ASLR on the Line: Practical Cache Attacks on the MMU

Ben Gras* Kaveh Razavi* Erik Bosman Herbert Bos Cristiano Giuffrida
Vrije Universiteit Amsterdam
{beng, kaveh, ejbosman, herbertb, giuffrida}@cs.vu.nl

* Equal contribution joint first authors

***Abstract*—Address space layout randomization (ASLR) is an important first line of defense against memory corruption attacks and a building block for many modern countermeasures. Existing attacks against ASLR rely on software vulnerabilities and/or on repeated (and detectable) memory probing.**

In this paper, we show that neither is a hard requirement and that ASLR is *fundamentally* insecure on modern cache-based architectures, making ASLR and caching conflicting requirements (ASLR⊕Cache, or simply AnC). To support this claim, we describe a new `EVICT+TIME` cache attack on the virtual address translation performed by the memory management unit (MMU) of modern processors. Our AnC attack relies on the property that the MMU's page-table walks result in caching page-table pages in the shared last-level cache (LLC). As a result, an attacker can derandomize virtual addresses of a victim's *code and data* by locating the cache lines that store the page-table entries used for address translation.

Relying only on basic memory accesses allows AnC to be implemented in JavaScript without any specific instructions or software features. We show our JavaScript implementation can break code and heap ASLR in two major browsers running on the latest Linux operating system with 28 bits of entropy in 150 seconds. We further verify that the AnC attack is applicable to every modern architecture that we tried, including Intel, ARM and AMD. Mitigating this attack without naively disabling caches is hard, since it targets the low-level operations of the MMU. We conclude that ASLR is fundamentally flawed in sandboxed environments such as JavaScript and future defenses should not rely on randomized virtual addresses as a building block.

I. Introduction

Address-space layout randomization (ASLR) is the first line of defense against memory-related security vulnerabilities in today's modern software. ASLR selects random locations in the large virtual address space of a protected process for placing code or data. This simple defense mechanism forces attackers to rely on secondary software vulnerabilities (e.g., arbitrary memory reads) to directly leak pointers [16], [57] or ad-hoc mechanisms to bruteforce the randomized locations [5], [6], [17], [19], [23], [47], [55].

Finding secondary information leak vulnerabilities raises the effort on an attacker's side for exploitation [22]. Also bruteforcing, if at all possible [16], [60], requires repeatedly generating anomalous events (e.g., crashes [5], [17], [55], exceptions [19], [23], or huge allocations [47]) that are easy to detect or prevent. For instance, for some attacks [6] disabling non-fundamental memory management features is enough [63]. Consequently, even if ASLR does not stop the more advanced attackers, in the eyes of many, it still serves as a good first line of defense for protecting the users and as a pivotal building block in more advanced defenses [9], [15], [36], [42], [52]. In this paper, we challenge this belief by systematically derandomizing ASLR through a side-channel attack on the memory management unit (MMU) of processors that we call ASLR⊕Cache (or simply AnC).

Previous work has shown that ASLR breaks down in the presence of specific weaknesses and (sometimes arcane) features in software. For instance, attackers may derandomize ASLR if the application is vulnerable to thread spraying [23], if the system turns on memory overcommit and exposes allocation oracles [47], if the application allows for crash tolerant/resistant memory probing [5], [17], [19], [55], or if the underlying operating system uses deduplication to merge data pages crafted by the attacker with pages containing sensitive system data [6]. While all these conditions hold for *some* applications, none of them are universal and they can be mitigated in software.

In this paper, we show that the problem is much more serious and that ASLR is *fundamentally insecure* on modern cache-based architectures. Specifically, we show that it is possible to derandomize ASLR completely from JavaScript, without resorting to esoteric operating system or application features. Unlike all previous approaches, we do not abuse weaknesses in the software (that are relatively easy to fix). Instead, our attack builds on hardware behavior that is central to efficient code execution: the fast translation of virtual to physical addresses in the MMU by means of page tables. As a result, all fixes to our attacks (e.g., naively disabling caching) are likely too costly in performance to be practical. To our knowledge, this is the first attack that side-channels the MMU and also the very first cache attack that targets a victim hardware rather than software component.

High level overview of the attack Whenever a process wants to access a virtual address, be it data or code, the MMU performs a translation to find the corresponding physical address in main memory. The translation lookaside buffer (TLB) in each CPU core stores most of the recent translations in order to speed up the memory access. However, whenever a TLB miss occurs, the MMU needs to walk the page tables (PTs) of the process (also stored in main memory) to perform the trans-

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23271

lation. To improve the performance of the MMU walk (i.e., a TLB miss), the PTs are cached in the fast data caches very much like the process data is cached for faster access [4], [28].

Relying on ASLR as a security mechanism means that the PTs now store security-sensitive secrets: the offset of a PT entry in a PT page at each PT level encodes part of the secret virtual address. To the best of our knowledge, the implications of sharing the CPU data caches between the secret PT pages and untrusted code (e.g., JavaScript code) has never been previously explored.

By executing specially crafted memory access patterns on a commodity Intel processor, we are able to infer which cache sets have been accessed after a targeted MMU PT walk when dereferencing a data pointer or executing a piece of code. As only certain addresses map to a specific cache set, knowing the cache sets allows us to identify the offsets of the target PT entries at each PT level, hence derandomizing ASLR.

Contributions Summarizing, we make the following contributions:

1) We design and implement AnC, the first cache side-channel attack against a hardware component (i.e., the processor's MMU), which allows malicious JavaScript code to derandomize the layout of the browser's address space, solely by accessing memory. Since AnC does not rely on any specific instruction or software feature, it cannot be easily mitigated without naively disabling CPU caches.
2) To implement AnC, we needed to implement better synthetic timers than the one provided by the current browsers. Our timers are practical and can tell the difference between a cached and an uncached memory access. On top of AnC, these timers make previous cache attacks (e.g., [48]) possible.
3) While AnC fundamentally breaks ASLR, we further show, counter-intuitively perhaps, that memory allocation patterns and security countermeasures in current browsers, such as randomizing the location of the browser heaps on every allocation, make AnC attacks more effective.
4) We evaluated end-to-end attacks with AnC on two major browsers running on Linux. AnC runs in tens of seconds and successfully derandomizes code and heap pointers, significantly reducing an attacker's efforts to exploit a given vulnerability.

Outline After presenting the threat model in Section II, we explain the details of address translation in Section III. In that section, we also summarize the main challenges and how we address them. Next, Sections IV—VI discuss our solutions for each of the challenges in detail. In Section VII, we evaluate AnC against Chrome and Firefox, running on the latest Linux operating system. We show that AnC successfully derandomizes ASLR of the heap in both browsers and ASLR of the JIT code in Firefox while being much faster and less demanding in terms of requirements than state-of-the-art derandomization attacks. We discuss the impact of AnC on browser-based exploitation and on security defenses that rely on information hiding in the address space or leakage-resilient code randomization in Section VIII. We then propose mitigations to limit (but not eliminate) the impact of the attack in Section IX and highlight the related work in Section X before concluding in Section XI. Further AnC results are collected at: https://vusec.net/projects/anc.

II. Threat Model

We assume the attacker can execute JavaScript code in the victim's browser, either by luring the victim into visiting a malicious website or by compromising a trusted website. Assuming all the common defenses (e.g., DEP) are enabled in the browser, the attacker aims to escape the JavaScript sandbox via a memory corruption vulnerability. To successfully compromise the JavaScript sandbox, we assume the attacker needs to first break ASLR and derandomize the location of some code and/or data pointers in the address space—a common attack model against modern defenses [54]. For this purpose, we assume the attacker cannot rely on ad-hoc disclosure vulnerabilities [16], [57] or special application/OS behavior [5], [6], [17], [19], [23], [47], [55]. While we focus on a JavaScript sandbox, the same principles apply to other sandboxing environments such as Google's Native Client [66].

III. Background and Approach

In this section, we discuss necessary details of the memory architecture in modern processors. Our description will focus on recent Intel processors due to their prevalence, but other processors use similar designs [4] and are equally vulnerable as we will show in our evaluation in Section VII.

A. Virtual Address Translation

Currently, the virtual address-space of a 64 bit process is 256 TB on x86_64 processors, whereas the physical memory backing it is often much smaller and may range from a few KBs to a few GBs in common settings. To translate a virtual address in the large virtual address-space to its corresponding physical address in the smaller physical address-space, the MMU uses the PT data structure.

The PT is a uni-directed tree where parts of the virtual address select the outgoing edge at each level. Hence, each virtual address uniquely selects a path between the root of the tree to the leaf where the target physical address is stored. As the current x86_64 architecture uses only the lower 48 bits for virtual addressing, the total address space is 256 TB. Since a PT maintains a translation at the granularity of a memory page (4096 bytes), the lower 12 bits of a virtual page and its corresponding physical page are always the same. The other 36 bits select the path in the PT tree. The PT tree has four levels of page tables, where each PT is itself a page that stores 512 PT entries (PTEs). This means that at each PT level, 9 of the aforementioned 36 bits decide the offset of the PTE within the PT page.

