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

Yossef Oren, Vasileios P. Kemerlis, Angelos D. Keromytis and Simha Sethumadhavan,
Computer Science Department, Columbia University

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 any 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, as well as extremely relevant and practical to today's Web, especially because most desktop browsers currently used to access the Internet are vulnerable to such side channel analyses. 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, other users, and even other 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. We show how this attack can be used to meaningfully compromise user privacy in a common setting, letting an attacker reliably spy after the browsing activity of a victim using a private browsing session. Defending against this side-channel is possible, but the required countermeasures can exact an impractical cost on benign uses of the browser.

1. INTRODUCTION

Side channel analysis is a remarkably powerful cryptanalytic technique. It allows attackers to extract secret information hidden inside a secure 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 et al. in 1996 [13], side channel analysis has been shown to be effective in breaking into a myriad of real-world systems, ranging from car immobilizers to high-security cryptographic coprocessors [7, 19]. A particular kind of side-channel attack that is relevant to personal computers is the cache attack, which exploits the use of cache memory as a shared resource between different processes, to disclose secret information [10, 18].

While the potency of side-channel attacks is established without question, their application to practical settings is debatable. The main limiting factor to the practicality of side-channel attacks is the problematic *attack model* they assume; with the exception of network-based timing attacks [6], most side-channel attacks require the attacker be in "close proximity" to the victim. Cache attacks, in particular, typically assume that the attacker is capable of executing arbitrary binary code on the victim's machine. While this assumption holds true for Infrastructure/Platform-as-a-Service (IaaS/PaaS) environments, like Amazon's cloud computing 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. 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 still launch an attack in a practical time frame and extract meaningful information from the system under attack. Keeping in tune with this computing setting, we chose not to focus on cryptographic key recovery, but rather on *tracking user behaviour*. The attack(s) described herein are therefore 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 personal computer powered by a late-model Intel processor. We furthermore assume that the user is accessing the web through a browser with comprehensive HTML5 support. As we show in Section 6.2, 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. 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, its 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 code is quite challenging, as Javascript cannot load shared libraries or execute native programs. Even more importantly, Javascript code is forced to make timing measurements using scripting language function calls instead of dedicated assembler instructions.

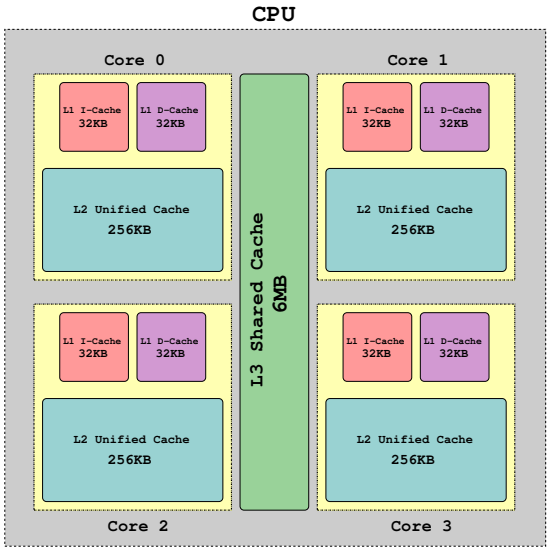

Figure 1: Cache memory hierarchy of Intel CPUs (based on Ivy Bridge Core i5-3470).

These challenges notwithstanding, we have 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 the system to be configured for large page support, and as such, 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 fully functional last-level cache attack using *unprivileged* Javascript code. 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 methods can be used to effectively *track the behaviour* of the user. 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 executable. While other works have consistently shown that cache attacks are capable of performing cryptographic key recovery, we believe this particular application of cache attacks is more relevant to our specific attack model.

- We describe possible countermeasures and discuss their system-wide cost(s).

2. BACKGROUND AND RELATED WORK

2.1 Memory Hierarchy of Modern Intel CPUs

Computer systems typically incorporate a high-speed Central Processing Unit (CPU) and a large amount of lower-speed Random Access Memory (RAM). To bridge the performance gap between these two components, modern systems 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 it is typically arranged in a *cache hierarchy* (i.e., a series of progressively larger and slower memory elements are placed in various levels between the CPU and the RAM). Figure 1, shows the cache hierarchy used by 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.[1] 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 which 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 4.1).

