of the original program to maintain a logical order. Furthermore, several modern CPU architectures do not only have the ability to execute instructions out-of-order, but they also have the ability to re-order memory operations.

To measure the execution time of a single instruction on such CPU architectures, we have to inject memory barriers before and after the timing instructions, and code barriers before and after the resulting code to flush the instructions and memory operations that are in-flight. To serialise the memory order, we can use the `dsb` instruction `ARMv7-A` and `ARMv8-A`, whereas on `x86_64` both the `rdtscp` and `mfence` instructions guarantee a serialised memory order. To serialise the instruction order, we can use the `cpuid` instruction on `x86_64`, and the `isb sy` instruction on both `ARMv7-A` and `ARMv8-A`.

3.3 Timing

Since the difference between a cache hit and a cache miss is in the order of hundreds of nanoseconds or even tens of nanoseconds, a highly accurate timing source is required to be able to distinguish a cache hit from a cache miss. While timing information can be obtained through `clock_gettime()` on POSIX-compliant operating systems, the timing information is not accurate enough on various `ARMv7-A` and `ARMv8-A` platforms.

Many modern architectures have dedicated registers to count the amount of processor cycles providing a highly accurate timing source. While such registers are accessible through the unprivileged `rdtsc` of `rdtscp` instructions, the `PMCCNTR` register offered by the performance monitoring unit (PMU) on both `ARMv7-A` and `ARMv8-A` is not accessible by default. Furthermore, when these registers were introduced initially, they were not guaranteed to be synchronised among cores and the processor clock was used directly to increment them. In these cases, process migration and dynamic frequency scaling may influence the timings up to the point that they become unreliable.

Given that most processors today have multiple cores, a thread that simply increments a global variable in a loop can provide a software-based cycle counter. We found this method to work reliably on various platforms with a high precision. Note that the JavaScript version of the original AnC attack deploys a similar technique to build a high precision timer.

3.4 Discussion

While RevAnC can detect split MMU caches, it cannot currently distinguish a unified from a split MMU cache that only caches level 2 entries as these behave the same. Further, RevAnC cannot distinguish page table caches from translation caches. Some architectures use TLBs to implement page table or translation caches. In the case that these TLBs are unified, evicting the TLB also evicts the PTEs. While it is possible to detect this by allocating huge pages for the eviction set, we have left this for future work.

4. EVALUATION

We evaluate RevAnC using 23 different CPUs from Intel, ARM and AMD produced from 2008 to 2017. We report the discovered sizes of page table caches for each page table level (we refer to e.g., level 2 page table as PL2) and finally the time that our technique needs for reverse engineering this information. We have also included the sizes of the different caches and TLBs available for each of the CPU for comparison and completeness.

Our findings are summarized in Table 1. RevAnC uses the information in Table 1 to flush TLBs and page table caches by allocating as many pages as there are cache entries and touching each of these pages for each page level. Listing 2 shows this access pattern. By touching the pages, the MMU will be forced to perform a virtual address translation to replace an existing entry in the cache. Furthermore, by using the page size as the stride for each of the page levels, RevAnC can flush the page structure caches or TLBs in the case of huge pages.

We will now go through interesting points and differences between each vendor discovered by RevAnC.

4.1 Intel

On Intel the last-level cache is inclusive, which means that data available in the last-level cache must also be available in the lower level cache(s). Because of this property, it is sufficient to only evict cache lines from the last-level cache, as

