The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications

Yossef Oren   Vasileios P. Kemerlis   Simha Sethumadhavan   Angelos D. Keromytis

Columbia University
Department of Computer Science
{yos, vpk, simha, angelos}@cs.columbia.edu

ABSTRACT

We present a micro-architectural side-channel attack that runs entirely in the browser. In contrast to previous work in this genre, our attack does not require the attacker to install software on the victim's machine; to facilitate the attack, the victim needs only to browse to an untrusted webpage that contains attacker-controlled content. This makes our attack model highly scalable, and extremely relevant and practical to today's Web, as most desktop browsers currently used to access the Internet are affected by such side channel threats. Our attack, which is an extension to the last-level cache attacks of Liu et al. [14], allows a remote adversary to recover information belonging to other processes, users, and even virtual machines running on the same physical host with the victim web browser.

We describe the fundamentals behind our attack, and evaluate its performance characteristics. In addition, we show how it can be used to compromise user privacy in a common setting, letting an attacker spy after a victim that uses private browsing. Defending against this side channel is possible, but the required countermeasures can exact an impractical cost on benign uses of the browser.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*information flow controls*; K.6 [**Management of Computing and Information Systems**]: Miscellaneous—*security*

General Terms

Languages, Measurement, Security

Keywords

side-channel attacks; cache-timing attacks; JavaScript-based cache attacks; covert channel; user tracking

1. INTRODUCTION

Side-channel analysis is a powerful cryptanalytic technique. It allows attackers to extract information hidden inside a device, by analyzing the physical signals (e.g., power, heat) that the device emits as it performs a secure computation [15]. Allegedly used by the intelligence community as early as in WWII, and first discussed in an academic context by Kocher in 1996 [13], side-channel analysis has been shown to be effective in a plethora of real-world systems, ranging from car immobilizers to high-security cryptographic coprocessors [6,20]. A particular kind of side-channel attacks that are relevant to personal computers are cache attacks, which exploit the use of cache memory as a shared resource between different processes to disclose information [9,19].

Even though the effectiveness of side-channel attacks is established without question, their application to practical settings is debatable, with the main limiting factor being the *attack model* they assume; excluding network-based timing attacks [4], most side-channel attacks require an attacker in "close proximity" to the victim. Cache attacks, in particular, assume that the attacker is capable of executing binary code on the victim's machine. While this assumption holds true for IaaS environments, like Amazon's cloud platform, where multiple parties may share a common physical machine, it is less relevant in other settings.

In this paper, we challenge this limiting assumption by presenting a successful cache attack that assumes a far more relaxed and practical attacker model. Specifically, in our model, the victim merely has to *access a website* owned by the attacker. Despite this minimal model, we show how the attacker can launch an attack in a practical time frame and extract meaningful information from the victim's machine. Keeping in tune with this computing setting, we choose not to focus on cryptographic key recovery, but rather on *tracking user behaviour*. The attacks described herein are highly practical: (a.) practical in the assumptions and limitations they cast upon the attacker, (b.) practical in the time they take to run, and (c.) practical in terms of the benefit they deliver to the attacker.

For our attack we assume that the victim is using a computer powered by a late-model Intel processor. In addition, we assume that the victim is accessing the web through a browser with comprehensive HTML5 support. As we show in Section 6.1, this covers the vast majority of personal computers connected to the Internet. The victim is coerced to view a webpage containing an attacker-controlled element, like an advertisement, while the attack code itself, which we describe in more detail in Section 3, executes a JavaScript-

based cache attack, which lets the attacker track accesses to the victim's last-level cache over time. Since this single cache is shared by all CPU cores, this access information can provide the attacker with a detailed knowledge regarding the user and system under attack.

Crafting a last-level cache attack that can be launched over the web using JavaScript is quite challenging; JavaScript code cannot load shared libraries or execute native code. More importantly, it is forced to make timing measurements using scripting language function calls instead of high-fidelity timing instructions. Despite these challenges, we successfully extended cache attacks to the web environment:

- We present a novel method for creating a *non-canonical eviction set* for the last-level cache. In contrast to the recent work by Liu et al. [14], our method does not require system support for large pages, and therefore, it can immediately be applied to a wider variety of systems. More importantly, we show that our method runs in a practical time frame.

- We demonstrate a last-level cache attack using JavaScript code only. We evaluate its performance using a covert channel method, both among different processes running on the same machine and between a VM client and its host. The nominal capacity of the JavaScript-based channel is in the order of hundreds of Kbit/s, comparable to that of native code approaches [14].

- We show how cache-based attacks can be used to *track the behaviour* of users. Specifically, we present a simple classifier-based attack that lets a malicious webpage spy on the user's browsing activity, detecting the use of common websites with an accuracy of over 80%. Remarkably, it is even possible to spy on the private browsing session of a completely different browser.

2. BACKGROUND AND RELATED WORK

2.1 Memory Hierarchy of Intel CPUs

Modern computer systems incorporate high-speed CPUs and a large amount of lower-speed RAM. To bridge the performance gap between these two components, they make use of *cache memory*: a type of memory that is smaller but faster than RAM (in terms of access time). Cache memory contains a subset of the RAM's contents recently accessed by the CPU, and is typically arranged in a *cache hierarchy*— series of progressively larger and slower memory elements are placed in various levels between the CPU and RAM.

Figure 1 shows the cache hierarchy of Intel Haswell CPUs, incorporating a small, fast level 1 (L1) cache, a slightly larger level 2 (L2) cache, and finally, a larger level 3 (L3) cache, which in turn is connected to RAM. Whenever the CPU wishes to access physical memory, the respective address is first searched for in the cache hierarchy, saving the lengthy round-trip to RAM. If the CPU requires an element that is not currently in the cache, an event known as a *cache miss*, one of the elements currently residing in the cache is *evicted* to make room for this new element. The decision of which element to evict in the event of a cache miss is made by a heuristic algorithm that has changed between processor generations (see Section 6.2).
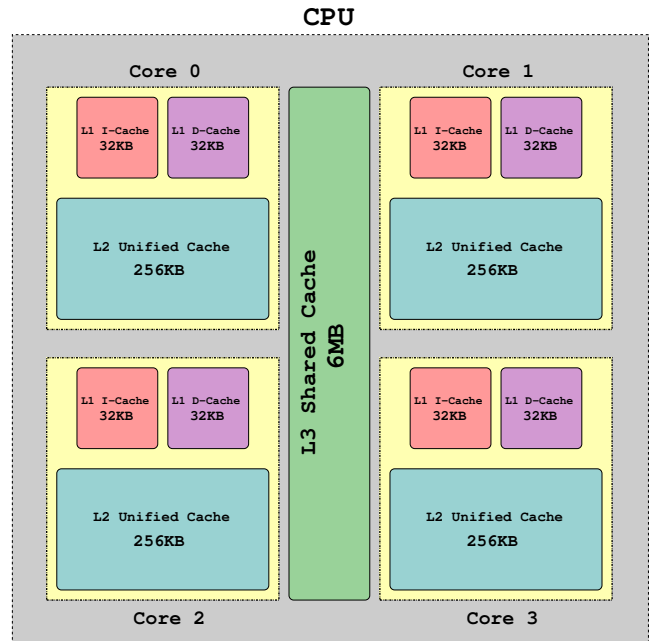

Figure 1: Cache memory hierarchy of Intel CPUs (based on Ivy Bridge Core i5-3470).

Intel's cache micro-architecture is *inclusive*: all elements in the L1 cache exist in the L2 and L3 caches. Conversely, if a memory element is evicted from the L3 cache, it is also immediately evicted from the L2 and L1 cache. It should be noted that the AMD cache micro-architecture is exclusive, and thus, the attacks described in this paper are not immediately applicable to that platform.

In this work, we focus on the L3 cache, commonly referred to as the last-level cache (LLC). The LLC is shared among all cores, threads, processes, and even virtual machines running on a certain CPU chip, regardless of protection rings or other isolation mechanisms. On Intel CPUs, the LLC is divided into several *slices*: each core of the CPU is directly connected to one of these cache slices, but can also access all other slices by using a ring bus interconnection.