Intel's cache micro-architecture is *inclusive*: all elements in the L1 cache must also 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 protection 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 LLC's relatively large size, it is not efficient to search its entire contents whenever the CPU accesses the RAM. Instead, the LLC is 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, which belongs to the Haswell family, includes 8192 (2^{13}) cache sets, each of which is 12-way associative. This means that each 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 that we make great use 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 4.1. In the case of Sandy Bridge, this mapping was reverse-engineered in 2013 by Hund et al. [11], where they

[1]The current generation of Intel CPUs, codenamed Haswell/Broadwell, extends this hierarchy by another level of embedded DRAM (eDRAM), which is not discussed here.

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 use a *virtual memory mechanism*, where user processes do not have direct knowledge or access to the system's physical memory. Instead, these processes are allocated virtual memory *pages*. When a virtual memory page is accessed by a currently executing process, the Operating System (OS) dynamically associates the page with a *page frame* in physical memory. The Memory Management Unit (MMU) of the CPU is in charge of mapping between the virtual memory accesses made by different processes and accesses to physical memory. The size of pages and page frames in most Intel processors is typically set to 4KB[2], 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 the most well-known representative of the general class of micro-architectural side-channel attacks, which are defined by Aciiçmez in his excellent survey [3], 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—both secure and insecure processes can interact through their shared use of the cache. This allows an attacker to craft a "spy" process which can measure and make inferences about the internal state of a secure process through their shared use of cache memory. First identified by Hu in 1992 [10], several results have shown how the cache side-channel can be used to recover AES keys [5,18], RSA keys [20], and even allow one virtual machine to compromise another virtual machine running on the same host [22].

Our attack is modeled after the PRIME+PROBE method, which was first described by Osvik et al. [18] 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 which 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 code 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 yet again. A low access latency suggests that the attacker's eviction set is still in the cache, while 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 assembler instruction `rdtsc`, which provides a very sensitive measurement of the processor's cycle count. Iterating over the eviction set in the probing phase also serves a secondary purpose of forcing the cache set yet again into an attacker-controlled state, thus preparing for the next round of measurements.

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

2.3 The Web Runtime Environment

Javascript is a dynamically typed, object-based scripting language with runtime evaluation, which powers the client side of the modern web. Websites deliver Javascript programs to the browser in source-code form, which in turn are (typically) compiled and optimized using a Just-In-Time mechanism. The fierce competition among different browsers has caused browser vendors to focus considerably on improving Javascript performance; in fact, in certain scenarios, Javascript code performs on a level which is on par with that of native code.

The core functionality of the Javascript language is defined by the Ecma International industry association in the standard ECMA-262 [1]. The language standard is complemented by a large set of application programming interfaces (APIs) defined by the World Wide Web Consortium [23], which make the language practical for developing web content. The Javascript API set is constantly evolving, and browser vendors add support to 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 [8], 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 6.2, the vast majority of Web browsers that are in use today support both APIs.

In their default configurations, all common browsers will automatically compile and execute every Javascript program delivered to them by any webpage. To limit the potential damage of this property, Javascript code runs in a highly *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 language 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.

3. DESIGNING THE ATTACK

As described in Section 2.2, the four steps involved in a successful PRIME+PROBE attack are the following: (i) creating an eviction set for one or more relevant cache sets, (ii) priming the cache set, (iii) triggering the victim operation, and finally, (iv.) probing the cache set again. In this section, we describe how each of these steps was designed and implemented in Javascript.

3.1 Creating an Eviction Set

Design. As stated by Liu et al. [14], in the first step of a PRIME+PROBE attack the attacker creates an eviction set for a cache set whose activity would like to track. This

eviction set consists of a sequence of variables 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.

PRIME+PROBE attacks were first discussed in the context of the L1 cache [18]. More specifically, the L1 cache determines the cache set assignment for a variable based on the lower bits of its virtual address. Since the attacker is assumed to know the virtual addresses of its own variables, it is straightforward to create an eviction set in the L1 attack model. In contrast, set assignments for variables in the LLC are made by reference to their physical memory addresses, which are not generally available to unprivileged processes.[3] 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 4K 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 set. Instead, we use a heuristic algorithm as described below.

Let us now assume that the victim system has $s = 8192$ cache sets, each with a $l = 12$-way associativity. While investigating the way physical memory addresses are mapped into cache set indices in the Sandy Bridge micro-architecture, Hund et al. [11] discovered that accessing a contiguous 8MB *eviction buffer* of physical memory will completely invalidate 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 operating system into an arbitrary and non-contiguous set of 4K physical memory pages), and measured the system-wide effects of iterating over this buffer. We discovered that access latencies to unrelated variables in memory were slowed by a noticeable amount when 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 once per every 64 bytes (this behaviour was recently extended into a full covert channel [17]). However, it was 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 did not know the physical memory locations of the various pages of our buffer.