Figure 1 shows how the MMU uses the PT for translating an example virtual address, 0x644b321f4000. On the x86 architecture, the CPU's `CR3` control register points to the highest level of the page table hierarchy, known as level 4 or PTL4. The top 9 bits of the virtual address index into this single PTL4 page, in this case selecting PTE 200. This PTE has a reference to the level 3 page (i.e., PTL3), which the next 9 bits of the virtual address index to find the target PT

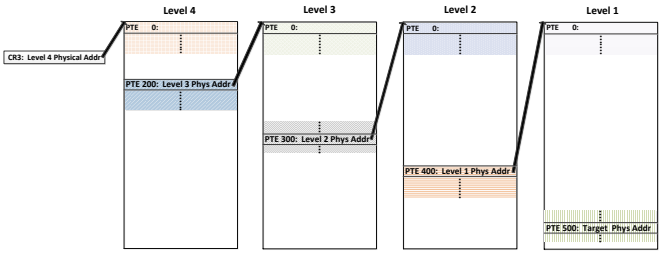

Fig. 1. MMU's page table walk to translate 0x644b321f4000 to its corresponding memory page on the x86_64 architecture.

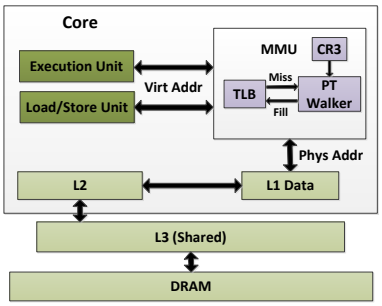

Fig. 2. Memory organization in a recent Intel processor.

entry (this time at offset 300). Repeating the same operation for PT pages at level 2 and 1, the MMU can then find the corresponding physical page for 0x644b321f4000 at the PT entry in the level 1 page.

Note that each PTE will be in a cache line, as shown by different colors and patterns in Figure 1. Each PTE on x86_64 is eight bytes, hence, each 64 byte cache line stores eight PTE. We will discuss how we can use this information for derandomizing ASLR of a given virtual address in Section III-D after looking into the memory organization and cache architecture of recent Intel x86_64 processors.

B. Memory Organization

Recent commodity processors contain a complex memory hierarchy involving multiple levels of caches in order to speed up the processor's access to main memory. Figure 2 shows how the MMU uses this memory hierarchy during virtual to physical address translation in a recent Intel Core microarchitecture. Loads and stores as well as instruction fetches on virtual addresses are issued from the core that is executing a process. The MMU performs the translation from the virtual address to the physical address using the TLB before accessing the data or the instruction since the caches that store the data are tagged with physical addresses (i.e., physically-tagged caches). If the virtual address is in the TLB, the load/store or the instruction fetch can proceed. If the virtual address is not in the TLB, the MMU needs to walk the PT as we discussed in Section III-A and fill in the TLB. The TLB may include translation caches for different PT levels (e.g., paging-structure caches on Intel described in Section 4.10.3 of [29]). As an example, if TLB includes a translation cache for PTL2, then the MMU only needs to walk PTL1 to find the target physical address.

During the PT walk, the MMU reads PT pages at each PT level using their physical addresses. The MMU uses the same path as the core for loading data to load the PTEs during translation. As a result, after a PT walk, the cache lines that store the PTE at each PT level are available in the L1 data cache (i.e., L1D). We now briefly discuss the cache architecture.

C. Cache Architecture

In the Intel Core microarchitecture, there are three levels of CPU caches[1]. The caches that are closer to the CPU are smaller and faster whereas the caches further away are slower but can store a larger amount of data. There are two caches at the first level, L1D and L1I, to cache data and instructions, respectively. The cache at the second level, L2, is a unified cache for both data and instructions. L1 and L2 are private to each core, but all cores share L3. An important property of these caches is their inclusivity. L2 is exclusive of L1, that is, the data present in L1 is not necessarily present in L2. L3, however, is inclusive of L1 and L2, meaning that if data is present in L1 or L2, it also has to be present in L3. We later exploit this property to ensure that a certain memory location is not cached at any level by making sure that it is not present in L3. We now discuss the internal organization of these CPU caches.

To adhere to the principle of locality while avoiding expensive logic circuits, current commodity processors partition the caches at each level. Each partition, often referred to as a *cache set*, can store only a subset of physical memory. Depending on the cache architecture, the physical or virtual address of a memory location decides its cache set. We often associate a cache set with *wayness*. An *n-way* set-associative cache can store n items in each cache set. A replacement policy then decides which of the n items to replace in case of a miss in a cache set. For example, the L2 cache on an Intel Skylake processor is 256 KB and 4-way set-associative with a cache line size of 64 B [28]. This means that there are 1024 cache sets ($256\,\text{KB}/(4\text{-way}\times 64\,\text{B})$) and bits 6 to 16 of a physical address decide its corresponding cache set (the lower 6 bits decide the offset within a cache line).

In the Intel Core microarchitecture, all the cores of the processor share the LLC, but the microarchitecture partitions it in so-called slices, one for each core, where each core has faster access to its own slice than to the others. In contrast to L1 and L2 where the lower bits of a physical address decide its corresponding cache set, there is a complex addressing function (based on an XOR scheme) that decides the slice for each physical memory address [27], [44]. This means that each slice gets different cache sets. For example, a 4-core Skylake i7-6700K processor has an 8 MB 16-way set associative LLC with 4 slices each with 2048 cache sets. We now show how PT pages are cached and how we can evict them from the LLC.

D. Derandomizing ASLR

As discussed earlier, any memory access that incurs a TLB miss requires a PT walk. A PT walk reads four PTEs from main memory and stores them in four different cache lines in

[1]The mobile version of the Skylake processors has a level 4 cache too.

L1D if they are not there already. Knowing the offset of these cache lines within a page already derandomizes six bits out of nine bits of the virtual address at each PT level. The last three bits will still not be known because the offset of the PTE within the cache line is not known. We hence need to answer three questions in order to derandomize ASLR: (1) which cache lines are loaded from memory during the PT walk, (2) which page offsets do these cache lines belong to, and (3) what are the offsets of the target PTEs in these cache lines?

1) Identifying the cache lines that host the PTEs: Since the LLC is inclusive of L1D, if the four PTEs cache lines are in L1D, they will also be in the LLC and if they are not in the LLC, they will also not be in L1D. This is an important property that we exploit for implementing AnC: rather than requiring a timer that can tell the difference between L1D and LLC (assuming no L2 entry), we only require one that can tell the difference between L1D and memory by evicting the target cache line from the LLC rather than from L1D.

The PTE cache lines could land in up to four different cache sets. While we cannot directly identify the cache lines that host the PTE, by monitoring (or controlling) the state of various cache sets at the LLC, we can detect MMU activity due to a PT walk at the affected cache sets. While the knowledge of MMU activity on cache sets is coarser than on cache lines, it is still enough to identify the offset of the PTE cache lines within a page as we describe next.

2) Identifying page offsets of the cache lines: Oren et al. [48] realized that given two different (physical) memory pages, if their first cache lines (i.e., first 64 B) belong to the same cache set, then their other 63 cache lines share (different) cache sets as well. 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 addresses of both pages that decide the cache set and the slice have to be the same and an offset within both memory pages will share the lower 12 bits. Given, for example, 8192 unique cache sets, this means that there are 128 (8192/64) unique page *colors* in terms of the cache sets they cover.

This simple fact has an interesting implication for our attack. Given an identified cache set with PT activity, we can directly determine its page color, and more importantly, the offset of the cache line that hosts the PT entry.

3) Identifying cache line offsets of the PT entries: At this stage, we have identified the cache sets for PTEs at each PT level. To completely derandomize ASLR for a given virtual address, we still need to identify the PTE offset within a cache line (inside the identified cache set), as well as mapping each identified cache set to a PT level.

We achieve both goals via accessing pages that are x bytes apart from our target virtual address v. For example, the pages that are 4 KB, 8 KB, ..., 32 KB away from v, are 1 to 8 PTEs away from v at PTL1 and if we access them, we are ensured to see a change in one of the four cache sets that show MMU activity (i.e., the new cache set will directly follow the previous cache set). The moving cache set, hence, uniquely identifies as the one that is hosting the PT entry for PTL1, and the point when the change in cache set happens uniquely identifies the PT entry offset of v within its cache line, derandomizing the unknown 3 least significant bits in PTL1. We can apply the same principle for finding the PT entry offsets at other PT levels. We call this technique *sliding* and discuss it further in Section V-E.

E. ASLR on Modern Systems

Mapped virtual areas for position-independent executables in modern Linux systems exhibit 28 bit of ASLR entropy. This means that the PTL1, PTL2 and PTL3 fully contribute to creating 27 bits of entropy, but only the last bit of the PTE offset in PTL4 contributes to the ASLR entropy. Nevertheless, if we want to identify this last bit, since it falls into the lowest three bits of the PTE offset (i.e., within a cache line), we require a crossing cache set at PTL4. Each PTE at PTL4 maps 512 GB of virtual address-space, and hence, we need a virtual mapping that crosses a 4 TB mark in order for a cache set change to happen at PTL4. Note that a cache set change in PTL4 results in cache sets changing in the other levels as well. We will describe how we can achieve this by exploiting the behavior of memory allocators in various browsers in Section VI.

Note that the entropy of ASLR in Linux is higher than other popular operating systems such as Windows 10 [45], [64] which provides only 24 bits of entropy for the heap and 17–19 bits of entropy for executables. This means that on Windows 10, PTL4 does not contribute to ASLR for the heap area. Since each entry in PTL3 covers 1 GB of memory, a mapping that crosses an 8 GB will result in cache set change at PTL3, resulting in derandomization of ASLR. The lower executable entropy means that it is possible to derandomize executable locations on Windows 10 when crossing only the two lower level (i.e., with 16 MB). In this paper we focus on the much harder case of Linux which provides the highest entropy for ASLR.