Due to the relatively large size of the LLC, it is not efficient to search its entire contents whenever the CPU accesses the RAM. Instead, the LLC is further divided into *cache sets*, each covering a fixed subset of the physical memory space. Each of these cache sets contains several *cache lines*. For example, the Intel Core i7-4960HQ processor, belonging to the Haswell family, includes 8192 (2^{13}) cache sets, each of which is 12-way associative. This means that every cache set can hold 12 lines of 64 (2^6) bytes each, giving a total cache size of 8192x12x64=6MB. When the CPU needs to check whether a given physical address is present in the L3 cache, it calculates which cache set is responsible for this address, and then only checks the cache lines corresponding to this set. As a consequence, a cache miss event for a physical address will result in the eviction of only one of the relatively small amount of lines sharing its cache set, a fact we make great use of in our attack. The method by which 64-bit physical addresses are mapped into 12-bit or 13-bit cache set indices is undocumented and varies among processor generations, as we discuss in Section 6.2.

In the case of Sandy Bridge, this mapping was reverse-engineered by Hund et al. [10], where they showed that of the 64 physical address bits, bits 5 to 0 are ignored, bits 16 to 6 are taken directly as the lower 11 bits of the set index, and bits 63 to 17 are hashed to form the *slice index,* a 2-bit (in the case of quad-core) or 1-bit (in the case of dual-core) value assigning each cache set to a particular LLC slice.

In addition to the above, modern computers typically support virtual memory, restricting user processes from having direct access to the system's RAM. Instead, these processes are allocated virtual memory *pages.* The first time a page is accessed by an executing process, the Operating System (OS) dynamically associates the page with a *page frame* in RAM. The Memory Management Unit (MMU) is in charge of mapping the virtual memory accesses made by different processes to accesses in physical memory. The size of pages and page frames in most Intel processors is typically set to 4KB[1], and both pages and page frames are page-aligned (i.e., the starting address of each page is a multiple of the page size). This means that the lower 12 bits of any virtual address and its corresponding physical address are generally identical, another fact we use in our attack.

2.2 Cache Attacks

The cache attack is a well-known representative of the general class of micro-architectural side-channel attacks, which are defined by Aciiçmez [1] as attacks that "exploit deeper processor ingredients below the trust architecture boundary" to recover secrets from various secure systems. Cache attacks make use of the fact that—regardless of higher-level security mechanisms, like protection rings, virtual memory, hypervisors, and sandboxing—secure and insecure processes can interact through their shared use of the cache. This allows an attacker to craft a "spy" program that can make inferences about the internal state of a secure process. First identified by Hu [9], several results have shown how the cache side-channel can be used to recover AES keys [3, 19], RSA keys [21], or even allow one virtual machine to compromise another virtual machine running on the same host [24].

Our attack is modeled after the PRIME+PROBE method, which was first described by Osvik et al. [19] in the context of the L1 cache, and later extended by Liu et al. [14] to last-level caches on systems with large pages enabled. In this work, we further extend this attack to last-level caches in the more common case of 4KB-sized pages.

In general, the PRIME+PROBE attack follows a four-step pattern. In the first step, the attacker creates one or more *eviction sets.* An eviction set is a sequence of memory addresses that are all mapped by the CPU into the same cache set. The PRIME+PROBE attack also assumes that the victim code uses this cache set for its own code or data. In the second step, the attacker *primes* the cache set by accessing the eviction set in an appropriate way. This forces the eviction of the victim's data or instructions from the cache set and brings it to a known state. In the third step, the attacker triggers the victim process, or passively waits for it to execute. During this execution step, the victim may potentially utilise the cache and evict some of the attacker's elements from the cache set. In the fourth step, the attacker *probes* the cache set by accessing the eviction set again.

---

[1] 2MB and 1GB pages are also supported in newer CPUs.

A probe step with a low access latency suggests that the attacker's eviction set is still in the cache. Conversely, a higher access latency suggests that the victim's code made use of the cache set and evicted some of the attacker's memory elements. The attacker thus learns about the victim's internal state. The actual timing measurement is carried out by using the (unprivileged) instruction rdtsc, which provides a high-fidelity measurement of the CPU cycle count. Iterating over the eviction set in the probing phase forces the cache set yet again into an attacker-controlled state, thus preparing for the next round of measurements.

3. PRIME+PROBE IN JAVASCRIPT

JavaScript is a dynamically typed, object-based scripting language with runtime evaluation that powers the client side of the modern web. Websites deliver JavaScript programs to the browser, which in turn are (typically) compiled and optimized using a Just-In-Time (JIT) mechanism.

The core functionality of the JavaScript language is defined in the standard ECMA-262 [5]. The language standard is complemented by a large set of application programming interfaces (APIs) defined by the World Wide Web Consortium [27], which make the language practical for developing web content. The JavaScript API set is constantly evolving, and browser vendors add support for new APIs over time according to their own development schedules. Two specific APIs that are of use to us in this work are the Typed Array Specification [7], which allows efficient access to unstructured binary data, and the High Resolution Time API [16], which provides JavaScript with submillisecond time measurements. As we show in Section 6.1, the vast majority of Web browsers that are in use today support both APIs.

By default, browsers will automatically execute every JavaScript program delivered to them by a webpage. To limit the potential damage of this property, JavaScript code runs in a *sandboxed* environment—code delivered via JavaScript has severely restricted access to the system. For example, it cannot open files, even for reading, without the permission of the user. Also, it cannot execute native code or load native code libraries. Most importantly, JavaScript code has no notion of pointers. Thus, it is impossible to determine the virtual address of a JavaScript variable.

Methodology. The four steps involved in a successful PRIME+PROBE attack (see Section 2.2) are the following: (a.) creating an eviction set for one or more relevant cache sets; (b.) priming the cache set; (c.) triggering the victim operation; (d.) probing the cache set again. Each of these steps must be implemented in JavaScript and overcome the unique limitations of the web environment.

3.1 Creating an Eviction Set

In the first step of a PRIME+PROBE attack the attacker creates an eviction set for a cache set whose activity should be tracked [14]. This eviction set consists of a sequence of variables (data) that are all mapped by the CPU into a cache set that is also used by the victim process. We first show how we create an eviction set for an arbitrary cache set, and later address the problem of finding which cache set is particularly interesting from the attacker's perspective.

Set assignments for variables in the LLC are made by reference to their physical memory addresses, which are not available to unprivileged processes.[2] Liu et al. [14] partially circumvented this problem by assuming that the system is operating in large page (2MB) mode, in which the lower 21 bits of the physical and virtual addresses are identical, and by the additional use of an iterative algorithm to resolve the unknown upper (slice) bits of the cache set index.

In the attack model we consider, the system is not running in large page mode, but rather in the more common 4KB page mode, where only the lower 12 bits of the physical and virtual addresses are identical. To our further difficulty, JavaScript has no notion of pointers, so even the virtual addresses of our own variables are unknown to us. This makes it very difficult to provide a deterministic mapping of memory address to cache sets. Instead, we use the heuristic algorithm described below.

We assume a victim system with $s = 8192$ cache sets, each with $l = 12$-way associativity. Hund et al. [10] suggest accessing a contiguous 8MB physical memory *eviction buffer* for completely invalidating all cache sets in the L3 cache. We could not allocate such an eviction buffer in user-mode; in fact, the aforementioned work was assisted by a kernel-mode driver. Instead, we allocated an 8MB byte array in virtual memory using JavaScript (which was assigned by the OS into an arbitrary and non-contiguous set of 4KB physical memory pages), and measured the system-wide effects of iterating over this buffer.

We discovered that access latencies to unrelated variables in memory increased by a noticeable amount when they were accessed immediately after iterating through this eviction buffer. We also discovered that the slowdown effect persisted even if we did not access the entire buffer, but rather accessed it in offsets of 1 per every 64 bytes (this behaviour was recently extended into a full covert channel [17]). However, it is not immediately clear how to map each of the 131K offsets we accessed inside this eviction buffer into each of the 8192 possible cache sets, since we know neither the physical memory locations of the various pages of our buffer, nor the mapping function used by our specific micro-architecture to assign cache sets to physical memory addresses.