A naive approach to solving this problem would be to fix an arbitrary "victim" address in memory, and then find by brute force which subset of size $l = 12$ offsets, out of the 8MB/64=131K possible addresses in the buffer, serves as the eviction set for this address. To do so, we could randomly choose a subset of 12 offsets, and then measure whether the access latency to this victim address is increased after iterating through these offsets. If the latency increases, this means the subset contains the 12 addresses sharing the set with the victim address. If the latency does not change, then the subset does not contain at least one of these 12 addresses, allowing the victim address to remain in the cache. By repeating this process 8192 times, each time with a different victim address, we would be able to identify each cache set and create our data structure.

[3]In Linux, until recently, the mapping between virtual pages and physical page frames was exposed to unprivileged user processes through the `/proc/<pid>/pagemap` interface [**?**]. Starting with v4.0 this is no longer possible [**?**].

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.

Optimization #1. An immediate application of this heuristic would take an impractically long time to run. One simple optimization would be to start with a subset containing all 131K possible offsets, then gradually attempt to shrink it by removing random elements and checking that the access latency to the victim address stays high. 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 in uniquely determining the cache set index. 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 described in Algorithm 1.

By running Algorithm 1 multiple times, we can gradu-

CPU Model	Micro-arch.	L3 Cache 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).

ally create eviction sets covering most of the cache, except for those parts which 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 therefore, 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 it is run. 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.

```
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 =
10 window.performance.now();
11 currentEntry =
12 primeView.getUint32(variableToAccess);
13 var endTime = window.performance.now();
```

Evaluation. We implemented Algorithm 1 in Javascript and evaluated it on Intel machines using CPUs from the Ivy Bridge, Sandy 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 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 in Listing **??** illustrates lines 1.d and 1.e of the algorithm, and demonstrates how we iterate over the eviction set and measure latencies using Javascript. The algorithm requires some additional steps to run under Chrome and under Internet Explorer, which we describe in Section 6.2.

Figure 2 shows the performance of the profiling algorithm, as evaluated on an Intel i7-4960HQ running Firefox v35 for Mac OS X v10.10. We were pleased to find that the algorithm 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 very simple to parallelize, since most of its execution time is spent on data structure maintenance and only a small part is spent in the actual invalidate-and-measure portion. Finally, note that the entire algorithm is implemented in $\sim$ 500 lines of Javascript code.

To verify that our algorithm was indeed capable of identifying cache sets, we designed an experiment that compares

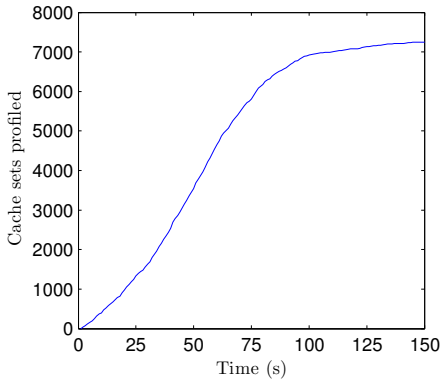

Figure 2: Cumulative performance of the profiling algorithm (Haswell i7-4960HQ).

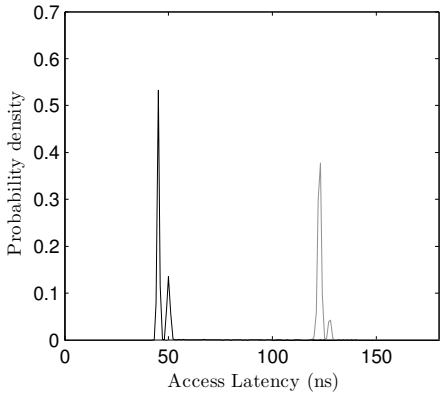

Figure 3: Probability distribution of access times for flushed vs. unflushed variable (Haswell i7-4960HQ).

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.

3.2 Priming and Probing

Once the attacker has an eviction set consisting of 12 entries that share the same cache set, his 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. While this step seems trivial, there are several performance-enhancing features of modern Intel CPUs which must be considered.

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 the operation was 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 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. [18]), and by 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 4.1).