F. Summary of Challenges and Approach

We have discussed the memory hierarchy on modern x86_64 processors and the way in which an attacker can monitor MMU activity to deplete ASLR's entropy. The remainder of this paper revolves around the three main challenges that we need to overcome to implement a successful attack:

- C1 Distinguishing between a memory access and a cache access when performed by the MMU in modern browsers. To combat timing attacks from a sandboxed JavaScript code [6], [48], browsers have decreased the precision of the accessible timers in order to make it difficult, if not impossible, for the attackers to observe the time it takes to perform a certain action.
- C2 Observing the effect of MMU's PT walk on the state of the data cache. Recent work [48] shows that it is possible to build cache eviction sets from JavaScript in order to bring the last-level cache (LLC) to a known state for a well-known `PRIME+PROBE` attack [39], [49]. In a typical `PRIME+PROBE` attack, the victim is a process running on a core, whereas in our attack the victim is the MMU with a different behavior.
- C3 Distinguishing between multiple PT entries that are stored in the same cache line and uniquely identifying PT entries that belong to different PT levels. On e.g., x86_64 each PT entry is 8 bytes, hence, each 64-byte

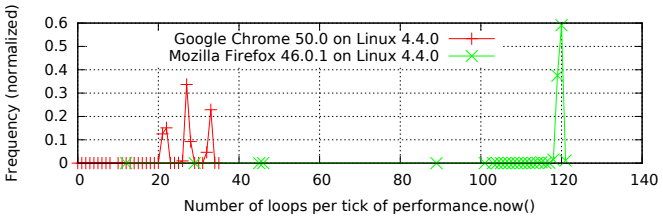

Fig. 3. Measuring the quality of the low-precision `performance.now()` in Chrome and Firefox.

cache line can store 8 PT entries (i.e., the 3 lower bits of the PT entry's offset is not known). Therefore, to uniquely identify the location of a target PT entry within a PT page, we require the ability to access the virtual addresses that correspond to the neighboring PT entries in order to observe a cache line change. Further, in order to derandomize ASLR, we need PT entries to cross cache line at each PT level.

To address C1, we have created a new synthetic timer in JavaScript to detect the difference between a memory and a cache access. We exploit the fact that the available timer, although coarse, is precise and allows us to use the CPU core as a counter to measure how long each operation takes. We elaborate on our design and its implications on browser-based timing attacks in Section IV.

To address C2, we built a `PRIME+PROBE` attack for observing the MMU's modification on the state of LLC in commodity Intel processors. We noticed that the noisy nature of `PRIME+PROBE` that monitors the entire LLC in each round of the attack makes it difficult to observe the (faint) MMU signal, but a more directed and low-noise `EVICT+TIME` attack that monitors one cache set at a time can reliably detect the MMU signal. We discuss the details of this attack for derandomizing JavaScript's heap and code ASLR in Section V.

To address C3, we needed to ensure that we can allocate and access virtually contiguous buffers that span enough PT levels to completely derandomize ASLR. For example, on a 64 bit Linux system ASLR entropy for the browser's heap and the JITed code is 28 bits and on an x86_64 processor, there are 4 PT levels, each providing 9 bits of entropy (each PT level stores 512 PT entries). Hence, we need a virtually contiguous area that spans all four PT levels for complete derandomization of ASLR. In Section VI, we discuss how ASLR⊕Cache exploits low-level memory management properties of Chrome and Firefox to gain access to such areas.

IV. Timing by Counting

Recent work shows that timing side channels can be exploited in the browser to leak sensitive information such as randomized pointers [6] or mouse movement [48]. These attacks rely on the precise JavaScript timer in order to tell the difference between an access that is satisfied through a cache or main memory. In order to thwart these attacks, major browser vendors have reduced the precision of the timer. Based on our measurements, both Firefox and Chrome have decreased the precision of `performance.now()` to exactly 5μs.

We designed a small microbenchmark in order to better understand the quality of the JavaScript timer (i.e., `performance.now()`) in our target browsers. The microbenchmark measures how many times we can execute `performance.now()` in a tight loop between two subsequent ticks of `performance.now()` for a hundred times. Figure 3 shows the results of our experiment in terms of frequency. Firefox shows a single peak, while Chrome shows multiple peaks. This means that Firefox's `performance.now()` ticks exactly at 5μs, while Chrome has introduced some jitter around around the 5μs intervals. The decreased precision makes it difficult to tell the difference between a cached or memory access (in the order of tens of nanoseconds) which we require for AnC to work.

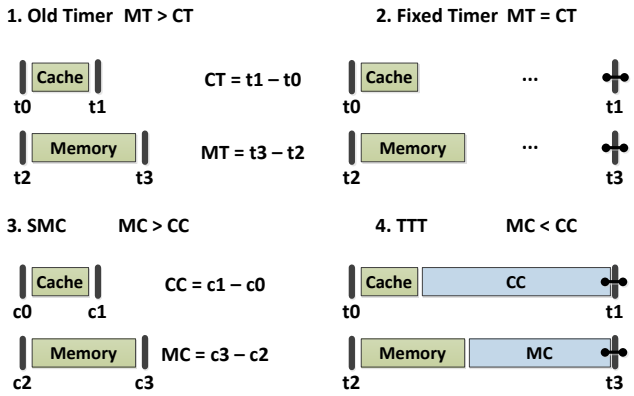

Fig. 4. 1. How the old `performance.now()` was used to distinguish between a cached or a memory access. 2. How the new `performance.now()` stops timing side-channel attacks. 3. How the SMC can be used to make the distinction in the memory reference using a separate counting core as a reference. 4. How TTT can make the distinction by counting in between ticks.

Figure 4.1 shows how the old timer was being used to distinguish between cached or memory access (CT stands for cache time and MT stands for memory time). With a low-precision timer, shown in Figure 4.2, it is no longer possible to tell the difference by simply calling the timer. In the following sections, we describe two techniques for measuring how long executing a memory reference takes by counting how long a memory reference takes rather than timing. Both techniques rely on the fact that CPU cores have a higher precision than `performance.now()`.

The first technique (Figure 4.3), shared memory counter (SMC), relies on an experimental feature (with a draft RFC [56]) that allows for sharing of memory between JavaScript's web workers[2]. SMC builds a high-resolution counter that can be used to reliably implement AnC in all the browsers that implement it. Both Firefox and Chrome currently support this feature, but it needs to be explicit enabled due to its experimental nature. We expect shared memory between JavaScript web workers to become a default-on mainstream feature in a near future. The second technique (Figure 4.4), time to tick (TTT), relies on the current low-precision `performance.now()` for building a timer that allows us to measure the difference between a cached reference and a memory reference, and allows us to implement AnC in low-jitter browsers such as Firefox.

[2]JavaScript webworkers are threads used for long running background tasks.

The impact of TTT and SMC goes beyond AnC. All previous timing attacks that were considered mitigated by browser vendors are still applicable today. We now discuss the TTT and SMC timers in further detail.

A. Time to Tick

The idea behind the TTT measurement, as shown in Figure 4.4, is quite simple. Instead of measuring how long a memory reference takes with the timer (which is no longer possible), we count how long it takes for the timer to tick *after* the memory reference takes place. More precisely, we first wait for `performance.now()` to tick, we then execute the memory reference, and then count by executing `performance.now()` in a loop until it ticks. If memory reference is a fast cache access, we have time to count more until the next tick in comparison to a memory reference that needs to be satisfied through main memory.

TTT performs well in situations where `performance.now()` does not have jitter and ticks at regular intervals such as in Firefox. We, however, believe that TTT can also be used in `performance.now()` with jitter as long as it does not *drift*, but it will require a higher number of measurements to combat jitter.

B. Shared Memory Counter

Our SMC counter uses a dedicated JavaScript web worker for counting through a shared memory area between the main JavaScript thread and the counting web worker. This means that during the attack, we are practically using a separate core for counting. Figure 4-3 shows how an attacker can use SMC to measure how long a memory reference takes. The thread that wants to perform the measurement (in our case the main thread) reads the counter and stores it in $c1$, executes the memory reference, and then reads the counter again and stores it in $c2$. Since the other thread is incrementing the counter during the execution of the memory reference, in case of a slow memory access, we see a larger $c2 - c1$ compared to the case where a faster cache access is taking place.

SMC is agnostic to the quality of `performance.now()` since it only relies on a separate *observer* core for its measurements.

C. Discussion

We designed a microbenchmark that performs a cached access and a memory access for a given number of iterations. We can do this by accessing a huge buffer (an improvised eviction set), ensuring the next access of a test buffer will be uncached. We measure this access time with both timers. We then know the next access time of the same test buffer will be cached. We time this access with both timers. In all cases, TTT and SMC could tell the difference between the two cases.

TTT is similar to the clock edge timer also described in concurrent work [35], but does not require a learning phase because it relies on counting the invocations of `performance.now()` insted. It is worth mentioning that the proposed fuzzy time defense for browsers [35], while expensive, is not effective against SMC.

We use TTT on Firefox and SMC on Chrome for our evaluation in Section VII. The shared memory feature, needed for SMC, is currently enabled by default in the nightly build of Firefox, implemented in Chrome [10], implemented and currently enabled under experimental flag in Edge [8]. We have notified major browser vendors, warning them of this danger.

V. Implementing AnC

Equipped with our TTT and SMC timers, we now proceed with the implementation of AnC described in Section III-D. We first show how we managed to trigger MMU walks when accessing our target heap and when executing code on our target JIT area in Section V-A. We then discuss how we identified the page offsets that store PT entries of a target virtual address in Sections V-B, V-C and V-D. In Sections V-E and V-F, we describe the techniques that we applied to observe the signal and uniquely identify the location of PT entries inside the cache lines that store them. In Sections V-G and V-H we discuss the techniques we applied to clear the MMU signal by flushing the page table caches and eliminating noise.