A naive approach to solving this problem would be to fix an arbitrary "victim" address in memory, and then find by brute force which of the 8MB/64B=131K possible addresses in the eviction buffer are in the same cache set as this victim address, and as a consequence, within the same cache set as each other. To carry out the brute-force search, the attacker iterates over all subsets of size $l = 12$ of all possible addresses. For each subset, the attacker checks whether the subset serves as the eviction set for the victim address by checking whether accessing this subset slows down subsequent accesses to the victim variable. By repeating this process 8192 times, each time with a different victim address, the attacker can identify 12 addresses that reside in each cache set and thereby create the eviction set data structure.

Optimization #1. An immediate application of this heuristic would take an impractically long time to run. One simple optimization is to start with a subset containing all 131K possible offsets, then gradually attempt to shrink it

---

Algorithm 1 Profiling a Cache Set.

Let S be the set of currently unmapped page-aligned addresses, and address x be an arbitrary page-aligned address in memory.

1. Repeat k times:

   (a) Iteratively access all members of S.

   (b) Measure t_1, the time it takes to access x.

   (c) Select a random page s from S and remove it.

   (d) Iteratively access all members of $S \backslash s$.

   (e) Measure t_2, the time it takes to access x.

   (f) If removing s caused the memory access to speed up considerably (i.e., $t_1 - t_2 > thres$), then this address is part of the same set as x. Place it back into S.

   (g) If removing s did not cause memory access to speed up considerably, then s is not part of the same set as x.

2. If $|S| = 12$, return S. Otherwise report failure.

---

by removing random elements and checking that the access latency to the victim address stays high. The final data structure should be of size 12 and contain only the entries sharing a cache set with the victim variable. Even this optimization, however, is too slow for practical use. Fortunately, the page frame size of the Intel MMU, as described in Section 2.1, could be used to our great advantage. Since virtual memory is page aligned, the lower 12 bits of each virtual memory address are identical to the lower 12 bits of each physical memory address. According to Hund et al., 6 of these 12 bits are used to uniquely determine the cache set index [10]. Thus, a particular offset in our eviction buffer can only share a cache set with an offset whose bits 12 to 6 are identical to its own. There are only 8K such offsets in the 8MB eviction buffer, speeding up performance considerably.

Optimization #2. Another optimization comes from the fact that if physical addresses P_1 and P_2 share a cache set, then for any value of Δ, physical addresses $P_1 \oplus \Delta$ and $P_2 \oplus \Delta$ also share a (possibly different) cache set. Since each 4KB block of virtual memory maps to a 4KB block in physical memory, this implies that discovering a single cache set can immediately teach us about 63 additional cache sets. Joined with the discovery that JavaScript allocates large data buffers along page frame boundaries, this finally leads to the greedy approach outlined in Algorithm 1.

By running Algorithm 1 multiple times, we gradually create eviction sets covering most of the cache, except for those parts that are accessed by the JavaScript runtime itself. We note that, in contrast to the eviction sets created by the algorithm of Liu et al. [14], our eviction set is *non-canonical*: JavaScript has no notion of pointers, and hence, we cannot identify which of the CPU's cache sets correspond to any particular eviction set we discover. Furthermore, running the algorithm multiple times on the same system will result in a different mapping each time. This property stems from the use of traditional 4KB pages instead of large 2MB pages, and will hold even if the eviction sets are created using native code and not JavaScript.

CPU Model	Micro-arch.	LLC Size	Cache Assoc.
Core i5-2520M	Sandy Bridge	3MB	12-way
Core i7-2667M	Sandy Bridge	4MB	16-way
Core i5-3427U	Ivy Bridge	3MB	12-way
Core i7-3667U	Ivy Bridge	4MB	16-way
Core i7-4960HQ	Haswell	6MB	12-way
Core i7-5557U	Broadwell	4MB	16-way

Table 1: CPUs used to evaluate the performance of the profiling cache set technique (Algorithm 1).
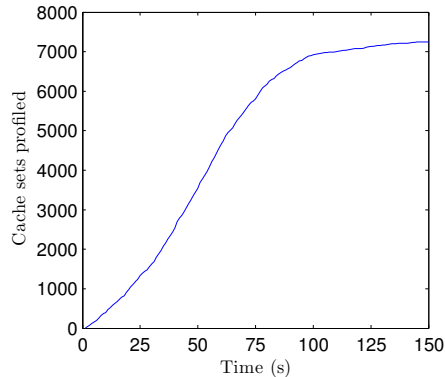

Figure 2: Cumulative performance of the profiling algorithm (Haswell i7-4960HQ).

```
 1  // Invalidate the cache set
 2  var currentEntry = startAddress;
 3  do {
 4      currentEntry =
 5          probeView.getUint32(currentEntry);
 6  } while (currentEntry != startAddress);
 7
 8  // Measure access time
 9  var startTime = window.performance.now();
10  currentEntry   =
11          primeView.getUint32(variableToAccess);
12  var endTime    = window.performance.now();
```

Evaluation. We implemented Algorithm 1 in JavaScript and evaluated it on Intel machines using CPUs from the Sandy Bridge, Ivy Bridge, and Haswell families, running the latest versions of Safari and Firefox on Mac OS X v10.10 and Ubuntu 14.04 LTS, respectively. The setting of the evaluation environment represented a typical web browsing session, with common applications, such as an email client, calendar, and even a music player running in the background. The attack code was loaded from an untrusted website into one tab of a multi-tabbed browsing session. Attacks were performed when the tab was the foreground tab, when the browser process was in the foreground but a different tab was the foreground tab, and when the web browser process was running in the background. The specifications of the CPUs we evaluated are listed in Table 1; the systems were not configured to use large pages, but instead were running with the default 4KB page size. The code snippet shown above illustrates lines 1.d and 1.e of Algorithm 1, and demonstrates how we iterate over the eviction set and measure latencies using JavaScript. The algorithm requires some additional steps to run under Internet Explorer (IE) and Chrome, which we describe in Section 6.1.
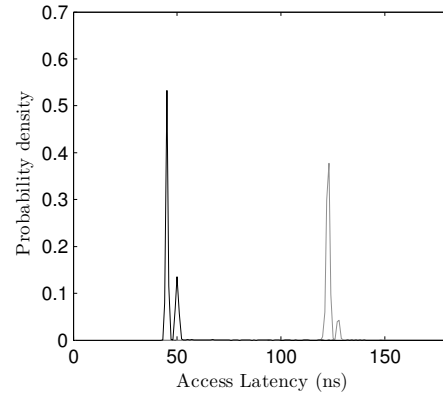

Figure 3: Probability distribution of access times for a flushed vs. unflushed variable (Haswell i7-4960HQ).

Figure 2 shows the performance of our profiling algorithm implemented in JavaScript, as evaluated on an Intel i7-4960-HQ running Firefox 35 for Mac OS X 10.10. We were pleased to find that our approach was able to map more than 25% of the cache in under 30 seconds of operation, and more than 50% of the cache after 1 minute. On systems with smaller cache sizes, such as the Sandy Bridge i5-2520M, profiling was even faster, taking less than 10 seconds to profile 50% of the cache. The profiling technique itself is simple to parallelize, since most of its execution time is spent on data structure maintenance and only a small part is spent on the actual invalidate-and-measure portion; multiple worker threads can prepare several data structures to be measured in parallel, with the final measurement step being carried out by a single master thread.[3] Finally, note that the entire algorithm is implemented in ∼ 500 lines of JavaScript code.

To verify that Algorithm 1 is capable of identifying cache sets, we designed an experiment that compares the access latencies for a flushed and an unflushed variable. Figure 3 shows two probability distribution functions comparing the time required to access a variable that has recently been flushed from the cache by accessing the eviction set (gray line), with the time required to access a variable that currently resides in the L3 cache (black line). The timing measurements were carried out using JavaScript's high resolution timer, and thus include the additional delay imposed by the JavaScript runtime. It is clear that the two distributions are distinguishable, confirming the correct operation of our profiling method. We further discuss the effects of background noise on this algorithm in Section 6.3.