A final challenge is the issue of timing jitter. In contrast to native code PRIME+PROBE attacks, which use a single 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 now correlate the cache sets he has profiled to data or code locations belonging to the victim. This learning/classification problem was addressed earlier by Zhang et al. [25] 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 (c) of the algorithm was actually quite challenging due to the limited permissions granted to Javascript code. This can be contrasted with the ability of Irazoqui et al. to trigger a minimal kernel operation by invoking an empty `sysenter` call [4]. To carry out this step, we had to survey the Javascript runtime to discover function calls which may trigger interesting behaviour, such as file access, network access, memory allocation, etc. We were also interested in functions which take a relatively short time to run and left no background "trails", such as garbage collection, which would impact our measurement in step (d). Several such functions were discovered in a different context by Ho et al. [9]. 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 his keyboard) on the behalf of the attacker. 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. EVALUATION

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 (the highest temporal resolution we were able to achieve while monitoring 128 cache sets in parallel is approximately 5µs). The intensity of each pixel corresponds to the access latency of this 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 immediate 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 which 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 which spans more than 64 bytes of assembler instructions. Several smaller groups of cache sets are also accessed together. We theorise that the these smaller groups correspond to variable accesses. Finally, the white horizontal

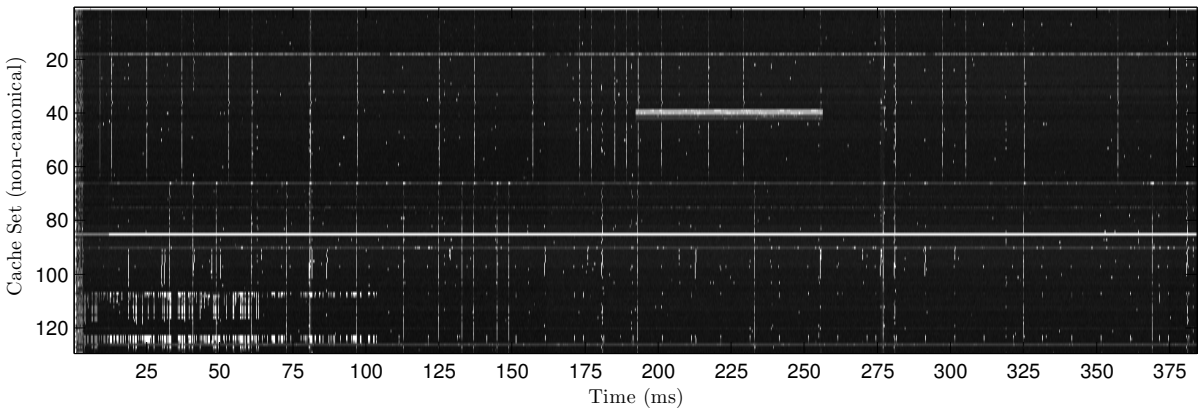

Figure 4: Sample memorygram collected over an idle period of 400ms. The X axis corresponds to time, while the Y axis corresponds to different cache sets. The sample shown has a temporal resolution of 250µs and monitors a total of 128 cache sets. The intensity of each pixel illustrates the access latency of the particular cache set, with black representing a low latency and white representing a higher latency.

line indicates a variable which is constantly accessed during our measurements. This variable probably belongs to the measurement code or to the underlying Javascript runtime. It is remarkable that such a wealth of information about the system is available to an unprivileged webpage!

4.1 Micro-architecture insights

Despite the high-level language in which our attack was written, it provides a glimpse into extremely low-level elements of the victim machine. As a consequence, we were affected by minute design choices made by the designers of the Intel microprocessor among the different processor generation we surveyed.

As stated in [3], two concepts can affect the functional behavior of a cache: the *mapping strategy* and the *replacement policy*. The mapping strategy determines which memory locations are mapped to each set in the cache, while the replacement policy determines how the cache set will be modified after a cache miss.

We noticed differing behaviour in the *mapping strategy* of the systems we surveyed, specifically in the choice of the *slice index* of each memory address. In the processors we surveyed, each cache slice is assigned to a specific CPU core, while all of the slices are interconnected via ring buffer.