A. Triggering MMU Page Table Walks

In order to observe the MMU activities on the CPU caches we need to make sure that 1) we know the offset in pages within our buffer when we access the target, and 2) we are able to evict the TLB in order to trigger an MMU walk on the target memory location. We discuss how we achieved these goals for heap memory and JITed code.

1) Heap: We use the `ArrayBuffer` type to back the heap memory that we are trying to derandomize. An `ArrayBuffer` is always page-aligned which makes it possible for us to predict the relative page offset of any index in our target `ArrayBuffer`. Recent Intel processors have two levels of TLB. The first level consists of an instruction TLB (iTLB) and a data TLB (dTLB) while the second level is a larger unified TLB cache. In order to flush both the data TLB and the unified TLB, we access every page in a TLB eviction buffer with a larger size than the unified TLB. We later show that this TLB eviction buffer can be used to evict LLC cache sets at the desired offset as well.

2) Code: In order to allocate a large enough JITed code area we spray 2^{17} JavaScript functions in an `asm.js` [26] module. We can tune the size of these functions by changing the number of their statements to be compiled by the JIT engine. The machine code of these functions start from a browser-dependent but known offset in a page and follow each other in memory and since we can predict their (machine code) size on our target browsers, we know the relative offset of each function from the beginning of the `asm.js` object. In order to minimize the effect of these functions on the cache without affecting their size, we add an `if` statement in the beginning of all the functions in order not to execute their body. The goal is to hit a single cache line once executed so as to not obscure the pagetable cache line signals, but still maintain a large offset between functions. To trigger a PT walk when executing one of our functions, we need to flush the iTLB and the unified TLB. To flush the iTLB, we use a separate `asm.js` object and execute some of its functions that span enough pages beyond the size of the iTLB. To flush the unified TLB, we use the same TLB eviction buffer that we use for the heap.

As we will discuss shortly, AnC observes one page offset in each round. This allows us to choose the iTLB eviction functions and the page offset for the unified TLB eviction buffer in a way that does not interfere with the page offset under measurement.

B. PRIME+PROBE and the MMU Signal

The main idea behind ASLR⊕Cache is the fact that we can observe the effect of MMU's PT walk on the LLC. There are two attacks that we can implement for this purpose [49]: PRIME+PROBE or EVICT+TIME. To implement a PRIME+PROBE attack, we need to follow a number of steps:

1) Build optimal LLC eviction sets for all available page colors. An optimal eviction set is the precise number of memory locations (equal to LLC set-associativity) that once accessed, ensures that a target cache line has been evicted from the LLC cache set which hosts the target cache line.
2) Prime the LLC by accessing all the eviction sets.
3) Access the target virtual address that we want to derandomize, bringing its PT entries into LLC.
4) Probe the LLC by accessing all the eviction sets and measure which ones take longer to execute.

The eviction sets that take longer to execute presumably need to fetch one (or more) of their entries from memory. Since during the prime phase, the entries in the set have been brought to the LLC, and the only memory reference (besides TLB eviction set) is the target virtual address, four of these "probed" eviction sets have hosted the PT entries for the target virtual address. As we mentioned earlier, these cache sets uniquely identify the upper six bits of the PT entry offset at each PT level.

There are, however, two issues with this approach. First, building optimal LLC eviction sets from JavaScript, necessary for PRIME+PROBE, while has recently been shown to be possible [48] takes time, specially without a precise timer. Second, and more fundamental, we cannot perform the PRIME+PROBE attack reliably, because the very thing that we are trying to measure, will introduce noise in the measurements. More precisely, we need to flush the TLB before accessing our target virtual address. We can do this either before or after the priming step, but in either case evicting the TLB will cause the MMU to perform some unwanted PT walks. Assume we perform the TLB eviction *before* the prime step. In the middle of accessing the LLC eviction sets during the prime step, potentially many TLB misses will occur, resulting in PT walks that can potentially fill the already primed cache sets, introducing many false positives in the probe step. Now assume we perform the TLB eviction step *after* the prime step. A similar situation happens: some of the pages in the TLB eviction set will result in a PT walk, resulting in filling the already primed cache sets and again, introducing many false positives in the probe step.

Our initial implementation of AnC used PRIME+PROBE. It took a long time to build optimal eviction sets and ultimately was not able to identify the MMU signal due to the high ratio of noise. To resolve these issues, we exploited unique properties of our target in order not to build optimal eviction sets (Section V-C), and due to the ability to control the trigger (MMU's PT walk), we could opt for a more exotic EVICT+TIME attack that allowed us to avoid the drawbacks of PRIME+PROBE (Section V-D).

C. Cache Colors Do Not Matter for AnC

Cache-based side-channel attacks benefit from the fine-grained information available in the state of cache after a secret operation—the cache sets that were accessed by a victim. A cache set is uniquely identified by a color (i.e., page color) and a page (cache line) offset. For example, a cache set in an LLC with 8192 cache sets can be identified by a (color, offset) tuple, where $0 \leq \text{color} < 128$ and $0 \leq \text{offset} < 64$.

ASLR encodes the secret (i.e., the randomized pointer) in the page offsets. We can build one eviction set for each of the 64 cache line offsets within a page, evicting all colors of that cache line offset with each set. The only problem is that the PT entries at different PT levels may use different page colors, and hence, show us overlapping offset signals. But given that we can control the observed virtual address, relative to our target virtual address, we can control PT entry offsets within different PT levels as discussed in Section III-D to resolve this problem.

Our EVICT+TIME attack, which we describe next, does not rely on the execution time of eviction sets. This means that we do not require to build optimal eviction sets. Coupled with the fact that ASLR is agnostic to color, we can use *any* page as part of our eviction set. There is no way that page tables might be allocated using a certain color layout scheme to avoid showing this signal, as all of them appear in our eviction sets. This means that with a sufficiently large number of memory pages, we can evict any PT entry from LLC (and L1D and L2) at a given page offset, not relying on optimal eviction sets that take a long time to build.

D. EVICT+TIME Attack on the MMU

The traditional side-channel attacks on cryptographic keys or for eavesdropping benefit from observing the state of the entire LLC. That is the reason why side-channel attacks such as PRIME+PROBE [48] and FLUSH+RELOAD [65] that allow attackers to observe the entire state of the LLC are popular [27], [30], [31], [39], [49], [67].

Compared to these attacks, EVICT+TIME can only gain information about one cache set at each measurement round, reducing its bandwidth compared to attacks such as PRIME+PROBE [49]. EVICT+TIME further makes a strong assumption that the attacker can observe the performance of the victim as it performs the secret computation. While these properties often make EVICT+TIME inferior compared to more recent cache attacks, it just happens that it easily applies to AnC: AnC does not require a high bandwidth (e.g., to break a cryptographic key) and it can monitor the performance of the victim (i.e., the MMU) as it performs the secret computation (i.e., walking the PT). EVICT+TIME implements AnC in the following steps:

1) Take a large enough set of memory pages to act as an eviction set.
2) For a target cache line at offset t out of the possible 64 offsets, evict that cache line by reading the same

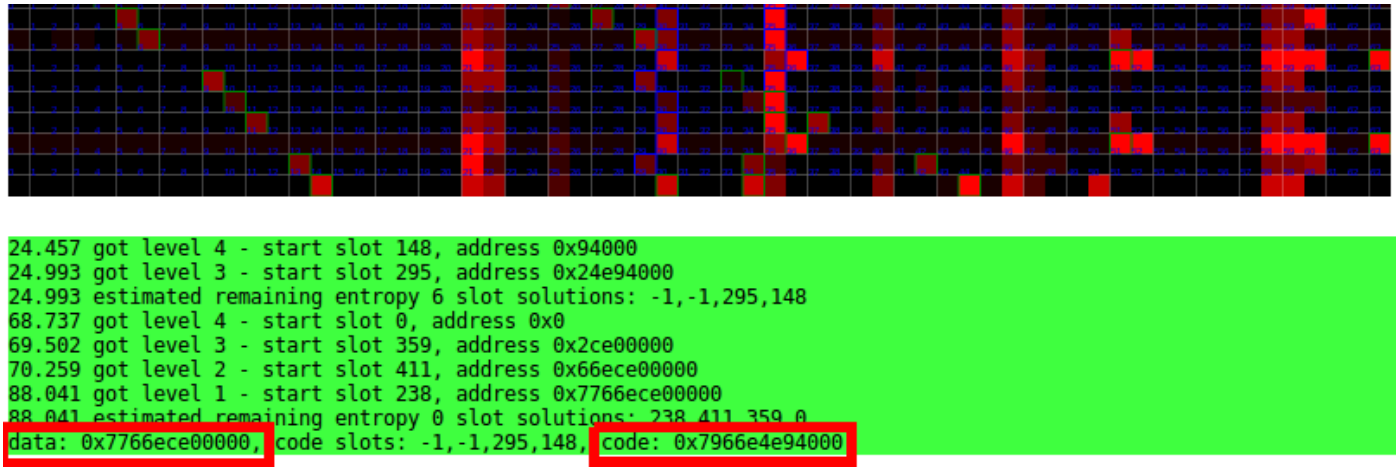

Fig. 5. The MMU memorygram as we access the target pages in a pattern that allows us to distinguish between different PT signals. Each row shows MMU activity for one particular page within our buffer. The activity shows different cache lines within the page table pages that are accessed during the MMU translation of this page. The color is brighter with increased latency when there is MMU activity. We use different access patterns within our buffer to distinguish between signals of PT entries at different PT levels. For example, the stair case (on the left) distinguishes the PT entry at PTL1 since we are accessing pages that are 32 KB apart in succession (32 KB is 8 PT entries at PTL1 or a cache line). Hence, we expect this access pattern to make the moving PTL1 cache line create a stair case pattern. Once we identify the stair case, it tells us the PT entry slot in PTL1 and distinguishes PTL1 from the other levels. Once a sufficiently unique solution is available for both code and data accesses at all PT levels, AnC computes the (derandomized) 64 bit addresses for code and data, as shown.

offset in all the memory pages in the eviction set. Accessing this set also flushes the dTLB and the unified TLB. In case we are targeting code, flush the iTLB by executing functions at offset t.