3.2 Priming and Probing

Once the attacker identifies an eviction set consisting of 12 entries that share the same cache set, the next goal is to replace all entries in the cache of the CPU with the elements of this eviction set. In the case of the probe step, the attacker has the added goal of precisely measuring the time required to perform this operation.

---

[3] The current revision of the JavaScript specification does *not* allow multiple worker threads to share a single buffer in memory. An updated specification, which supports this functionality, is currently undergoing a ratification process and is expected to be made official by the end of 2015.

Algorithm 2 Identifying Interesting Cache Regions.

Let S_i be the data structure matched to eviction set i.

- For each set i:

  1. Iteratively access all members of S_i to prime the cache set.
  2. Measure the time it takes to iteratively access all members of S_i.
  3. Perform an interesting operation.
  4. Measure once more the time it takes to iteratively access all members of S_i.
  5. If performing the interesting operation caused the access time to slow down considerably, then this operation is associated with cache set i.

Modern high-performance CPUs are highly out-of-order, meaning that instructions are not executed by their order in the program, but rather by the availability of input data. To ensure the in-order execution of critical code parts, Intel provides "memory barrier" functionality through various instructions, one of which is the (unprivileged) instruction `mfence`. As JavaScript code is not capable of running it, we had to artificially make sure that the entire eviction set was actually accessed before the timing measurement code was run. We did so by accessing the eviction set in the form of a linked list (as was also suggested by Osvik et al. [19]), and making the timing measurement code artificially dependent on the eviction set iteration code. The CPU also has a *stride prefetching* feature, which attempts to anticipate future memory accesses based on regular patterns in past memory accesses. To avoid the effect of this feature we randomly permute the order of elements in the eviction set. We also access the eviction set in alternating directions to avoid an excessive amount of cache misses (see Section 6.2).

A final challenge is the issue of timing jitter. In contrast to native code PRIME+PROBE attacks, which use an assembler instruction to measure time, our code uses an interpreted language API call (`Window.Performance.now()`), which is far more likely to be impacted by measurement jitter. In our experiments we discovered that while some calls to `Window.Performance.now()` indeed took much longer to execute than expected (e.g., milliseconds instead of nanoseconds), the proportion of these jittered events was very small and inconsequential.

3.3 Identifying Interesting Cache Regions

The eviction set allows the attacker to monitor the activity of arbitrary cache sets. Since the eviction set we receive is non-canonical, the attacker must correlate the profiled cache sets to data or code locations belonging to the victim. This learning/classification problem was addressed earlier by Zhang et al. [29] and by Liu et al. [14], where various machine learning methods were used to derive meaning from the output of cache latency measurements.

To effectively carry out the learning step, the attacker needs to induce the victim to perform an action, and then examine which cache sets were touched by this action, as formally defined in Algorithm 2.

Finding a function to perform the step (3) of Algorithm 2 was actually quite challenging, due to the limited permissions granted to JavaScript code. This can be contrasted with the ability of Gorka et al. [2] to trigger kernel code by invoking `sysenter`. To carry out this step, we had to survey the JavaScript runtime and discover function calls which may trigger interesting behaviour, such as file access, network access, memory allocation, etc. We were also interested in functions that take a relatively short time to run and leave no background "trails", such as garbage collection, which would impact our measurement in step (4). Several such functions were discovered in a different context by Ho et al. [8]. Since our code will always detect activity caused by the JavaScript runtime, the high performance timer code, and other components of the web browser that are running regardless of the call being executed, we actually call two similar functions and examine the *difference* between the activity profile of the two evaluations to identify relevant cache sets. Another approach would be to induce the user to perform an interesting behaviour (such as pressing a key on her keyboard). The learning process in this case might be structured (the attacker knows exactly when the victim operation was executed), or unstructured (the attacker can only assume that relatively busy periods of system activity are due to victim operations). We examine both of these approaches in the attack we present in Section 5.

4. NON-ADVERSARIAL SETTING

In this section, we evaluate the capabilities of JavaScript-based cache probing in a non-adversarial context. By selecting a group of cache sets and repeatedly measuring their access latencies over time, the attacker is provided with a very detailed picture of the real-time activity of the cache. We call the visual representation of this image a *memorygram*, since it looks quite similar to an audio spectrogram.

A sample memorygram, collected over an idle period of 400ms, is presented in Figure 4. The X axis corresponds to time, while the Y axis corresponds to different cache sets. The sample shown has a temporal resolution of 250μs and monitors a total of 128 cache sets (note that the highest temporal resolution we were able to achieve while monitoring 128 cache sets in parallel was ∼ 5μs). The intensity of each pixel corresponds to the access latency of a particular cache set at this particular time, with black representing a low latency, suggesting no other process accessed this cache set between the previous measurement and this one, and white representing a higher latency, suggesting that the attacker's data was evicted from the cache between this measurement and the previous one.

Observing this memorygram can provide several insights. First, it is clear to see that despite the use of JavaScript timers instead of machine language instructions, measurement jitter is quite low and that active and inactive sets are clearly differentiated. It is also easy to notice several vertical line segments in the memorygram, indicating multiple adjacent cache sets that were all active during the same time period. Since consecutive cache sets (within the same page frame) correspond to consecutive addresses in physical memory, we believe this signal indicates the execution of a function call that spans more than 64 bytes of instructions. Several smaller groups of cache sets are accessed together; we theorise that such groups correspond to variable accesses.
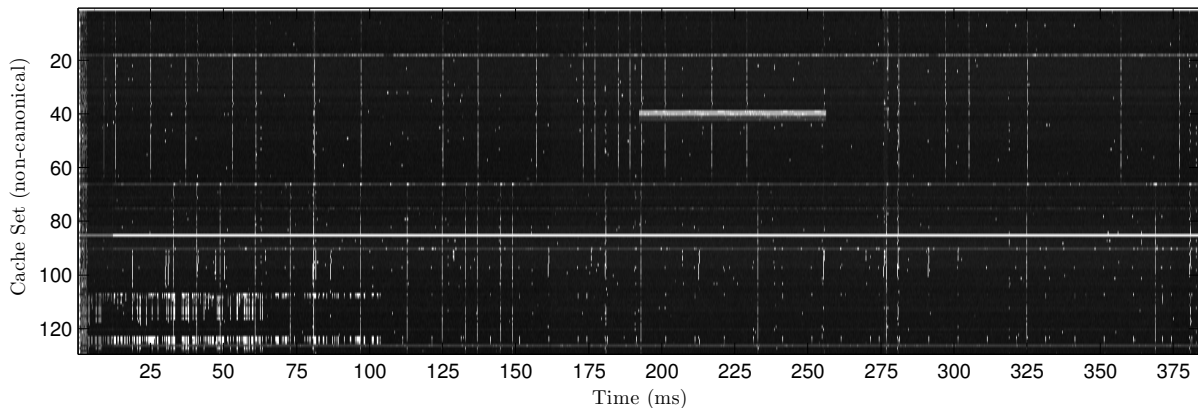
Figure 4: Sample memorygram collected over an idle period of 400ms. The X axis corresponds to time, while the Y axis corresponds to different cache sets. The sample shown has a temporal resolution of 250µs and monitors a total of 128 cache sets. The intensity of each pixel illustrates the access latency of the particular cache set, with black representing low latency and white representing a higher latency.

Finally, the white horizontal line indicates a variable that was constantly accessed during our measurements (e.g., a variable that belongs to the measurement code or the JavaScript runtime).

4.1 Covert Channel Bandwidth Estimation

Liu et al. [14] and Maurice et al. [17] demonstrated that last-level cache access patterns can be used to construct a high-bandwidth covert channel between virtual machines co-resident on the same physical host, and exfiltrate sensitive information. We used such a construction to estimate the measurement bandwidth of our attack. The design of our covert channel system was influenced by two requirements. First, we wanted the transmitter part to be as simple as possible, and in particular we did not want it to carry out the eviction set algorithm of Section 3.1. Second, since the receiver's eviction set is non-canonical, it should be as simple as possible for the receiver to search for the sets onto which the transmitter was modulating its signal.

