Side Channel Communications

Alexis Callemard

November 2017

1 Introduction

In the current world of modern computing, security has been taken seriously and operating systems as well as software have taken measures to reduce the information available to processes. That is especially the case in multi-processes software, such as web browsers, which run arbitrary code and isolate each process. From virtualization to sand-boxing, malevolent information access is being cracked down on. An answer to this challenge for clandestine communications is to use covert channels. One of such possible channels relies on hardware properties of the cache, which possesses, in the case of contemporary Intel processors, a level physically shared between all cores (see figure 1).

The first use for those kind of cache attacks required binary code to be run locally on the target, usually on shared servers running virtual machines for each guest as is the case with many cloud instances. In such a setup, the attacker goal was to recover cryptographic keys used on the machine.

We will study the papers published by Oren et al. [1], Maurice et al. [2] and Schwarz et al. [3]. In those papers, our interest will lean towards how to make use of cache workings to exchange information, which will involve cache hits and cache misses, but also how we can get a precise enough timer to measure the access time of the data and observe the difference between a hit and a miss.

For our purpose, we will look at it from a practical side of things, and look at how such an attack can be achieved using only JavaScript embedded in a web-page, which targets almost every modern browser in use. We will also focus only on Intel CPUs[1], and on covert channels rather than cryptographic key recovery.

2 Background

As hardware gets faster, I/O[2] operations are the bottleneck when doing computations. For faster access than on a disk, we use RAM, however RAM[3] memory is still really slow. To cope with it, modern CPUs implement a cache memory that is both smaller and faster, usually divided in different levels. The cache stores content that has recently been addressed by the CPU, since they are more likely to be re-accessed. Below in figure 1 is pictured the cache memory hierarchy of Intel CPUs.

[1] Central Processing Unit
[2] Input/Output
[3] Random-Access Memory

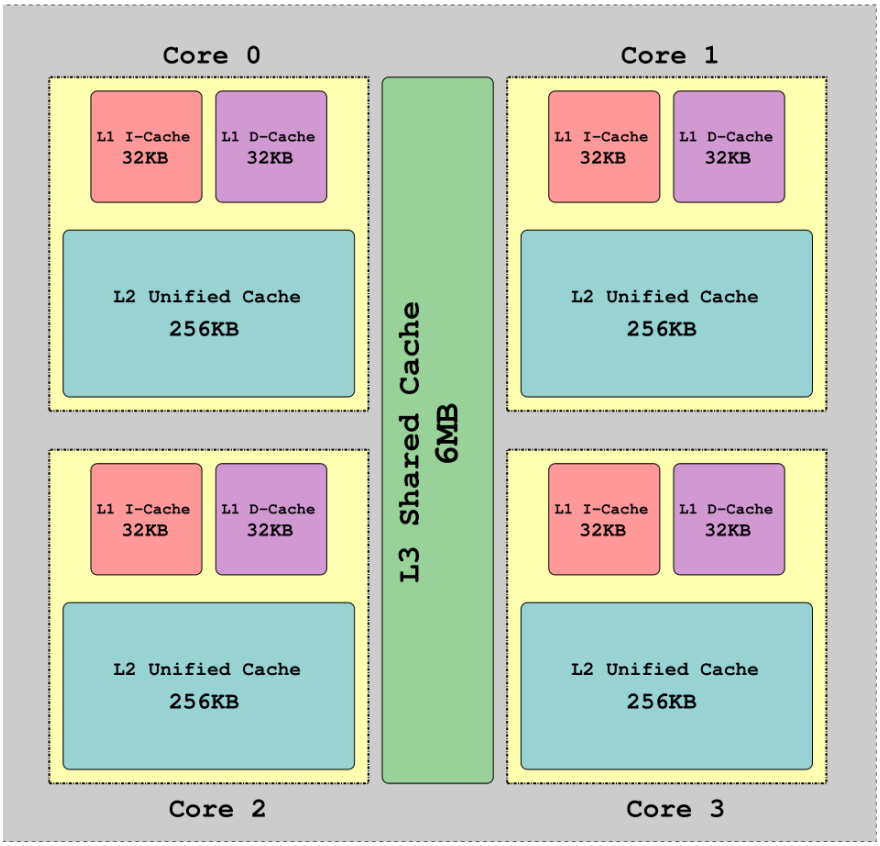

Figure 1: Cache memory hierarchy of Intel CPUs [1]

The functioning of the cache is the following: if the data accessed is in the cache, you will access it from there and benefit from the speed up (hit). In the case it is not, then another line of cache will be removed so the data can be added in (miss).