3) Time the access to the target virtual address that we want to derandomize at a different cache line offset than t, by dereferencing it in case of the heap target or executing the function at that location in case of the code target.

The third step of `EVICT+TIME` triggers a PT walk and depending on whether t was hosting a PT entry cache line, the operation will take longer or shorter. `EVICT+TIME` resolves the issues that we faced with `PRIME+PROBE`: first, we do not need to create optimal LLC eviction sets, since we do not rely on eviction sets for providing information and second, the LLC eviction set is unified with the TLB eviction set, reducing noise due to fewer PT walks. More importantly, these PT walks (due to TLB misses) result in significantly fewer false positives, again because we do not rely on probing eviction sets for timing information.

Due to these improvements, we could observe cache line offsets corresponding to the PT entries of the target virtual address when trying `EVICT+TIME` on all 64 possible cache line offsets in JavaScript both when dereferencing heap addresses and executing JIT functions. We provide further evaluation in Section VII, but before that, we describe how we can uniquely identify the offset of the PT entries inside the cache lines identified by `EVICT+TIME`.

E. Sliding PT Entries

At this stage, we have identified the (potentially overlapping) cache line offsets of the PT entries at different PT levels. There still remains two sources of entropy for ASLR: it is not possible to distinguish which cache line offset belongs to which PT level, and the offset of the PT entry within the cache line is not yet known. We address both sources of entropy by allocating a sufficiently large buffer (in our case a 2 GB allocation) and accessing different locations within this buffer in order to derandomize the virtual address where the buffer has been allocated from. We derandomize PTL1 and PTL2 differently than how we derandomize PTL3 and PTL4. We describe both techniques below.

1) Derandomizing PTL1 and PTL2: Let's start with the cache line that hosts the PT entry at PTL1 for a target virtual address v. We observe when one of the (possible) 4 cache line change as we access $v + i \times 4$ KB for $i = \{1, 2, \ldots, 8\}$. If one of the cache lines changes at i, it immediately provides us with two pieces of information: the changed cache line is hosting the PT entry for PTL1 and the PTL1's PT entry offset for v is $8 - i$. We can perform the same technique for derandomizing the PT entry at PTL2, but instead of increasing the address by 4 KB each time, we now need to increase by 2 MB to observe the same effect for PTL2. As an example, Figure 5 shows an example MMU activity that AnC observes as we change the cache line for the PT entry at PTL1.

2) Derandomizing PTL3 and PTL4: As we discussed in Section III-E, in order to derandomize PTL3, we require an 8 GB crossing in the virtual address space within our 2 GB allocation and to derandomize PTL4, we require a 4 TB virtual address space crossing to happen within our allocation. We rely on the behavior of memory allocators, discussed in Section VI, in the browsers to ensure that one of our (many) allocations satisfies this property. But assuming that we have a cache line change at PTL3 or PTL4, we would like to detect and derandomize the corresponding level. Note that a cache line crossing at PTL4 will inevitably cause a cache line crossing at PTL3 too.

Remember that each PT entry at PTL3 covers 1 GB of virtual memory. Hence, if a cache line crossing at PTL3 happens within our 2 GB allocation, then our allocation could cover either two PTL3 PT entries, when crossing is exactly at the middle of our buffer, or three PT entries. Since a crossing

exactly in the middle is unlikely, we consider the case with three PT entries. Either two of the three or one of three PT entries are in the new cache line. By observing the PTL3 cache line when accessing the first page, the middle page, and the last page in our allocation, we can easily distinguish between these two cases and fully derandomize PTL3.

A cache line crossing at PTL4 only occurs if the cache line at PTL3 is in the last slot in its respective PT page. By performing a similar technique (i.e., accessing the first and last page in our allocation), if we observe a PT entry cache line PTE_2 change from the last slot to the first slot and another PT entry cache line PTE_1 move one slot ahead, we can conclude a PTL4 crossing and uniquely identify PTE_2 as the PT entry at PTL3 and PTE_1 as the PT entry at PTL4.

F. ASLR Solver

We created a simple solver in order to rank possible solutions against each other as we explore different pages within our 2 GB allocation. Our solver assumes 512 possible PT entries for each PT level for the first page of our allocation buffer, and ranks the solutions at each PT level independently of the other levels.

As we explore more pages in our buffer according to patterns that we described in Section V-E1 and Section V-E2, our solver gains a significant confidence in one of the solutions or gives up and starts with a new 2 GB allocation. A solution will always derandomizes PTL1 and PTL2 and also PTL3 and PTL4 if there was a cache line crossing at these PT levels.

G. Evicting Page Table Caches

As mentioned in Section III-B, some processors may cache the translation results for different page table levels in their TLB. AnC needs to evict these caches in order to observe the MMU signal from all PT levels. This is straightforward: we can access a buffer that is larger than that the size of these caches as part of our TLB and LLC eviction.

For example, a Skylake i7-6700K core can cache 32 entries for PTL2 look ups. Assuming we are measuring whether there is page table activity in the i-th cache line of page table pages, accessing a 64 MB (i.e., 32×2 MB) buffer at $0 + i \times 64$, 2 MB $+ i \times 64$, 4 MB $+ i \times 64$, ..., 62 MB $+ i \times 64$ will evict the PTL2 page table cache.

While we needed to implement this mechanism natively to observe the signal on all PT levels, we noticed that due to the JavaScript runtime activity, these page table caches are naturally evicted during our measurements.

H. Dealing with Noise

The main issue when implementing side-channel attacks is noise. There are a number of countermeasures that we deploy in order to reduce noise. We briefly describe them here:

1) Random exploration: In order to avoid false negatives caused by the hardware prefetcher, we select t (the page offset that we are evicting) randomly within the possible remaining offsets that we (still) need to explore. This randomization also helps by distributing the localized noise caused by system events.

2) Multiple rounds for each offset: In order to add reliability to the measurements, we sample each offset multiple times ('rounds') and consider the median for deciding a cached versus memory access. This simple strategy reduces the false positives and false negatives by a large margin. For large scale experiment and visualization on the impact of measurement rounds vs other solving parameters, please see section VII-C.

For an AnC attack, false negatives are harmless due to the fact that the attacker can always retry with a new allocation as we discuss in the next section. We evaluate the success rate, false positives and false negatives of the AnC attack using Chrome and Firefox in Section VII.

I. Discussion

We implemented two versions of AnC. A native implementation in C in order to study the behavior of the MMU PT walk activity without the JavaScript interference and a JavaScript-only implementation.

We ported the native version to different architectures and Microsoft Windows 10 to show the generality of AnC presented in Section VII-D. Our porting efforts revolved around implementing a native version of SMC (Section IV-B) to accurately differentiate between cached and uncached memory accesses on ARM which only provides coarse-grained (0.5μs) timing mechanism and dealing with different page table structures. Our native implementation amounts to 1283 lines of code

Our JavaScript-only implementation works on Chrome and Firefox browsers and is aimed to show the real-world impact of the AnC attack presented in various experiments in Section VII. We needed to handtune the JavaScript implementation using `asm.js` [26] in order to make the measurements faster and more predictable. This limited our allocation size to the maximum of 2 GB. Our JavaScript implementation amounts to 2370 lines of code.

VI. Allocators and AnC

As mentioned in Section V-E2, we rely on the behavior of the memory allocators in the browsers to get an allocation that crosses PTL3 and PTL4 in the virtual address space. We briefly discuss the behavior of the memory allocators in Firefox and Chrome and how we could take advantage of them for AnC.

A. Memory Allocation in Firefox

In Firefox, memory allocation is based on demand paging. Large object allocations from a JavaScript application in the browser's heap is backed by `mmap` without `MAP_POPULATE`. This means that memory is only allocated when the corresponding page in memory is touched.

Figure 6 shows how Firefox's address space is laid out in memory. Firefox uses the stock `mmap` provided by the Linux kernel in order to randomize the location of JITed code and heap using 28 bits of entropy. The (randomized) base address for `mmap` is only chosen once (by the OS) and after that the subsequent allocations by Firefox grow backward from the previous allocation towards low (virtual) memory. If an object is deleted, Firefox reuses its virtual memory for the subsequent allocations. Hence, to keep moving backward in the virtual

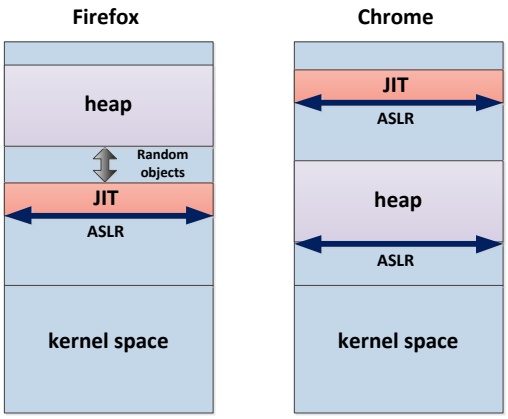

Fig. 6. Low-level memory allocation strategies in Firefox and Chrome. Firefox uses the stock `mmap` in order to gain ASLR entropy for its JITed code and heap while Chrome does not rely on `mmap` for entropy and randomizes each large code/data allocation.

address space, a JavaScript application should linger to its old allocations.