While the work of [11] showed that on Sandy Bridge CPUs the slice index was only a function of high-order bits of the physical address, it was suggested in [14] that lower-order bits are also considered in the calculation on newer microarchitectures. We confirmed this by measuring the *cache hit timing* of each of the cache sets we were able to profile on a quad-core Haswell processor. In a such a system there are three possible times for an L3 cache hit - the slice associated with the current core, the two slices a single core away, and the single slice which is two cores away. As illustrated in Figure 5, each consecutive set of 64 cache sets (associated in memory with 4K consecutive addresses in physical memory) exhibits a distinct 3-level timing graduation, suggesting that newer CPUs choose the slice index by consulting lower-order bits of the address. There are operative outcomes to this discovery – 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

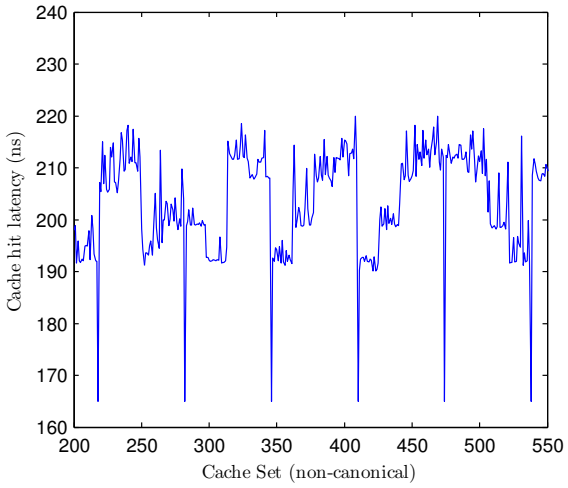

Figure 5: L3 cache hit times show a 3-level graduation (Haswell i7-4960HQ)

the same cache sets. 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 some information about the physical addresses of their own variables, reducing the entropy of several types of attacks such as ASLR derandomization.

When investigating the *cache replacement policy*, we noticed that the CPUs we surveyed transitioned between two distinct replacement policies. As discussed in [**?**], modern Intel CPUs usually employ least-recently-used (LRU) replacement policy, where the new entry added to the cache is marked as least 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 most 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 remained in 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. On the Haswell and Broadwell processors we noticed the system switching between policies with high frequency, but could not locate regions in time where both policies were in effect in different cache sets. This suggests that Haswell and newer CPUs use 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 more difficult. 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 zigzag access pattern (as suggested by [18]), and by actively pruning our measurement data set to remove overly active cache sets.

4.2 Covert Channel Bandwidth Estimation

As shown in [14,17], last-level cache access patterns can be used to construct a high-bandwidth covert channel and effectively exfiltrate sensitive information between virtual machines co-resident on the same physical host. We used such a construction to estimate the measurement bandwidth of our attack system. 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 Subsection 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 4K 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 4K array, allowing the us to transmit 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 physical memory, 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 attempted to search for this memory access pattern using a Javascript cache attack, then measured the maximum sampling frequency at which the Javascript code could be run. We first evaluated our code when both transmitter and receiver were running on a normal host, then 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 the covert channel was in the standard case was measured to be approximately 320kbps, a figure which compares well with the 1.2Mbps bandwidth achieved by the native code cross-VM covert channel implemented by [14]. When the receiver code was not running directly on the host, but rather on a virtual machine, the peak bandwidth of the covert channel was severely degraded to approximately 8kbps. Nevertheless, the fact that a webpage running inside a virtual machine is capable of probing the hardware of the underlying host is still quite surprising.

5. USER BEHAVIOUR TRACKING

Most works which evaluate cache attacks assume that the attacker and the victim share a colocated machine inside a cloud-provider data center. Such a machine is not typically configured to accept interactive input, and accordingly most works in this field focus on the recovery of cryptographic keys or other secret state elements, such as random number generator states [26]. For 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 [22] 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 operating system code that handles them is non-negligible. Thus, we expected them to have a relatively large cache footprint. They are also easily triggered by content running within the restricted Javascript security model, allowing our attack to have a training phase.

5.1.1 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 (i.e. network activity or mouse activity) is triggered, and the cache activity 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].

By comparing the cache activity 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 the user's 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. We then calculated the mean access latency of each cache set in every cluster, arriving at 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 as the closest one.

In the classification phase, we generated network traffic using the command-line tool wget and moved the mouse

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

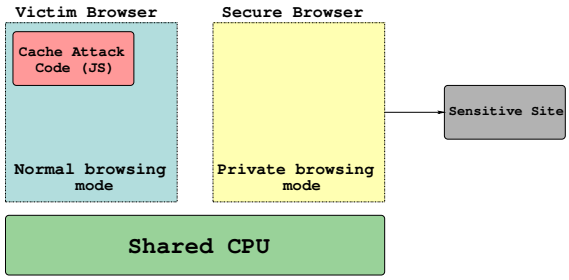

Figure 6: End-to-end attack scenario

outside of the browser window. To provide ground truth for the network activity scenario, we concurrently measured the traffic on the system using tcpdump, then mapped the timestamps logged by tcpdump 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 moved the mouse over this webpage.