To satisfy these requirements, our transmitter code simply allocates a 4KB array in its own memory and continuously modulates the collected data into the pattern of memory accesses to this array. There are 64 cache sets covered by this array, allowing the transmission of 64 bits per time period. To make sure the memory accesses are easily located by the receiver, the same access pattern is repeated in several additional copies of the array. Thus, a considerable percentage of the cache is actually exercised by the transmitter.

The receiver code profiles the system's RAM, and then searches for one of the page frames containing the data modulated by the transmitter. To evaluate the bandwidth of this covert channel, we wrote a simple program that iterates over memory in a predetermined pattern. Next, we search for this memory access pattern using a JavaScript cache attack, and measure the maximum sampling frequency at which the JavaScript code could be run. We first evaluated our code when both the transmitter and receiver were running on a normal host. Next, we repeated our measurements when the receiver was running inside a virtual machine (Firefox 34 running on Ubuntu 14.01 inside VMware Fusion 7.1.0). The nominal bandwidth of our covert channel was measured to be 320kbps, a figure which compares well with the 1.2Mbps

achieved by the native code, cross-VM covert channel of Liu et al. [14]. When the receiver code was not running directly on the host, but rather on a virtual machine, the peak bandwidth of our covert channel was ∼8kbps.

5. TRACKING USER BEHAVIOR

The majority of the related work in this field assumes that the attacker and the victim share a machine inside the data center of a cloud-provider. Such a machine is not typically configured to accept interactive input, and hence, previous work focused on the recovery of cryptographic keys or other secret state elements, such as random number generator states [30]. In this work, we chose to examine how cache attacks can be used to track the interactive behaviour of the user, a threat which is more relevant to the attack model we consider. We note that Ristenpart et al. [24] have already attempted to track keystroke timing events using coarse-grained measurements of system load on the L1 cache.

5.1 Detecting Hardware Events

Our first case study investigated whether our cache attack can detect hardware events generated by the system. We chose to focus on mouse and network activity because the OS code that handles them is non-negligible. In addition, they are also easily triggered by content running within the restricted JavaScript sandbox, allowing our attack to have a training phase.

Design. The structure of both attacks is similar. First, the profiling phase is carried out, allowing the attacker to probe individual cache sets using JavaScript. Next, during a training phase, the activity to be detected (e.g., network activity, mouse activity) is triggered, and the cache state is sampled multiple times with a very high temporal resolution. While the network activity was triggered directly by the measurement script (by executing a network request), we simply waved the mouse around over the webpage during the training period[4].

---

[4]In a full attack, the user can be enticed to move the mouse by having her play a game or fill out a form.

By comparing the cache state during the idle and active periods of the training phase, the attacker learns which cache sets are uniquely active during the relevant activity and trains a classifier on these cache sets. Finally, during the classification phase, the attacker monitors the interesting cache sets over time to learn about user activity.

We used a basic unstructured training process, assuming that the most intensive operation performed by the system during the training phase would be the one being measured. To take advantage of this property, we calculated the Hamming weight of each measurement over time (equivalent to the count of cache sets which are active during a certain time period), then applied a k-means clustering of these Hamming weights to divide the measurements into several clusters. Finally, we calculated the mean access latency of each cache set in every cluster, creating a *centroid* for each cluster. To classify an unknown measurement vector, we measured the Euclidean distance between this vector and each of these centroids, classifying it to the closest one.

Evaluation. We evaluated our hardware event detection strategy on an Intel Core i7-4960HQ processor, belonging to the Haswell family, running Safari 8.0.6 for Mac OS 10.10.3. We generated network traffic using the command-line tool `wget` and mouse activity by using the computer's internal trackpad to move the mouse cursor outside of the browser window. To provide ground truth for the network activity scenario, we concurrently measured the traffic on the system using `tcpdump`, and then mapped the `tcpdump` timestamps to the times detected by our classifier. To provide ground truth for the mouse activity scenario, we wrote a webpage that timestamps and logs all mouse events, then opened this webpage using a different browser (Chrome 43) and moved the mouse over this browser window. The memorygrams we collected for both experiments spanned 512 different cache sets and had a sampling rate of 500 Hz.

Our results indicate that it is possible to reliably detect mouse and network activity. The measurement rate of our network classifier did not allow us to count individual packets, but rather monitor periods of network (in)activity. Our detector was able to correctly detect 58% of these active periods, with a false positive rate of 2.86%. The mouse detection code actually logged *more* events than the ground truth collection code. We attribute this to the fact that the Chrome browser (or the OS) throttles mouse events at a rate of ∼60Hz. Yet, 85% of our mouse detection events were followed by a ground truth event in less than 10ms. The false positive rate was 3.86%, but most of the false positives were immediately followed by a series of true positives. This suggests that our classifier was also firing on other mouse-related events, such as "mouse down" or simply touches on the trackpad. Note that the mouse activity detector did not detect network activity (or vice versa).

Interestingly, we discovered that our measurements were affected by the ambient light sensor of the victim machine. Ambient light sensors are always-on sensors that are installed on high-end laptops, like MacBooks, Dell Latitude, Sony Vaio, and HP EliteBooks. They are enabled by default, and allow the OS to dynamically adjust the brightness of the computer screen to accommodate different lighting conditions. During our experiments we discovered that waving our hand in front of the laptop generated a noticeable burst of hardware events. This could be either the result of hard-
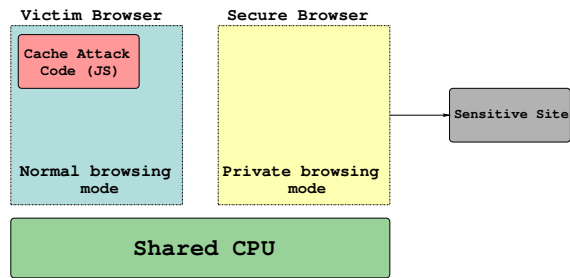

Figure 5: End-to-end attack scenario.

ware interrupts generated by the ambient light sensor itself, or hardware interrupts generated by the display panel, as it automatically adjusts its brightness. This side-channel leakage means that cache-based attacks can detect the *presence of a user* in front of the computer, an item of information which is highly desirable to advertisers.

5.2 End-to-End Privacy Attacks

5.2.1 Motivation

Modern browsers implement a private or incognito mode, which allows users to carry out sensitive online activities. When private browsing mode is enabled, the web browser does not disclose or collect any cookies, and disables web cache entries or other forms of local data storage. One browser executable that is considered extremely secure is the Tor Browser: a specially-configured browser bundle, built around the Firefox codebase, which is designed to block most privacy-sensitive APIs and connect to the Internet only through the Tor network. Since private browsing sessions disable certain network functionality, and do not retain the login credentials of the current user, they are cumbersome for general-purpose use. Instead, users typically run concurrently standard browsing sessions and private browsing sessions, side-by-side, on the same computer, either as two open windows belonging to the same browser process, or as two independent browser processes.

We assume that one of the websites opened during the standard browsing session is capable of performing our JavaScript cache attack (either by malicious design, or incidentally via a malicious banner ad or other affiliate content item). As Figure 5 illustrates, we show how an attacker can detect which websites are being loaded in the victim's private browsing session, thus compromising her privacy.