CPU	Year	Caches			TLBs			Detected			Time
		L1	L2	L3	PL1	PL2	PL3	PL2	PL3	PL4	
Intel Xeon E3-1240 v5 (Skylake) @ 3.50GHz	2015	32K	256K	8M	1600	32	20	24	3-4	0	3m08s
Intel Core i7-6700K (Skylake) @ 4.00GHz	2015	32K	256K	8M	1600	32	20	24	3-4	0	3m41s
Intel Celeron N2840 (Silvermont) @ 2.16GHz	2014	24K	1M	N/A	128	16	N/A	12-16	0	0	52s
Intel Core i7-4500U (Haswell) @ 1.80GHz	2013	32K	256K	4M	1088	32	4	24	3-4	0	2m53
Intel Core i7-3632QM (Ivy Bridge) @ 2.20GHz	2012	32K	256K	6M	576	32	4	24-32	3	0	3m05s
Intel Core i7-2620QM (Sandy Bridge) @ 2.00GHz	2011	32K	256K	6M	576	32	4	24	2-4	0	3m11s
Intel Core i5 M480 (Westmere) @ 2.67GHz	2010	32K	256K	3M	576	32	N/A	24-32	2-6	0	2m44s
Intel Core i7 920 (Nehalem) @ 2.67GHz	2008	32K	256K	8M	576	32	N/A	24-32	3	0	4m26s
AMD Ryzen R7 1700 8-Core (Zen) @ 3.3GHz	2017	32K	512K	16M	1600	1600	64	0	64	0	13m16s
AMD FX-8350 8-Core (Piledriver) @ 4.0GHz	2012	64K	2M	8M	1088	1088	1088	0	0	0	2m50s
AMD FX-8320 8-Core (Piledriver) @ 3.5GHz	2012	64K	2M	8M	1088	1088	1088	0	0	0	2m47s
AMD FX-8120 8-Core (Bulldozer) @ 3.4GHz	2011	16K	2M	8M	1056	1056	1056	0	0	0	2m33s
AMD Athlon II 640 X4 (K10) @ 3.0GHz	2010	64K	512K	N/A	560	176	N/A	24	0	0	7m50s
AMD E-350 (Bobcat) @ 1.6GHz	2010	32K	512K	N/A	552	8-12	N/A	8-12	0	0	5m38s
AMD Phenom 9550 4-Core (K10) @ 2.2GHz	2008	64K	512K	2M	560	176	48	24	0	0	6m52s
Rockchip RK3399 (ARM Cortex A72) @ 2.0GHz	2017	32K	1M	N/A	32	512	N/A	16	6	N/A	17m49s
Rockchip RK3399 (ARM Cortex A53) @ 1.4GHz	2017	32K	512K	N/A	10	512	N/A	64	0	N/A	7m06s
Allwinner A64 (ARM Cortex A53) @ 1.2GHz	2016	32K	512K	N/A	10	512	N/A	64	0	N/A	52m26s
Samsung Exynos 5800 (ARM Cortex A15) @ 2.1GHz	2014	32K	2M	N/A	32	512	N/A	16	0	N/A	13m28s
Nvidia Tegra K1 CD580M-A1 (ARM Cortex A15) @ 2.3GHz	2014	32K	2M	N/A	32	512	N/A	16	0	N/A	24m19s
Nvidia Tegra K1 CD570M-A1 (ARM Cortex A15; LPAE) @ 2.1GHz	2014	32K	2M	N/A	32	512	N/A	16	0	N/A	6m35s
Samsung Exynos 5800 (ARM Cortex A7) @ 1.3GHz	2014	32K	512K	N/A	10	256	N/A	64	0	N/A	17m42s
Samsung Exynos 5250 (ARM Cortex A15) @ 1.7GHz	2012	32K	1M	N/A	32	512	N/A	16	0	N/A	6m46s

Table 1: the specifications and results for 23 different microarchitectures.

```c
/* Flush the TLBs and page structure caches. */
for (j = 0, level = fmt->levels; j <= page_level; ++level, ++j)
{
        p = cache->data + cache_line * cache->line_size;

        for (i = 0; i < level->ncache_entries; ++i) {
                *p = 0x5A;
                p += level->page_size;
        }
}
```

Listing 2: flushing the TLBs and page structure caches.

this will cause them to be evicted from the lower-level caches as well. We have found that Intel's Page-Structure Caches or split translation caches are implemented by Intel Core and Xeon processors since at least the Nehalem microarchitecture. On Intel Core and Xeon processors there are caches available for 24-32 PL2 entries and 4-6 PL3 entries, whereas on Silvermont processors there is only a single cache available for 12-16 PL2 entries. Note that during multiple runs of our solution, we converged to different numbers that are close to each other. A conservative attacker can always pick the larger number. On Intel Core and Xeon processors, as well as Silvermont processors, we have found that the sizes of the TLBs as reported by `cpuid` can be used as an appropriate guideline to fully flush these caches, as it is very likely that the TLBs used to cache huge pages also contain the logic to cache intermediate page walks. Finally, we have found that while Sandy Bridge and Ivy Bridge implement a TLB to cache 1G pages, the `cpuid` instruction does not report the presence of this TLB, and that both Nehalem and Westmere implement a PL3 page structure cache without even having such a TLB implemented.

4.2 AMD

On AMD the last-level cache is exclusive, which means that data is guaranteed to be in at most one of the caches allowing more data to be stored at once. To be able to properly evict cache lines, we have to allocate an eviction set of which the size is equal to the total sum of the cache sizes. Our tests have found that the AMD K10 microarchitectures implements AMD's Page Walking Cache with 24 entries. Because AMD Bulldozer and Piledriver implement a 1024-entry L2 TLB that also acts as a Page Walking Cache [9], it is sufficient to only flush the TLB, as indicated by our tests. Similarly, AMD Zen implements a 1536-entry L2 TLB that also acts as a Page Walking Cache. However, another L2 TLB with 64 entries dedicated to 1G pages has been introduced with AMD Zen. Finally, AMD Bobcat seems to implement a page directory cache with 8 to 12 entries.