In our experiments we discovered that we were capable of reliably detecting mouse and network activity. Our the network classifier's measurement rate was only 500Hz. Thus, it could not count individual packets but rather periods of network activity and inactivity. In contrast, our mouse detection code actually logged more events than the ground truth collection code. This is due to the fact that the Chrome browser throttles mouse events to web pages down to a rate of approximately 60Hz. We stress that the mouse activity detector did not detect network activity, and vice versa.

During our hardware measurement we unearthed a surprising artifact. We discovered that our measurements were affected by the *ambient light sensor* of the victim machine – waving our hand in front of the laptop screen generated a noticeable burst of hardware events. This means that cache-based attacks can detect the *presence of a user* browsing the computer, an item of information which is highly desirable to advertisers.

5.2 An End-to-End Attack on Privacy

We now demonstrate how a cache attack can practically and meaningfully compromise the privacy of a web user.

5.2.1 Motivation

Most consumer browsers on the market today implement a private or incognito mode, which users use to carry out more sensitive types of online activities. While private browsing mode is enabled, the web browser does not disclose or collect any cookies, web cache entries or other forms of local data storage. Since private browsing sessions do not retain the login credentials of the current user, they are cumbersome to use for general purposes; Instead, users typically have concurrent *standard browsing sessions* and *private browsing sessions* running side-by-side on the same computer. Users may even use different browser executables for the different sessions, carrying out their standard browsing sessions in one browser, but opening their private browsing sessions in a different browser, perhaps one with more restrictive security settings.

Let us now assume that one of the windows belonging to the *standard browsing session* is capable of performing a cache attack (either by malicious design, or incidentally via a malicious ad banner or other affiliate content item). As Figure 6 illustrates, we show how this attacker can detect which websites are being loaded in the victim's *private browsing session*, thus compromising his privacy.

5.2.2 Experiment Setup

Our measurements were carried out on an Intel Core i7-2667M machine running the latest version of Mac OS (10.10.3). The attack code was run on a standard browsing session running on the latest version of Firefox (37.0.2), while the private browsing session was run on the latest version of Safari (8.0.6). To increase our measurement bandwidth we chose to filter all hardware-related events. We thus we began our attack with an extremely simple training phase, in which the attacker measured which cache sets were idle when the user was touching the trackpad but not moving his finger. We detected a total of 61 such cache sets, out of a total of 2048 output by the profiling step.

In each experiment, we opened the private mode browsing window, typed the URL of a website the address bar and allowed the website to load completely. During this operation, our attack code collected a 10-second memorygram with a temporal resolution of 2ms. We collected a total of 39 memorygrams for 8 of the top 10 sites on the web, according to Alexa's Top Global Sites ranking of 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 measurement. A representative set of these memorygrams is provided in Figure 7[5].

Our classification step was extremely simple - we calculated the mean absolute value of the Fourier transforms for each website's memorygrams (discarding the DC component), calculated the absolute value of the Fourier transform of the current memorygram, then output the label of the closest website according to the ℓ^2 distance metric. We performed no other preprocessing, alignment or other modification of the data. In each experiment, we trained the classifier on all traces but one, then recorded the label output by the classifier for the missing trace.

5.2.3 Results

The confusion matrix of our classifier is shown in Table 2. The overall classification accuracy of our classifier was 82.1%, a value which can certainly be improved by more advanced classification heuristics such as measuring the timing of the keystrokes of the URL as it is entered. 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 which generates the distinct 2Hz pulse seen in Figure 7.

Preliminary results suggest that it is possible to track websites loaded by the highly secured Tor Browser using the same method. However, as pages loaded over the Tor network typically take longer than 10 seconds to load, our current test setup was not able to capture enough data to mount a successful attack.

6. DISCUSSION

This work shows how side-channel attacks can be applied to an entirely different attack model. To the best of our

[5]While the memorygrams shown in the figure were manually aligned for readability, our attack code did not perform this alignment step.