5.2.2 Experimental Setup

Our measurements were carried out on an Intel Core i7-2667M laptop, running Mac OS X 10.10.3. The attack code was executed on a standard browsing session, running on the latest version of Firefox (37.0.2), while the private browsing session ran on both the latest version of Safari (8.0.6) and the Tor Browser Bundle (4.5.1). The system was connected to the WiFi network of Columbia University, and had all non-essential background tasks stopped. To increase our measurement bandwidth, we chose to filter all hardware-related events. We began our attack with a simple training phase, in which the attacker measured the cache sets that were idle when the user was touching the trackpad, but not moving his finger.
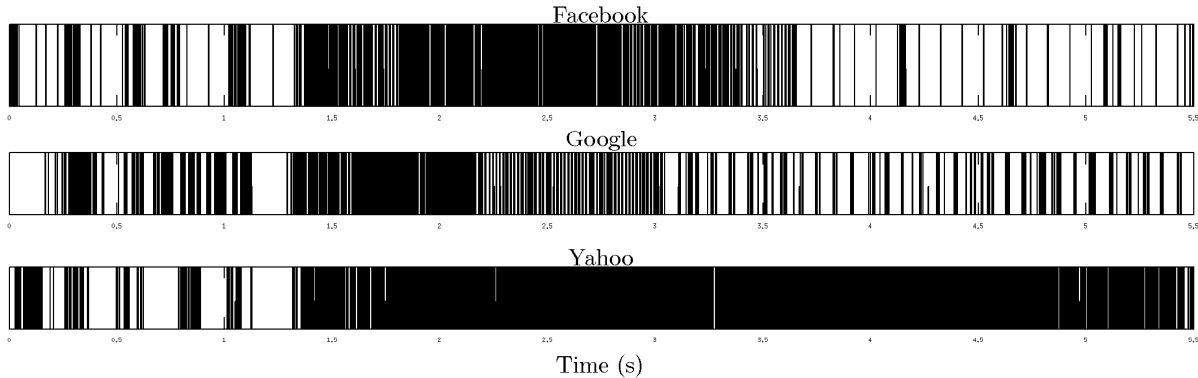
Figure 6: Memorygrams for three popular websites (Facebook, Google, Yahoo).

In each experiment, we opened the private-mode browsing window, typed the URL of a website to the address bar, and allowed the website to load completely. During this operation, our attack code collected memorygrams that represent cache activity. The memorygrams had a temporal resolution of 2ms, and a duration of 10 seconds for Safari private browsing and 50 seconds for the higher-latency Tor Browser. We collected a total of 90 memorygrams for 8 out of the top 10 sites on the web (according to Alexa ranking; May 2015). To further reduce our processing load, we only saved the mean activity of the cache sets over time, resulting in a 5000-element vector for each Safari measurement and a 25000-element vector for each Tor measurement. A representative set of the Safari memorygrams is depicted in Figure 6 (note that the memorygrams shown in the figure were manually aligned for readability; our attack code does not perform this alignment step).

Next follows the classification step, which is extremely simple. We calculated the mean absolute value of the Fourier transforms for each website's memorygrams (discarding the DC component), computed the absolute value of the Fourier transform for the current memorygram, and then output the label of the closest website according to the ℓ^2 distance.

We performed no other preprocessing, alignment, or modification to the data. In each experiment, we trained the classifier on all traces but one, and recorded the label output by the classifier for the missing trace. We expected that multiple memorygrams would be difficult to align, both since the attacker does not know the precise time when browsing begins, and since network latencies are unknown and may change between measurements.

We chose the Fourier transform method, as it is not affected by time shifting and because of its resistance to background measurement noise—as we discuss in Section 6.3, our primary sources of noise were timing jitter and spurious cache activity due to competing processes. Both sources manifested as high-frequency additive noise in our memorygrams, while most of the page rendering activity was centered in the low frequency ranges. We thus limit our detector to the low-pass components of the FFT output.

5.2.3 Results

Table 2 (Safari) and Table 3 (Tor Browser) show the confusion matrices of our classifiers. The overall accuracy was 82.1% for Safari and 88.6% for Tor.

Classifier Output→, Ground Truth↓	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Amazon (1)	.8	-	-	-	-	-	-	.2
Baidu (2)	.2	.8	-	-	-	-	-	-
Facebook (3)	-	-	.5	-	-	.5	-	-
Google (4)	-	-	-	1	-	-	-	-
Twitter (5)	-	-	-	-	1	-	-	-
Wikipedia (6)	-	-	.2	-	-	.8	-	-
Yahoo (7)	-	-	-	-	-	-	1	-
Youtube (8)	-	-	-	-	.4	-	-	.6

Table 2: Confusion matrix for FFT-based classifier (Safari Private Browsing).

The longer network round-trip times introduced by the Tor network did not diminish the performance of our classifier, nor did the added load of background activities, which unavoidably occurred during the 50 seconds of each measurement. The classifier was the least successful in telling apart the Facebook and Wikipedia memorygrams. We theorize that this is due to the fact that both websites load a minimal website with a blinking cursor that generates the distinct 2 Hz pulse shown in Figure 6. The accuracy of the detector can certainly be improved with more advanced classification heuristics (e.g., timing the keystrokes of the URL as it is entered, characterizing and filtering out frequencies with switching noise).

Our evaluation was limited to a *closed-world* model of the Internet, in which only a small set of websites was considered, and where template creation was performed based on traces from the victim's own machine. It is possible to justify this model for our specific attacker, who can easily carry out profiling on the victim's machine by instructing it to load known pages via JavaScript while recording memorygrams. Nevertheless, it would still be interesting to scale up the evaluation to an *open-world* model, where many thousands of websites are considered, and where the templates are created in a different time and place than the victim's current browsing session [11].

Classifier Output→, Ground Truth↓	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Amazon (1)	1	-	-	-	-	-	-	-
Baidu (2)	-	1	-	-	-	-	-	-
Facebook (3)	-	.2	.8	-	-	-	-	-
Google (4)	-	-	-	1	-	-	-	-
Twitter (5)	-	-	-	.17	.83	-	-	-
Wikipedia (6)	-	-	.33	-	.17	.5	-	-
Yahoo (7)	-	-	-	-	-	-	1	-
Youtube (8)	-	-	-	-	.4	-	-	1

Table 3: Confusion matrix for FFT-based classifier (Tor Browser).

Brand	Hi-Res. Time Support	Typed Arrays Support	Worldwide Prevalence
Internet Explorer	v10	v11	11.77%
Safari	v8	v6	1.86%
Chrome	v20	v7	50.53%
Firefox	v15	v4	17.67%
Opera	v15	v12.1	1.2%
Total	–	–	83.03%

Table 4: Affected desktop browsers: minimal version and prevalence [26].

6. DISCUSSION

6.1 Prevalence of Affected Systems

Our attack requires a personal computer powered by an Intel CPU based on the Sandy Bridge, Ivy Bridge, Haswell or Broadwell micro-architecture. According to data from IDC, more than 80% of all PCs sold after 2011 satisfy this requirement. We furthermore assume that the user is using a web browser that supports the HTML5 High Resolution Time API and the Typed Arrays specification. Table 4 notes the earliest version at which these APIs are supported for each common browser, as well as the proportion of global Internet traffic coming from such browser versions, according to StatCounter measurements (January 2015) [26]. As the table shows, more than 83% of desktop browsers in use today are affected by the attack we describe.

The effectiveness of our attack depends on being able to perform precise measurements using the JavaScript High Resolution Time API. While the W3C recommendation of this API [16] specifies that the a high-resolution timestamp should be "a number of milliseconds accurate to a thousandth of a millisecond", the maximum resolution of this value is not specified, and indeed varies between browser versions and OSes. During our tests, we discovered that the actual resolution of this timestamp for Safari on Mac OS X was on the order of nanoseconds, while IE for Windows had a 0.8μs resolution. Chrome, on the other hand, offered a uniform resolution of 1μs on all OSes we tested.