An interesting observation that we made is that a JavaScript application can allocate TBs of (virtual) memory for its objects as long as they are not touched. AnC exploits this fact and allocates a few 2 GB buffers for forcing a cache line change at PTL3 (i.e., 1 bit of entropy remaining), or if requested, a large number of 2 GB objects forcing a cache line change at PTL4 (i.e., fully derandomized).

To obtain a JIT code pointer, we rely on our heap pointer obtained in the previous step. Firefox reserves some virtual memory in between JIT and heap. We first spray a number of JITed objects to exhaust this area right before allocating our heap. This ensures that our last JITed object is allocated before our heap. There are however a number of other objects that JavaScript engine of Firefox allocates in between our last JITed object and the heap, introducing additional entropy. As a result, we can predict the PTL3 and PTL4 slots of our target JIT pointer using our heap pointer, but the PTL1 and PTL2 slots remain unknown. We now deploy our code version of AnC to find PTL1 and PTL2 slots of our code pointer, resulting in a full derandomization.

B. Memory Allocation in Chrome

In Chrome, memory allocations are backed by `mmap` and initialized. This means that every allocation of a certain size will consume the same amount of physical memory (plus a few pages to back its PT pages). This prohibits us from using multiple allocations similar to Firefox. Chrome internally chooses the randomized location for `mmap` and this means that for every new large object (i.e., a new heap). This allows for roughly 35 bits out of the available 36 bits of entropy provided by hardware (Linux kernel is always mapped in the upper part of the address space). Randomizing every new heap is devised in order to protect against the exploitation of use-after-free bugs that often rely on predictable reuse on the heap [58].

AnC exploits this very protection in order to acquire an object that crosses a PTL3 or a PTL4 cache line. We first

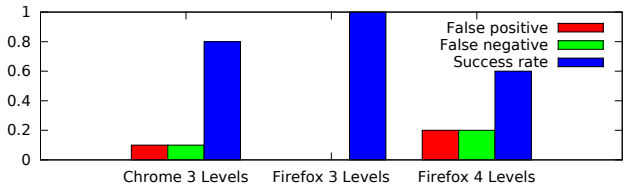

Fig. 7. The success rate, false positive and false negative rate of AnC.

allocate a buffer and use AnC to see whether there are PTL3 or PTL4 cache line crossings. If this is not the case, we delete the old buffer and start with a new allocation. Based on a given probability p, AnC's ith allocation will cross PTL3 based on the following formula using a Bernoulli trial (assuming a 2 GB allocation): $\sum_1^i \frac{1}{4}(\frac{3}{4})^i \geq p$. Calculating for average (i.e., $p = 0.5$), AnC requires around 6.5 allocations to get a PTL3 crossing. Solving the same equation for a PTL4 crossing, AnC requires on average 1421.2 allocations to get a crossing. In a synthetic experiment with Chrome, we observed a desired allocation after 1235 trials. While nothing stops AnC from derandomizing PTL4, the large number of trials makes it less attractive for attackers.

This technique works the same for allocations of both heap and JITed objects. The current version of AnC implements derandomization of heap pointers on Chrome using this technique.

VII. Evaluation

We show the success rate and feasibility of the AnC attack using Firefox and Chrome. More concretely, we like to know the success rate of AnC in face of noise and the speed in which AnC can reduce the ASLR entropy. We further compare AnC with other software-based attacks in terms of requirements and performance and showcase an end-to-end exploit using a real vulnerability with pointers leaked by AnC. For the evaluation of the JavaScript-based attacks, we used an Intel Skylake i7-6700K processor with 16 GB of main memory running Ubuntu 16.04.1 LTS as our evaluation testbed. We further show the generality of the AnC attack using various CPU architectures and operating systems.

A. Success Rate

To evaluate the reliability of AnC, we ran 10 trials of the attack on Firefox and Chrome and report success rate, false positive and false negative for each browser. For the ground truth, we collected run time statistics from the virtual mappings of the browser's process and checked whether our guessed addresses indeed match them. In case of a match, the trial counts towards the success rate. In case AnC fails to detect a crossing, that trial counts towards false negatives. False negatives are not problematic for AnC, since it results in a retry which ultimately makes the attack take longer. False positives, however, are problematic and we count them when AnC reports an incorrect guessed address. In case of Chrome, we report numbers for when there are PTL3 cache line crossings. We did not observe a PTL4 crossing (i.e., all levels) in our trials. In case of Firefox, we performed the measurement by restarting it to get a new allocation each time

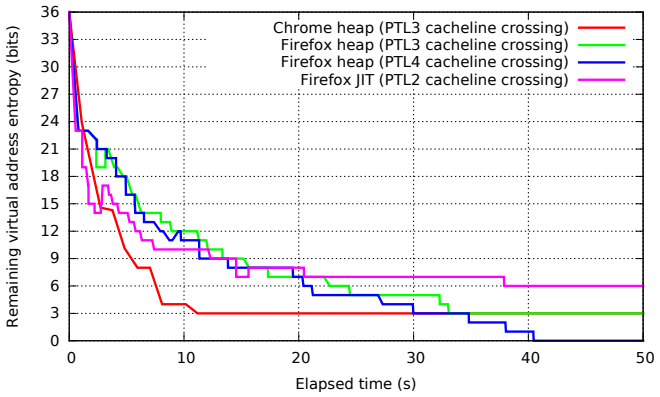

Fig. 8. Reduction in heap (Chrome and Firefox) and JIT (Firefox) ASLR entropy over time with the AnC attack. At the code stage of the attack, AnC already knows the exact PTE slots due to the obtained heap pointer. This means that for the code, the entropy reaches zero at 37.9 s, but our ASLR solver is agnostic to this information.

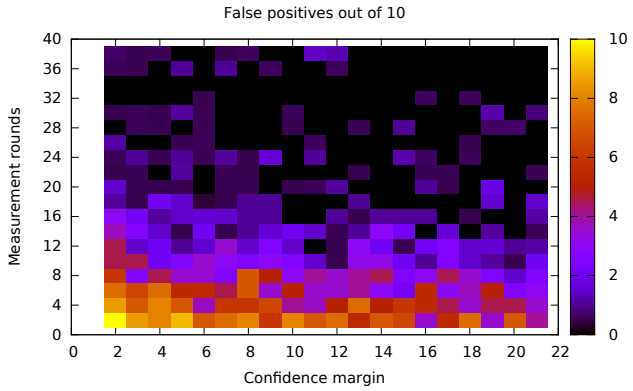

Fig. 9. The effects of our noise reduction techniques on the fidelity of AnC. The plotted intensity indicates false positive (wrong answer) rate, as a function of solver confidence requirement (X axis) vs. number of measurement repetitions (Y axis). This shows that the number of measurement rounds is critical to reliable conclusions while the confidence margin improves the results further.

and we used it to derandomize both JIT and heap pointers. We report numbers for both PTL3 and PTL4 cache line crossings.

Figure 7 reports the results of the experiment. In the case of Chrome, AnC manages to successfully recover 33 bits out of the 35 bits of the randomized heap addresses in 8 of the cases. In the case of Firefox, AnC reduces the entropy of JIT and heap to a single bit in all 10 cases. Getting the last bit is successful in 6 of the cases with 2 cases as false positive. The PTE hosting the last bit of entropy (i.e., in PTL4) is often shared with other objects in the Firefox runtime, making the measurements more noisy compared to PTEs in other levels.

B. Feasibility

To evaluate the feasibility of AnC from an attacker's perspective, we report on the amount of time it took AnC to reduce ASLR's entropy in the same experiment that we just discussed. Allocating the buffers that we use for the AnC do not take any time on Chrome. On Firefox, for crossing a cache line in PTL3, the buffer allocations take 5.3 s, and for crossing a cache line in PTL4, the buffer allocations take 72.7 s.

Figure 8 shows ASLR entropy as a function of time when AnC is applied to Firefox and Chrome as reported by our solver described in Section V-F. Both heap and code derandomization are visualized. Note that due to the noise our solver sometimes needs to consider more solutions as time progress resulting in a temporary increase in the estimated entropy. More importantly, our solver is agnostic to the limitations of the underlying ASLR implementations and always assumes 36 bits of entropy (the hardware limit). This means that AnC can reduce the entropy even if the implementation uses all available bits for entropy which is not possible in practice. In the case of Chrome, in 11.2 s the entropy is reduced to only 2 bits (our solver does not know about kernel/user space split and reports 3 bits). In the case of Firefox, in 33.1 s the entropy is reduced to 1 bit when crossing a cache line in PTL3 (our solver does not know about `mmap`'s entropy) and in 40.5 s to zero when crossing a cache line in PTL4.

As discussed in Section VI, after obtaining a heap pointer, our AnC attack proceeds to obtain a JITed code pointer. At this stage of the attack, AnC already knows the upper two PT slots of the JIT area (our solver does not know about this information). After 37.9 s, AnC reduces the code pointer entropy to only 6 bits as reported by our ASLR solver. These are the same entropy bits that are shared with our heap pointer. This means that at this stage of the attack we have completely derandomized code and heap ASLR in Firefox.

C. Noise

We evaluated the efficacy of our techniques against noise in the system. As we mentioned earlier, we used measurement rounds in order to combat temporal noise and a scoring system in our solver in order to combat more persistent noise in the system.