We can see in figure 1 that there are 3 levels of cache. The level 1 (L1) which is both the fastest and smallest and the level 2 (L2) which has more capacity with the inconvenient of being slower are both available individually for each core. The level of interest for us is the level 3, shared between all the cores. Intel caches are inclusive, meaning the higher levels of cache include the content of the lower levels. A line removed from a higher level will also be in the lower levels. That is contrary to the functioning of AMD processors, which are exclusive, and need the attack explained in section 3.1 to be reworked for the architecture. For convenience, the third level cache will hereafter be referred to by its common name, LLC (Last Level Cache).

In the case of Intel CPUs, the cache is set-associative, meaning it is separated in small chunks called *sets*, themselves containing *lines*, and to access data you need to access the whole line. To access a line, the CPU will calculate its index to deduce which set it belongs to, and look at the tag to know which line is to be accessed.

When trying to see if a physical address is cached, the corresponding line address will be calculated and if the line is already cached then you will get a speed up. Otherwise, the line will be loaded and the cache replacement policy will evict another line.

3 Cache attacks

3.1 Principles and JavaScript proof of concept

The work of Oren et al. [1] explains how a cache attack can be made more practical by having the target open an infected web-page. Such attacks were thought of as needing to run a malevolent binary on the target's physical host, and were used to recover cryptographic keys. The paper instead shifts the focus on using JavaScript and tracking user behaviour.

Modern web browsers feature a JavaScript engine, which runs any JavaScript code in a web-page to make the web more dynamic. Since this poses security concerns, the code is run in a sand-boxed environment, restricting its access to the system. In addition, each browser tab is its own separate process and his access to other processes data is also restricted.

Under those circumstances, JavaScript code: "cannot open files, even for reading [...]. 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." [1]

3.1.1 Prime+Probe

Prime+Probe attack is used to learn of the internal state of the victim's cache. It involves four step as summed up by Oren et al. [1]:

- creating an eviction set for one or more relevant cache sets;
- priming the cache set;
- triggering the victim operation;
- probing the cache set again.

If the probe access is fast, there is high probability that the eviction set is still in the cache. In the case the latency is higher, it suggests that some of the attacker's elements were evicted due to the victim using this cache set.

Creating an eviction set
We create an eviction set for a cache set we want to track. It "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". [1]. Since JavaScript has no notion of pointers, to map an address to a cache set we would need 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" [1].

Such a brute-force tactic takes too long to run, so Oren et al. devised two optimisations:

- gradually attempt to shrink the subset containing the 131K offsets by randomly removing elements and checking that the latency to the victim address stays high;
- 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, and discovering a single cache set can immediately teach us about 63 additional cache sets (each block of 4KB virtual memory maps to 4KB block in physical memory).

The method they used enabled them to create eviction sets that cover almost all of the cache with the exception of the parts accessed by the JavaScript runtime.

Prime and probing
After identifying an eviction set, the attacker's goal is to replace each entry in the CPU cache with elements of this eviction set. For the probe step, he also needs to measure the time needed to perform the operation precisely.

As modern CPUs execute instructions according to data availability, the eviction set was accessed in the form of a linked-list, to ensure that it was accessed before the timing measurement code was run. The CPUs also have a *stride prefecthing* feature which tries to guess the next memory accesses based on patterns in the the accesses history. This is avoided by randomly permutting elements in the eviction set. Oren et al. also access the set in alternating directions to avoid too much cache misses.

Identifying interesting cache regions
The cache sets received are not correlated to code or data belonging to the victim. To get a correlation, machine learning is used to derive meaning from cache sets latency measurement. For that step, the victim is incensed to perform an action.