Since the timing difference between a single cache hit and a cache miss is on the order of 50ns (see Figure 3), the profiling and measurement algorithms need to be slightly modified to support systems with coarser-grained timing resolution.
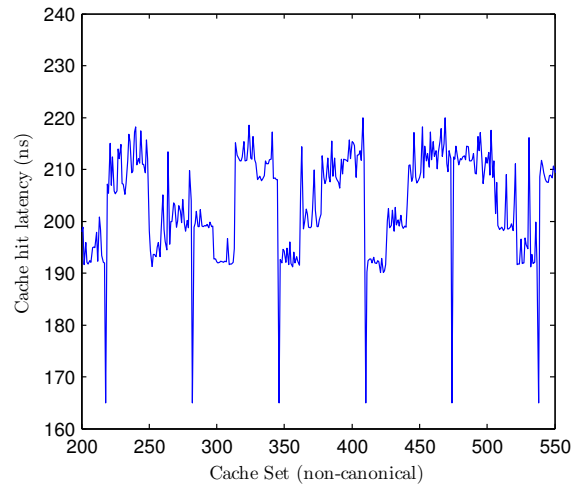

Figure 7: L3 cache hit times show a 3-level graduation (Haswell i7-4960HQ).

In the profiling stage, instead of measuring a single cache miss, we repeat the memory access cycle multiple times to amplify the time difference. We have used this observation to successfully perform cache profiling on versions of the Chrome browser whose timing resolution was limited[5]. For the measurement stage, we cannot amplify a single cache miss, but we can take advantage of the fact that code accesses typically invalidate multiple consecutive cache sets from the same page frame. As long as at least 20 out of the 64 cache sets, in a single page frame, register a cache miss, our attack is successful even with μs time resolution.

The attack we propose can also be applied to mobile devices, such as smartphones and tablets. It should be noted that the Android Browser supports High Resolution Time and Typed Arrays starting from version 4.4, but at the time of writing the most recent version of iOS Safari (8.1) did not support the High Resolution Time API.

6.2 Micro-architecture Insights

Despite the fact that our attack was implemented in a restricted, high-level language, it provides a glimpse into extremely low-level elements of the victim's machine. As a consequence, it is affected by the minute design choices made by Intel CPU architects. As stated by Aciiçmez [1], two concepts can affect the functional behavior of a cache: the *mapping strategy* and the *replacement policy*. The former determines which memory locations are mapped to each set in the cache, while the latter determines how the cache set will be modified after a cache miss.

We noticed different behaviour in the mapping strategy of the systems we surveyed, especially in the choice of the *slice index* of each memory address. In the processors we tested, the sets of the LLC are divided into slices, with each cache slice located in hardware with close proximity to one of the CPU's cores. All of the slices are interconnected via a ring buffer, allowing all cores to access cache entries in all slices.

---

[5]It should be noted that Chrome has an additional feature called Portable Native Client (PNaCl), which offers direct access to the native `clock_gettime()` API.

Cache sets are thus indexed first using the slice index, and next with the set index within the respective slice.

While the work of Hund et al. [10] showed that on Sandy Bridge CPUs the slice index is only a function of high-order bits of the physical address, Liu et al. [14] suggested that lower-order bits are also considered by newer microarchitectures. We confirmed this by measuring the *cache hit* of each of the cache sets we were able to profile on a quad-core Haswell processor. In such a system there are three possible times for an L3 cache hit. L3 cache entries located in a slice associated with the current core are the fastest to access. Hits on cache entries located in the two slices which are a single core's distance from the current core should be slightly slower, since the entry has to travel across a single hop on the ring buffer. Finally, hits on cache entries located in the slice which is two cores away from the current core should be the slowest to access, since the entries travel across two hops on the ring buffer. If lower-order address bits are used in the selection of the cache slice, we would expect to see a variation in the cache hit times for addresses within the same physical memory page. Figure 7 shows that this behaviour was indeed observed on a Haswell-generation CPU, confirming the suggestion of Liu et al.

The timing difference between the worst-case cache hit (which has to travel across two hops on the ring buffer) and a cache miss is still enough for Algorithm 1 to operate without modifications. However, an attacker can use this insight concerning LLC slices to his operative advantage. For example, two processes running on the same system can use this measurement to discover whether they are running on the same core or not, by comparing cache hit timings for the same cache sets. This can allow an attacker to optionally transition from LLC cache attacks to L1 cache attacks, which are considered to be more sensitive and simpler to carry out. More importantly, once the mapping of physical addresses to cache sets is reverse engineered on newer systems, this behaviour will allow low-privilege processes to infer information about the physical addresses of their own variables, reducing the entropy of several types of attacks such as ASLR derandomization [10].

When investigating the cache replacement policy, we noticed that the CPUs we surveyed transitioned between two distinct replacement policies. Modern Intel CPUs usually employ a least-recently-used (LRU) replacement policy [23], where a new entry added to the cache is marked as the most recently used, and is thus the *last* to be replaced in the case of future cache misses. In certain cases, however, these CPUs can transition to the bimodal insertion policy (BIP) policy, where the new entry added to the cache is marked most of the times as the least recently used, and is thus the *first* to be replaced in the case of future cache misses. In our measurements we noticed that Sandy Bridge CPUs kept using the LRU policy throughout our experiments. On Ivy Bridge processors, however, we witnessed situations where some sets operated in LRU mode and some in BIP mode. This suggests a "set dueling" mechanism, in which the two policies are compared in real time to examine which generates less cache misses. Haswell and Broadwell CPUs switched between policies with high frequency, but we could not locate regions in time where both policies were in effect (in different cache sets).

We hypothesize that Haswell (and newer) CPUs do not use simple set dueling, but rather a different method, to choose the optimal cache replacement policy. The choice of policy had a impact on our measurements, since the BIP policy makes the priming and probing steps harder. Priming is more difficult since sequentially accessing all entries in the eviction set does not bring the cache into a known state—some of the entries used by the victim process may still be in the cache set. As a result, the probing step may spuriously indicate that the victim has accessed the cache set in a certain time period. The combined effect of these two artifacts is an effective *low-pass filter* applied to the memorygram, which reduces our temporal resolution by a factor of up to 16. To avoid triggering the switch to BIP, we designed our attack code to minimize the amount of cache misses it generates in benign cases, both by choosing a zig-zag access pattern (as suggested by Osvik et al. [19]), and by actively pruning our measurement data set to remove overly active cache sets.

6.3 Noise Effects

Sources. Side-channel attacks have to deal with three general categories of noise [18]: electronic, switching, and quantization (or measurement). Electronic noise refers to the "thermal noise" which is inherent in any physical system. This source of noise is less prevalent in our attack setup due to its relatively low resolution. Switching noise refers to the fact that the measurements capture not only the victim's secret information, but also other activities of the device under test, either correlated or uncorrelated to the measurement. In our specific case, this noise is caused by the spurious cache events due to background process activity, as well as by the cache activity of the attack code and the underlying JavaScript runtime itself. Quantization noise refers to the inaccuracies introduced by the measurement apparatus. In our specific case, this noise can be caused by the limited resolution of the JavaScript performance counter, or by low-level events such as context switches that occur while the measurement code is running. It should be noted that, with the exception of timer granularity, all sources of noise in our system are additive (i.e., noise will only cause a measurement to take longer).

Effects. There are two main elements of our attack that can be impacted by noise. The first is the cache profiling process, in which the eviction sets are created. The second is the online step, in which the individual cache sets are probed. Noise during the profiling process, specifically during steps (1.b) and (1.e) of Algorithm 1, will cause the algorithm to erroneously include or exclude a memory address from an eviction set. Noise during the online step will cause the attacker to erroneously detect activity on a cache set when there is none, or to erroneously associate cache activity to a victim process when in fact it was caused by another source. Interestingly, one formidable source of switching noise is the measurement process itself—since a memorygram contains millions of measurements collected over a short period of time, creating it has a considerable impact on the cache.

Mitigations. To quantify the prevalence of measurement noise in our system, we measured the proportion of cache misses in an area with no cache activity. We discovered that around 0.3% of cache hits were mis-detected as cache misses due to timing jitter, mostly because off context switches in the middle of the measurement process.

Such events are easy to detect since the time that is returned is more than the OS multitasking quantum (10ms on our system). However, since we want our measurement loop to be as simple as possible, we did not apply this logic in our actual attack. To deal with the limited resolution of the timer on some targets, we could either use the workarounds suggested in the previous section or find an alternative form of time-taking that does not rely on JavaScript's built-in timer API. Timing jitter was generally not influenced by CPU-intensive background activities. However, memory-intensive activities, such as file transfers or video encoders caused a large amount of switching noise and degraded the effectiveness of our attack considerably. To deal with the switching noise caused by our measurement code, we spread out our data structures so that they occupied only the first 64 bytes of every 4KB block of memory. This guaranteed that at most 1/64 of the cache was affected by the construction of the memorygram.