Figure 9 shows different configuration of AnC with respect to the number of rounds and the confidence margin in our solver. As we decrease the number of rounds or confidence margin, we observe more false positives in the system. These results show that with our chosen configuration (confidence margin = 10 and rounds = 20) these techniques are effective against noise.

D. Generalization

Our evaluation so far shows that AnC generalizes to different browsers. We further studied the generality of the AnC attack by running our native implementation on different CPU architectures.

Table I shows a successful AnC attack on 11 different CPU architectures including Intel, ARM and AMD. We did not find an architecture on which the AnC attack was not possible. Except for on ARMv7, we could fully derandomize ASLR on all architectures. On ARMv7, the top level page table spans four pages which introduces two bits of entropy in the high virtual address bits. On ARMv7 with physical address extension (i.e., ARMv7+LPAE), there are only four entries in the top level page table which fit into a cache line, resulting again in two remaining bits of entropy. On ARMv8, AnC fully solves ASLR given a similar page table structure to x86_64.

TABLE I. CPU MODELS VERIFIED TO BE AFFECTED BY ANC.

Vendor	CPU Model	Year	Microarchitecture
Intel	Core i7-6700K	2015	Skylake
Intel	Core i3-5010U	2015	Broadwell
Allwinner	A64	2015	ARMv8-A, Cortex-A53
Nvidia	Jetson TK-1	2014	ARMv7, Cortex-A15
Nvidia	Tegra K1 CD570M	2014	ARMv7+LPAE, Cortex-A15
Intel	Core i7-4510U	2014	Haswell
Intel	Celeron N2840	2014	Silvermont
Intel	Atom C2750	2013	Silvermont
AMD	FX-8320 8-Core	2012	Piledriver
Intel	Core i7-3632QM	2012	Ivy Bridge
Intel	E56xx/L56xx/X56xx	2010	Westmere

Both ARM and AMD processors have exclusive LLCs compared to Intel. These results show that AnC is agnostic to the inclusivity of the LCC. We also successfully performed the AnC attack on the Microsoft Windows 10 operating system.

E. Comparison against Other Derandomization Attacks

Table II compares existing derandomization attacks against ASLR with AnC. Note that all the previous attacks rely on software features that can be mitigated. For example, the most competitive solution, Dedup Est Machina [6], relies on memory deduplication, which was only available natively on Windows and has recently been turned off by Microsoft [14], [46]. Other attacks require crash-tolerance or crash-resistance primitives, which are not always available and also yield much more visible side effects. We also note that AnC is much faster than all the existing attacks, completing in 150 seconds rather than tens of minutes.

F. End-to-end attacks

Modern browsers deploy several defenses such as ASLR or segment heaps to raise the bar against attacks [64]. Thanks to such defenses, traditional browser exploitation techniques such as heap spraying are now much more challenging to execute, typically forcing the attacker to derandomize the address space before exploiting a given vulnerability [54].

For example, in a typical vtable hijacking exploit (with many examples in recent Pwn2Own competitions), the attacker seeks to overwrite a vtable pointer to point to a fake vtable using type confusion [38] or other software [61] or hardware [6] vulnerabilities. For this purpose, the attacker needs to leak code pointers to craft the fake vtable and a heap pointer to the fake vtable itself. In this scenario, finding a dedicated information disclosure primitive is normally a sine qua non to mount the attack. With AnC, however, this is no longer a requirement: the attacker can directly leak heap and code addresses she controls with a cache attack against the MMU. This significantly reduces the requirements for end-to-end attacks in the info leak era of software exploitation [54].

As an example, consider CVE-2013-0753, a use-after-free vulnerability in Firefox. An attacker is able to overwrite a pointer to an object and this object is later used to perform a virtual function call, using the first field of the object to reference the object's vtable. On 32-bit Firefox, this vulnerability can be exploited using heap spraying, as done in the publicly available Metasploit module (https://goo.gl/zBjrXW). However, due to the much larger size of the address space, an information disclosure vulnerability is normally required on 64-bit Firefox. With AnC, however, we re-injected the vulnerability in Firefox and verified an attacker can mount an end-to-end attack without an additional information disclosure vulnerability. In particular, we were able to (i) craft a fake vtable containing AnC-leaked code pointers, (ii) craft a fake object pointing to the AnC-leaked address of the fake vtable, (iii) trigger the vulnerability to overwrite the original object pointer with the AnC-leaked fake object pointer, and (iv) hijack the control flow.

TABLE II. DIFFERENT ATTACKS AGAINST USER-SPACE ASLR.

Attack	Time	Probes	Pointers	Requirement
BROP [5]	20 m	4000	Code	Crash tolerance
CROP [19]	243 m	2^{28}	Heap/code	Crash resistance
Dedup Est Machina [6]	30 m	0	Heap/code	Deduplication
AnC	150 s	0	Heap/code	Cache

G. Discussion

We showed how AnC can quickly derandomize ASLR towards end-to-end attacks in two major browsers with high success rate. For example, AnC can derandomize 64 bit code and heap pointers *completely* in 150 seconds in Firefox. We also showed that AnC has a high success rate.

VIII. IMPACT ON ADVANCED DEFENSES

The AnC attack casts doubt on some of the recently proposed advanced defenses in academia. Most notably, AnC can fully break or significantly weaken defenses that are based on information hiding in the address-space and leakage-resilient code randomization. We discuss these two cases next.

A. Information Hiding

Hiding security-sensitive information (such as code pointers) in a large 64 bit virtual address-space in a common technique to bootstrap more advanced security defenses [9], [15], [36], [42], [52]. Once the location of the so-called *safe-region* is known to the attacker, she can compromise the defense mechanism in order to engage in, for example, control-flow hijacking attacks [17], [19], [23], [47].

With the AnC attack, in situations where the attacker can operate arbitrary memory accesses, the unknown (randomized) target virtual address can be derandomized. Already triggering a single memory reference in the safe-region allows AnC to reduce the entropy of ASLR on Linux to $log_2(2^{10} \times 4!) = 10.1$ bits (1 bit on the PTL4 and 3 bits on other levels) and on Windows to $log_2(2^9 \times 4!) = 9.4$ bits (9 bits on PTL3, PTL2 and PTL1). Referencing more virtual addresses in different memory pages allows AnC to reduce the entropy further.

For example, the original linear table implementation for the safe-region in CPI [36], spans 2^{42} of virtual address-space and moves the protected code pointers in this area. This means that, since the relative address of secret pointers with respect to each other is known, an attacker can also implement sliding to find the precise location of the safe-region using AnC, breaking CPI. Similarly, the more advanced two-level lookup table or hashtable versions of CPI [37] for hiding the safe-region will be prone to AnC attacks by creating a sliding mechanism on the target (protected) pointers.

We hence believe that randomization-based information hiding on modern cache-based CPU architectures is inherently prone to cache attacks when the attacker controls memory accesses (e.g., web browsers). We thereby caution future defenses not to rely on ASLR as a pivotal building block, even when problematic software features such as memory deduplication [6] or memory overcommit [47] are disabled.

B. Leakage-resilient Code Randomization

Leakage-resilient code randomization schemes based on techniques like XnR and code pointer hiding [1], [7], [13] aim to provide protection by making code regions execute-only and the location of target code in memory fully unpredictable. This makes it difficult to perform code-reuse attacks given that the attacker cannot directly or indirectly disclose the code layout.

AnC weakens all these schemes because it can find the precise memory location of executed code without reading it (Section VII-B). Like Information Hiding, the execution of a single function already leaves enough traces in the cache from the MMU activity to reduce its address entropy significantly.

IX. Mitigations

Detection It is possible to detect an on-going AnC attack using performance counters [50]. These types of anomaly-based defenses are, however, prone to false positives and false negatives by nature.

Cache coloring Partitioning the shared LLC can be used to isolate an application (e.g., the browser) from the rest of the system [33], [39], but on top of complications in the kernel's frame allocator [40], it has performance implications both for the operating system and the applications.

Secure timers Reducing the accuracy of the timers [35], [43], [59] makes it harder for attackers to tell the difference between cached and memory accesses, but this option is often costly to implement. Further, there are many other possible sources to craft a new timer. Prior work [11] shows it is hard if not impossible to remove all of them even in the context of simple microkernels. This is even more complicated with browsers, which are much more complex and bloated with features.

Isolated caches Caching PT entries in a separate cache rather than the data caches can mitigate AnC. Having a separate cache just for page table pages is quite expensive in hardware and adopting such solution as a countermeasure defeats the purpose of ASLR—providing a low-cost first line of defense.

The AnC attack exploits fundamental properties of cache-based architectures, which improve performance by keeping hot objects in a faster but smaller cache. Even if CPU manufacturers were willing to implement a completely isolated cache for PT entries, there are other caches in software that can be exploited to mount attacks similar to AnC. For example, the operating system often allocates and caches page table pages on demand as necessary [24]. This optimization may yield a timing side channel on memory management operations suitable for AnC-style attacks. Summarizing, we believe that the use of ASLR is fundamentally insecure on cache-based architectures and, while countermeasures do exist, they can only limit, but not eliminate the underlying problem.

X. Related Work

A. Derandomizing ASLR in Software

Bruteforcing, if allowed in software, is a well-known technique for derandomizing ASLR. For the first time, Shacham et al. [55] showed that it is possible to systematically derandomize ASLR on 32 bit systems. BROP [5] bruteforces values in the stack byte-by-byte in order to find a valid 64 bit return address using a few hundreds of trials. Bruteforcing, however, is not always possible as it heavily relies on application-specific behavior [5], [19]. Further, significant number of crashes can be used by anomaly detection systems to block an ongoing attack [55].