3.2 Fantastic timers in JavaScript

Having taken note of micro-architectural attacks, the web actors have moved to reinforce the security of browsers and try to prevent such threats. One such example is the JavaScript API[4] being evolved to close vulnerabilities, specifically, timers have been reworked to be less precise (`performance.now` used to provide nanosecond precise measurements and is now in the millisecond range). In their paper, Schwarz et al. explore more ways to get a reliable timer for a micro-architectural attack and develop "measurement methods that exceed the resolution of official timing sources by 3 to 4 orders of magnitude on all major browsers, and even more on Tor browser" [3].

3.3 Recovering a high resolution

Previous papers showed that clock edges allow the recovery of a high resolution, and they align the timestamp to the resolution (timestamp is a multiple of the rounded down resolution at the given time).

Edge thresholding
For many side-channel attacks distinguishing between slow f_{slow} and fast f_{fast} operations is enough. Their execution time is noted t_{slow} and t_{fast}.

Swharz et al. devised a new resolution method called edge thresholding. This new method relies on "the property that we can execute multiple constant-time operations between two edges of the clock". They generate a padding containing multiple constant-time operations after the function they want to measure, which increases the total execution time by a constant. They choose the padding "in such a way that $t_{slow} + t_{padding}$ crosses one more clock edge than $t_{fast} + t_{padding}$" which means the amount of clock edges taken by each function is different.

[4] Application programming interface

Alternative timing primitives
When the High Resolution Time API can not be used (such is the case for Tor Browser), there is a need to create our own timing sources. We do that by creating a fast-paced monotonically increasing counter as a timing primitive that is an approximation of a highly accurate monotonic timer, as explained by Schwarz et al. [3]. In JavaScript the primitive needs to be based on recurrent events or non-JavaScript browser features, as the execution is single threaded.

3.4 Covert channel communication

Covert channels make use of the micro-architecture to communicate "surreptitiously" between different processes. Such covert channels can be attained with the cache attacks explained earlier as a basis. As we understood the inner-workings of the cache, we understand that it susceptible to noise.

Maurice et al. [2] provide a comprehensive characterization on noise, build the first robust covert channel and use it to sustain an SSH connection between two virtual machines.

3.4.1 Measurement errors

Since the cache is small and shared between all cores and processes, the data from the covert channel gets evicted by other concurrent programs. As the sender process evicts data frequently to transmit bits, this creates interferences with the covert channel that the receiver sees as either false positives or false negatives.

A second kind of error that the covert channels suffer from is scheduling errors. Depending on the scheduler, the sender and receiver may not be running simultaneously. Having no way to signal if they are emitting or receiving, the receiver may suffer from unwanted insertions or deletions.

Both errors can be avoided by synchronising both the sender and the receiver. It is achieved by having a common clock between them.

3.4.2 Robust cache covert channels

To create the robust channel, they developed a new communication protocol. This protocol works at two layers, the physical layer and the data-link layer.

First, the data-link layer will receive the data. It will chunk it in equally sized chunks and add error correction to make it into a packet. Then the transmission starts, each packet is divided in small enough parts (words) to be transmitted in between scheduling, then encoded to prevent synchronisation errors.

At the physical layer, the words are transmitted as a sequence of '0's and '1's (symbols). To transmit information, the sender primes and probes to evict the receiver's line in the same LLC set resulting in the same eviction in the receiver's L1 cache set.

To receive information, the receiver probes its lines in the L1 cache set to recover the symbols.

4 Conclusion

Cache attacks are hard to prevent against and browser software among others have been working on the issue, for instance when removing a CPU-cycle precise timer in favour of a less precise one.

We saw however that from the attacker point of view it can be circumvented by using other timing methods (Schwarz et al. [3]).

We know through the work of Oren et al. [1] that every host running a modern browser is vulnerable to cache attacks, making the previous attack surface of such micro-architectural attacks many times wider.

We also saw that establishing a robust error-correcting, error-handling, high-throughput covert channel through the cache via JavaScript has been achieved, taking the practicality of cache attacks to new heights. Micro-architecture attacks can now run on almost every host, without effective countermeasures in place as of yet.

References

[1] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1406–1418. ACM, 2015.

[2] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS. The Internet Society, February 2017.

[3] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript . In *Proceedings of the 21st International Conference on Financial Cryptography and Data Security (FC'17)*, FC, April 2017.