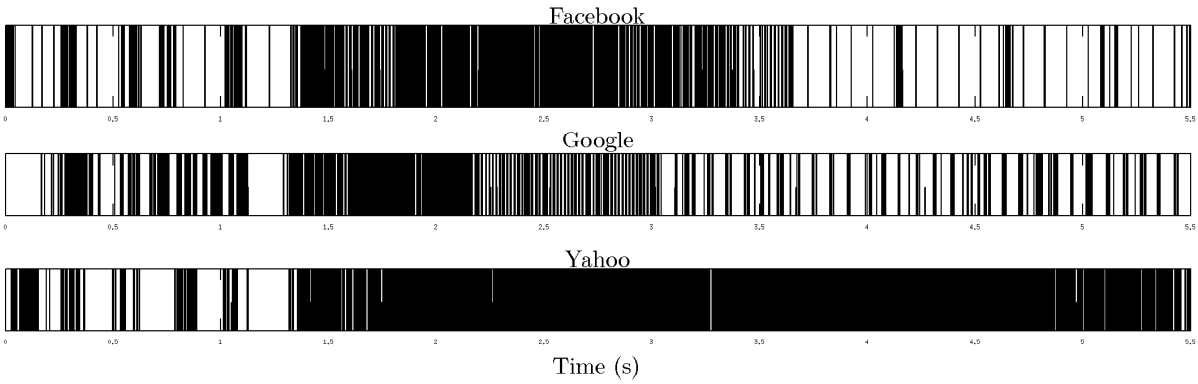

Figure 7: Memorygram outputs for three popular websites

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

knowledge, this is the first side-channel attack which can scale effortlessly into millions of targets, since in effect the DUT is measuring itself.

6.1 Implications of the Attack

Webpages are protected from each other, and from the underlying machine, by a series of trust boundaries. These include web-centric defenses such as the Web Origin concept the Javascript sandbox, but also OS-level mechanisms such as process separation and privilege rings. The fact that a webpage has read access outside of its own trust boundary has troubling implications to systems running other processes together with the web browser. In a sense, no process on the system can be considered more secure than the least secure webpage running on the system.

6.2 Prevalence of Vulnerable Systems

Our attack requires a personal computer powered by an Intel CPU based on the Sandy Bridge, Ivy Bridge, Haswell or Broadwell microarchitectures. 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 which supports the HTML 5 High Resolution Time API and the Typed Arrays specification. Table 3 notes the earliest version at which these APIs are supported for each of the common browser brands, as well as the proportion of global Internet traffic coming from vulnerable browser versions, according to StatCounter GlobalStats measurements as of January 2015 [2]. As the table shows, more than 80%

Brand	Hi-Res Time Support	Typed Arrays Support	Worldwide prevalence
Internet Explorer	10	11	11.77%
Safari	8	6	1.86%
Chrome	20[6]	7	50.53%
Firefox	15	4	17.67%
Opera	15	12.1	1.2%
Total	–	–	83.03%

Table 3: Prevalence of vulnerable desktop browsers [2].

of desktop browsers in use today are vulnerable to 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 operating systems. In our testing we discovered, for instance, that the actual resolution of this timestamp for Safari for MacOS was on the order of nanoseconds, while Internet Explorer for Windows had a 0.8µs resolution. Chrome, on the other hand, offered a uniform resolution of 1µ on all operating systems we tested.

Since, as shown in Figure 3, the timing difference between a single cache hit and a single cache miss is on the order of 50ns, the profiling and measurement algorithms need to be slightly modified to support systems with coarser-grained timing resolution. 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[7]. For the measurement stage, we cannot amplify a single cache miss, but we can take advantage of the fact that code access typically invalidates multiple consecutive cache sets from the same page frame. As long as at least 20 out

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

of the 64 cache sets in a single page frame register a cache miss, our attack is successful even with microsecond time resolution.

The attack we propose is also easily 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.

6.3 Additional Attack Vectors

The general attack 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:

Kernel Space Derandomization: Control-flow hijacking attacks, such as buffer overflow attacks, often rely on the existence of pre-existing program code deployed by the operating system. By forcing the program to jump to this code (for instance by using a buffer overflow which overwrites a function's return address), attackers can execute arbitrary code with heightened privileges and thus take over the entire system. A common countermeasure to these attacks is the Address Space Layout Randomization countermeasure (ASLR), which randomly relocates operating system libraries in the process virtual memory, making it impossible for the attacker to hard-code a jump to operating system code in their exploits. As discussed in [11], if a kernel-mode function jumps to an incorrect address, this will lead to a complete system crash, meaning an attacker only has a single chance to attempt a control flow hijack.