A key weakness of ASLR is the fact that large (virtual) memory allocations reduce its entropy. After a large memory allocation, the available virtual address-space is smaller for the next allocation, reducing entropy. This weakness has recently been exploited to show insecurities in software hardening techniques that rely on ASLR [23], [47].

Memory deduplication is an OS or virtual machine monitor feature that merges pages across processes or virtual machines in order to reduce the physical memory footprint. Writing to merged pages results in a copy-on-write that is noticeably slower than writing to a normal. This timing channel has been recently used to bruteforce ASLR entropy in clouds [2] or inside browsers [6].

All these attacks against ASLR rely on a flaw in software that allows an attacker to reduce the entropy of ASLR. While software can be fixed to address this issue, the microarchitectural nature of our attack makes it difficult to mitigate. We hence recommend decommissioning ASLR as a defense mechanism or a building block for other defense mechanisms.

B. Timing Attacks on CPU Caches

Closest to ASLR⊕Cache, in terms of timing attacks on CPU caches is the work by Hund et al. [27] in breaking Windows kernel-space ASLR from a local user-space application. Their work, however, assumes randomized physical addresses (instead of virtual addresses) with a few bits of entropy, and that the attacker has the ability to reference arbitrary virtual addresses. Similar attacks geared towards breaking kernel-level ASLR from a controlled process have been recently documented using the `prefetch` instruction [25], hardware transactional memory [32] and branch prediction [18]. In our work, we derandomized high-entropy virtual addresses in the browser process inside a sandboxed JavaScript. To the best of our knowledge, AnC is the first attack that breaks user-space ASLR from JavaScript using a cache attack, significantly increasing the impact of these types of attacks.

Timing side-channel attack on CPU caches have also been used to leak private information, such as cryptographic keys, mouse movements, and etc. The `FLUSH+RELOAD` attack [31], [65], [67] leaks data from a sensitive process by exploiting the timing differences when accessing cached data. `FLUSH+RELOAD` assumes the attacker has access to victims' code pages either via the shared page cache or some form of memory deduplication. The `PRIME+PROBE` attack [39], [49] lifts this requirement by only relying on cache misses from the attacker's process to infer the behavior of the victim's

process when processing secret data. This relaxation makes it possible to implement PRIME+PROBE in JavaScript in the browser [48], significantly increasing the impact of cache side-channel attacks for the Internet users.

While FLUSH+RELOAD and PRIME+PROBE observe the change in the state of the entire cache, the older EVICT+TIME [49] attack, used for recovering AES keys, observes the state of one cache set at a time. In situations where information on the entire state of the cache is necessary, EVICT+TIME does not perform as effectively as the other attacks, but it has a much higher signal to noise ratio since it is only observing one cache set at a time. We used the reliability of EVICT+TIME for observing the MMU signal.

C. Defending Against Timing Attacks

As we described in Section IX mitigating AnC is difficult. However, we discuss some attempts for reducing the capability of the attackers to perform timing side-channel attacks.

At the hardware-level, TimeWarp [43] reduces the fidelity of timers and performance counters to make it difficult for attackers to distinguish between different microarchitectural events. Stefan et al. [59] implement a new CPU instruction scheduling algorithm that is indifferent to timing differences from underlying hardware components, such as the cache, and is, hence, secure against cache-based timing attacks.

At the software-level, major browsers have reduced the accuracy of their timers in order to thwart cache attacks from JavaScript. Concurrent to our efforts, Kohlbrenner and Shacham [35] show that it is possible to improve the degraded timers by looking at when the degraded clock ticks and proposed introducing noise in the timer and the event loop of JavaScript. In this paper, we show that it is possible to build more accurate timers. TTT is similar to the clock edge timer [35], but does not require a learning phase. Others have warned about the dangers of shared memory in the browser for cache attacks [53]. We showed accurate timing through shared memory makes the AnC attack possible

Page coloring, in order to partition the shared cache, is another common technique for defending against cache side-channel attacks [39]. Kim et al. [33] propose a low-overhead cache isolation technique to avoid cross-talk over shared caches. Such techniques could be retrofitted to protect the MMU from side-channel attacks from JavaScript, but as mentioned in Section IX, they suffer from deployability and performance problems. As a result, they have not been adopted to protect against cache attacks in practical settings. By dynamically switching between diversified versions of a program, Crane et al. [12] change the mapping of program locations to cache sets, making it difficult to perform cache attacks on program's locations. Our AnC attack, however, targets the MMU operations and can already reduce the ASLR entropy significantly as soon as one program location is accessed.

D. Other Hardware-based Attacks

Fault attacks are pervasive for extracting secrets from secure processors [21]. Power, thermal and electromagnetic field analysis have been used for building covert channels and extracting cryptographic keys [3], [20], [41]. Recent Rowhammer attacks show the possibility of compromising the browser [6], cloud virtual machines [51] and mobile devices [62] using wide-spread DRAM disturbance errors [34].

XI. Conclusions

In this paper, we described how ASLR is fundamentally insecure on modern architectures. Our attack relies on the interplay between the MMU and the caches during virtual to physical address translation—core hardware behavior that is central to efficient code execution on modern CPUs. The underlying problem is that the complex nature of modern microarchitectures allows attackers with knowledge of the architecture to craft a carefully chosen series of memory accesses which manifest timing differences that disclose what memory is accessed where and to infer all the bits that make up the address. Unfortunately, these timing differences are fundamental and reflect the way caches optimize accesses in the memory hierarchy. The conclusion is that such caching behavior and strong address space randomization are mutually exclusive. Because of the importance of the caching hierarchy for the overall system performance, all fixes are likely to be too costly to be practical. Moreover, even if mitigations are possible in hardware, such as separate cache for page tables, the problems may well resurface in software. We hence recommend ASLR to no longer be trusted as a first line of defense against memory error attacks and for future defenses not to rely on it as a pivotal building block.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate the disclosure of AnC to the affected hardware and software vendors. Most of them acknowledged our findings and we are closely working with some of them to address some of the issues raised by AnC.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. Stephan van Schaik helped us observe the MMU signal on ARM and AMD processors. 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.

References

[1] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. CCS'14.

[2] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT'15.

[3] D. B. Bartolini, P. Miedl, and L. Thiele. On the Capacity of Thermal Covert Channels in Multicores. EuroSys'16.

[4] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-dimensional Page Walks for Virtualized Systems. ASPLOS XIII.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking Blind. SP'14.

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

[7] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. NDSS'16.

[8] Making SharedArrayBuffer to be experimental. https://github.com/Microsoft/ChakraCore/pull/1759.

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

[10] Shared Array Buffers, Atomics and Futex APIs. https://www.chromestatus.com/feature/4570991992766464.

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

[12] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. NDSS'15.

[13] 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.

[14] CVE-2016-3272. https://goo.gl/d8jqgt.

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

[16] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. NDSS'15.

[17] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. SP'15.

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

[19] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. NDSS'16.

[20] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. CCS'16.

[21] C. Giraud and H. Thiebeauld. A Survey on Fault Attacks. CARDIS'04.

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

[23] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining Entropy-based Information Hiding (And What to do About it). SEC'16.

[24] M. Gorman. Understanding the Linux virtual memory manager.

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

[26] D. Herman, L. Wagner, and A. Zakai. asm.js. http://asmjs.org/spec/latest/.

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

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

[29] Intel 64 and IA-32 Architectures Software Developer's Manual. Order Number: 253668-060US, September 2016.

[30] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. RAID'14.

[31] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Lucky 13 Strikes Back. ASIA CCS'15.

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

[33] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. SEC'12.

[34] 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. ISCA'14.

[35] D. Kohlbrenner and H. Shacham. Trusted browsers for uncertain times. SEC'16, 2016.

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

[37] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. SP'15.

[38] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. SEC'15.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. SP'15.

[40] LKML. Page Colouring. goo.gl/7o101i.

[41] J. Longo, E. De Mulder, D. Page, and M. Tunstall. SoC It to EM: ElectroMagnetic Side-Channel Attacks on a Complex System-on-Chip. CHES'15.

[42] 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.

[43] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. ISCA'12.

[44] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. RAID'15.

[45] M. Miller and K. Johnson. Exploit Mitigation Improvements in Win 8. BH-US'12.

[46] Microsoft Security Bulletin MS16-092. https://technet.microsoft.com/library/security/MS16-092.

[47] A. Oikonomopoulos, C. Giuffrida, E. Athanasopoulos, and H. Bos. Poking Holes into Information Hiding. SEC'16.

[48] 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.

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

[50] M. Payer. HexPADS: A Platform to Detect "Stealth" Attacks. ESSoS'16.

[51] 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.

[52] SafeStack. http://clang.llvm.org/docs/SafeStack.html.

[53] Chromium issue 508166. https://goo.gl/KalbZx.

[54] F. J. Serna. The Info Leak Era on Software Exploitation. BH-US'12.

[55] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-space Randomization. CCS'04.

[56] ECMAScript Shared Memory. https://goo.gl/WXpasG.

[57] 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. SP'13.

[58] A. Sotirov. Heap Feng Shui in JavaScript. BH-EU'07.

[59] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating Cache-Based Timing Attacks with Instruction-Based Scheduling. ESORICS'13.

[60] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. CCS'15.

[61] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. SEC'14.

[62] 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.

[63] VMWare. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing.

[64] D. Weston and M. Miller. Windows 10 Mitigation Improvements. BH-US'16.

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

[66] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. SP'09.

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