Another source of noise was Intel's Turbo Boost feature, which dynamically varied our CPU clock speed between 2.5 GHz and 3.2 GHz. This changed the cache hit timings on our CPU by a large factor between measurements, making it difficult to detect cache misses. To mitigate this effect, we periodically estimated the cache hit time (by measuring the access time of a cache set immediately after priming it), and measured cache misses against this baseline.

6.4 Additional Attack Vectors

The general mechanism we presented in this paper can be used for many purposes other than the attack we presented. We survey a few interesting directions below.

KASLR Derandomization. Kernel control-flow hijacking attacks often rely on pre-existing code deployed by the OS. By forcing the OS kernel to jump to this code (for instance by exploiting a memory corruption vulnerability that overwrites control data), attackers can take over the entire system [12]. A common countermeasure to such attacks is the Kernel Address Space Layout Randomization (KASLR), which shifts kernel code by a random offset, making it harder for an attacker to hard-code a jump to kernel code in her exploits. Hund et al. showed that probing the LLC can help defeat this randomization countermeasure [10].

We demonstrated that LLC probing can also be carried out in JavaScript, implying that the attack of Hund et al. can also be carried out by an untrusted webpage. Such attacks are specially suited to our attacker model, because of drive-by exploits that attempt to profile and then infect users with a particular strain of malware, tailored to be effective for their specific software configuration [22]. The derandomization method we present can be used for boot-strapping a drive-by exploit, dividing the attack into two phases. In the first phase, an unprivileged JavaScript function profiles the system and discovers the address of a kernel data structure. Next, the JavaScript code connects to the web server again and downloads a tailored exploit for the running kernel.

Note that cache sets are not immediately mappable to virtual addresses, especially in the case of JavaScript where pointers are not available. An additional building block used by Hund et al., which is not available to us, is the call to `sysenter` with an unused syscall number. This call resulted in a very quick and reliable trip into the kernel, allowing efficient measurements [10].

Secret State Recovery. Cache-based key recovery has been widely discussed in the scientific community and needs no justification. In the case of cache attacks in the browser, the adversary may be interested in discovering the user's TLS session key, any VPN or IPSec keys used by the system, or perhaps the secret key used by the system's disk encryption software. There are additional secret state elements that are even more relevant than cryptographic keys in the context of network attacks. One secret which is of particular interest in this context is the sequence number in an open TCP session. Discovering this value will enable traffic injection and session hijacking attacks.

6.5 Countermeasures

The attacks described in this paper are possible because of a confluence of design and implementation decisions starting at the micro-architectural level and ending at the Java-Script runtime: the method of mapping a physical memory address to cache set; the inclusive cache micro-architecture; JavaScript's high-speed memory access and high-resolution timer; and finally, JavaScript's permission model. Mitigation steps can be applied at each of these junctions, but each will impose a drawback on the benign uses of the system.

On the *micro-architectural* level, changes to the way physical memory addresses are mapped to cache lines will severely confound our attack, which makes great use of the fact that 6 out of the lower 12 bits of an address are used directly to select a cache set. Similarly, the move to an exclusive cache micro-architecture, instead of an inclusive one, will make it impossible for our code to trivially evict entries from the L1 cache, resulting in less accurate measurements. These two design decisions, however, were chosen deliberately to make the CPU more efficient in its design and use of cache memory, and changing them will exact a performance cost on many other applications. In addition, modifying a CPU's micro-architecture is far from trivial, and definitely impossible as an upgrade to already deployed hardware.

On the *JavaScript* level, it seems that reducing the resolution of the high-resolution timer will make our attack more difficult to launch. However, the hi-res timer was created to address a real need of JavaScript developers for applications ranging from music and games to augmented reality. A possible stopgap measure would be to restrict access to this timer to applications that gain the user's consent (e.g., by displaying a confirmation window) or the approval of some third party (e.g., downloaded from a trusted "app store").

An interesting approach could be the use of heuristic profiling to detect and prevent this specific kind of attack. Just like the abundance of arithmetic and bitwise instructions used by Wang et al. to indicate the existence of cryptographic primitives [28], it can be noted that the various (measurement) steps of our attack access memory in a very particular pattern. Since modern JavaScript runtimes already scrutinize the runtime performance of code as part of their profile-guided optimization mechanisms, it could be possible for the JavaScript runtime to detect profiling-like behavior from executing code, and modify its response accordingly (e.g., by jittering the high-resolution timer or dynamically moving arrays around in memory).

7. CONCLUSION

We demonstrated how a micro-architectural, side-channel cache attack, which is already recognised as an extremely potent attack method, can be effectively launched from an untrusted webpage. Instead of the traditional cryptanalytic applications of the cache attack, we instead showed how user behaviour can be successfully tracked using our method(s). The potential reach of side-channel attacks has been extended, meaning that additional classes of systems must be designed with side-channel countermeasures in mind.

Acknowledgments

8. REFERENCES

[1] O. Aciiçmez. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proc. of ACM CSAW*, pages 11–18, 2007.

[2] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *Proc. of RAID*, pages 299–319, 2014.

[3] D. J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming, April 2005. [Online; accessed August-2015].

[4] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *Proc. of USENIX Sec.*, pages 1–14, 2005.

[5] Ecma International. Standard ECMA-262: ECMAScript®Language Specification. http://www.ecma-international.org/ecma-262/5.1/index.html, June 2011. [Online; accessed August-2015].

[6] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KEELOQ Code Hopping Scheme. In *Proc. of CRYPTO*, pages 203–220, 2008.

[7] D. Herman and K. Russell. Typed Array Specification. https://www.khronos.org/registry/typedarray/specs/latest/, July 2013. [Online; accessed August-2015].

[8] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick Tock: Building Browser Red Pills from Timing Side Channels. In *Proc. of WOOT*, 2014.

[9] W. Hu. Lattice Scheduling and Covert Channels. In *Proc. of IEEE S&P*, pages 52–61, 1992.

[10] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Proc. of IEEE S&P*, pages 191–205, 2013.

[11] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *Proc. of IEEE S&P*, pages 143–157, 2012.

[12] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. of USENIX Sec*, pages 957–972, 2014.

[13] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of CRYPTO*, pages 104–113, 1996.

[14] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Proc. of IEEE S&P*, pages 605–622, 2015.

[15] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer, 2007.

[16] J. Mann. High Resolution Time. http://www.w3.org/TR/hr-time/, December 2012. [Online; accessed August-2015].

[17] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-Cores Cache Covert Channel. In *Proc. of DIMVA*, pages 46–64, 2015.

[18] Y. Oren, M. Kirschbaum, T. Popp, and A. Wool. Algebraic side-channel analysis in the presence of errors. In *Proc. of CHES*, pages 428–442, 2010.

[19] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proc. of CT-RSA*, pages 1–20, 2006.

[20] D. Oswald and C. Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *Proc. of CHES*, pages 207–222, 2011.

[21] C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan*, 2005.

[22] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proc. of USENIX Sec.*, pages 1–15, 2008.

[23] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *Proc. of ISCA*, pages 381–391, 2007.

[24] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. of CCS*, pages 199–212, 2009.

[25] K. A. Shutemov. pagemap: do not leak physical addresses to non-privileged userspace. https://lwn.net/Articles/642074/, March 2015. [Online; accessed August-2015].

[26] StatCounter. GlobalStats. http://gs.statcounter.com, January 2015. [Online; accessed August-2015].

[27] W3C. Javascript APIs Current Status. http://www.w3.org/standards/techs/js. [Online; accessed August-2015].

[28] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Proc. of ESORICS*, pages 200–215, 2009.

[29] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. of CCS*, pages 305–316, 2012.

[30] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proc. of ACM CCS*, pages 990–1003, 2014.