4.3 ARMv7-A and ARMv8-A

While configurable, on most `ARMv7-A` processors, the L2 cache are configured to be non-inclusive. The virtual address space is divided into two page levels with 256 entries and 4096 entries mapping 4K and 1M per entry respectively. With *Large Physical Address Extensions* (LPAE) this is divided into three page levels with 512, 512 and 4 entries mapping 4K, 2M and 1G per page table respectively. Despite the fact that the last page level only consists of four entries, the RevAnC can still be applied to the other page levels to determine the existence of page table or translation caches. Even though the last-level page table only consists of four entries that fit into a single cache line, we can still apply RevAnC to the other two page levels to determine the existence of page table and translation caches. Furthermore, the low-power variants, such as the ARM Cortex A7, implement a *page table cache*, while the performance-oriented variants, such as the ARM Cortex A15 implement a *translation cache*. We have found that these caches are 64 entries and 16 entries in size respectively. Furthermore, we have found that these sizes can be determined reliably with and without LPAE, and even on ARM big.LITTLE processors with heterogeneous cores, where all cores have been enabled.

`ARMv8-A` processors implement an inclusive last-level cache and the `x86-64` paging model, where Linux often only uses three page levels rather than four to improve the performance of page look-ups. We have found that the low-power variants like the ARM Cortex A53 implement a 64-entry *unified page table cache*, whereas the ARM Cortex A72 extends the 16-entry *split translation cache* with 6 additional PL3-entries.

5. RELATED WORK

AnC [12] launches an `EVICT+TIME` attack on the MMU which relies on how PTEs are cached in the LLC in order to derandomize user-space ASLR from JavaScript. Attacks using the `prefetch` instruction [13] rely on cached TLB entries and partial translations to derandomize kernel-space ASLR natively. Page tables are an attractive target for Rowhammer attacks. Drammer [26] and Seaborn's attacks [25] corrupt a PTE to make it point to a page table page. All these attacks will fail if page table caches are not properly flushed. This paper provides a robust technique for flushing these caches on various architectures.

Reverse engineering of commodity hardware has become popular with increasing attacks on hardware. Hund et al. [14] reverse engineered how Intel processors map physical memory addresses to the LLC. DRAMA [23] used a DRAM timing side channel in order to reverse engineer how memory controllers place data on DRAM modules. In this paper, we reverse engineered various undocumented properties of page table caches common in the MMU of modern processors.

6. CONCLUSIONS

Hardware-based attacks such as cache or Rowhammer attacks are gaining popularity as compromising hardened software is becoming more challenging. For robust attacks across various processors, the properties of various intra-processor caches become important. These caches are often hidden from software and as a result they are sometimes not properly documented. In this paper, we described RevAnC, a framework for reverse engineering the properties of page table caches commonly found on recent processors. We applied our technique on 23 different microarchitectures to detect that 20 of them implement these page table caches. Our open-source implementation provides a convenient interface for flushing these caches on these 20 microarchitectures and can automatically detect page table caches on future microarchitectures. More information about this project be found at: https://www.vusec.net/projects/anc

Acknowledgements

We would like to thank the anonymous reviewers for their feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 SHARCS under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI Dowsing.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. CCS'05.

[2] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). ISCA'10.

[3] A. Bhattacharjee. Large-reach memory management unit caches. MICRO'13.

[4] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16.

[5] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. NDSS.

[6] D. Cock, Q. Ge, T. Murray, and G. Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. CCS'14.

[7] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. NDSS'15.

[8] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. ASIA CCS'15.

[9] A. M. Devices. Software optimization guide for amd family 15h processors, 2014.

[10] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. MICRO'16.

[11] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. SEC'12.

[12] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. NDSS'17.

[13] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. CCS'16.

[14] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. SP'13.

[15] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. Publication No.: 24593, May 2013.

[16] Intel 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032, January 2016.

[17] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. CCS'16.

[18] K. Koning, H. Bos, and C. Giuffrida. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. DSN'16.

[19] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. OSDI'14.

[20] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. CCS'15.

[21] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. CCS'15.

[22] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. CT-RSA'06.

[23] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16.

[24] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16.

[25] M. Seaborn. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat USA*, BH-US'15.

[26] 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. CCS'16.

[27] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. SEC'14.