Hund et al. show in [11] how probing the last-level cache can help defeat this randomization countermeasure. 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. This attack is especially suited to our attacker model, specifically that of "drive-by exploit" sites, which attempt to profile and then infect users with a particular strain of malware tailored to be effective for their specific software configuration [21]. The derandomization method we present can be used for "bootstrapping" a drive-by exploit, by 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 custom-tailored binary exploit which jumps directly to that address.

It should be noted that cache sets are not immediately mappable to virtual addresses, specifically in the case of Javascript where pointers are not available. The intermediate steps used in [11] to overcome this limitation are also relevant to this discussion. An additional building block used by Hund et al., which was not available to us, was 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.

Secret State Recovery: Cache-based key recovery has been widely discussed in the scientific community and needs no justification. In the particular 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 which 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. It would be interesting to see if a cache attack can be used to discover this value.

System and user profiling: As shown in [9] and [12], unprivileged processes can gain considerable power by learning about the specific environment they are running in. A cache attack can be used to survey a system for access patterns related to common operating systems, programs or activities (i.e. VoIP), and thus allow pinpointing the system in future accesses. On the more offensive side, the system can also be examined for the presence of antivirus programs, vulnerable plugins or other loadable modules, and thus pave the way for further exploitation by other means.

6.4 Countermeasures

The attacks described in this report are possible because of a confluence of design and implementation decisions starting at the micro-architectural level and ending at the Javascript 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 the fact that 6 of the lower 12 bits of the 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, making measurement much more difficult. These two design decisions, however, were chosen deliberately to make the CPU more efficient in its design and in its 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 this 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 and telemedicine. A possible stopgap measure would be to restrict access to this timer to applications which gain the user's consent (for example, by displaying a confirmation window) or the approval of some third party (for example, by being 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 was used by Wang et al. to indicate the existence of cryptographic primitives [24], 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 should be possible for the Javascript runtime to detect profiling-like behavior from executing code and then modify its response accordingly (for example by jittering the high-resolution timer,

dynamically moving arrays around in memory, etc).

7. CONCLUSION

In this paper, we demonstrated how the micro-architectural, side-channel cache attack, which is already recognised as an extremely potent attack method, can be effectively launched from an untrusted web page. Instead of the traditional cryptanalytic application 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 secure systems must be designed with side-channel countermeasures in mind.

8. REFERENCES

[1] ECMA-262: ECMAScript Language Specification. Online, June 2011. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[2] Statcounter globalstats. Online, January 2015. http://gs.statcounter.com.

[3] O. Aciiçmez. Yet another MicroArchitectural Attack: Exploiting I-Cache. In *Proc. of ACM CSAW*, Fairfax, VA, USA, November 2007.

[4] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! A fast, cross-vm attack on AES. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*. Springer, 2014.

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

[6] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5), 2005.

[7] 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 IACR CRYPTO*, Santa Barbara, CA, USA, August 2008.

[8] D. Herman and K. Russell. Typed Array Specification. Online, July 2013. https://www.khronos.org/registry/typedarray/specs/latest/.

[9] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick tock: Building browser red pills from timing side channels. In S. Bratus and F. F. X. Lindner, editors, *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. USENIX Association, 2014.

[10] W. Hu. Lattice Scheduling and Covert Channels. In *Proc. of IEEE S&P*, Oakland, CA, USA, May 1992.

[11] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proc. of IEEE S&P*, San Fransisco, CA, USA, May 2013.

[12] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012.

[13] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of IACR CRYPTO*, Santa Barbara, CA, USA, August 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*, San Jose, CA, US, May 2015.

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

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

[17] C. Maurice, C. Neumann, O. Heen, and A. Francillon. C5: Cross-cores cache covert channel. In *DIMVA 2015, Detection of Intrusions and Malware, and Vulnerability Assessment, July 9-10, 2015, Milano, Italy*, Milano, ITALY, 07 2015.

[18] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proc. of CT-RSA*, San Jose, CA, USA, February 2006.

[19] D. Oswald and C. Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *Proc. of IACR CHES*, Nara, Japan, September 2011.

[20] C. Percival. Cache Missing for Fun and Profit. Online, May 2005. http://www.daemonology.net/hyperthreading-considered-harmful/.

[21] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In P. C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*. USENIX Association, 2008.

[22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. ACM, 2009.

[23] W3C. Javascript APIs. Online. http://www.w3.org/standards/techs/js.

[24] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In M. Backes and P. Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*. Springer, 2009.

[25] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In T. Yu, G. Danezis, and V. D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 2012.

[26] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proc. of ACM CCS*, Scottsdale, AZ, USA, November 2014.