SIDE-CHANNEL ATTACKS AND THEIR IMPLICATIONS

MAËL NOGUES

INTRODUCTION

Side-channel attacks have been known and used by the intelligence community as early as in WWII and were first discussed in an academic context in 1996 by Kocher (4). They allow attackers to extract information hidden in a device by analysing the physical signals it emits as it performs its secure computations.

Side-channel analysis has been shown to be very effective in a lot of different systems from the real-world, like car immobilizers or high-security cryptographic coprocessors. A particular kind of side-channel attacks, which can be used on personal computers, are cache attacks. It exploits the use of cache memory as a shared resource between different processes to disclose information.

They can be used in different ways. For example, early on, they were exploited by programs that were run locally on the victim's machine, nowadays, we are able to exploit them in JavaScript which enables the use from a web browser and therefore remotely. This is possible because JavaScript scripts run locally on the machine running the browser it is executed on.

Cache attacks that runs remotely are generally timer based, meaning they use the time it takes for the processor to answer a cache access to derive if the process, with which they share their slice of the cache, has used the cache or not.

BACKGROUND AND STATE OF THE ART

MEMORY HIERARCHY OF INTEL CPUS

Cache memory contains a subset of the RAM's contents, only content recently accessed by the CPU, and is arranged in a *cache hierarchy*: series of progressively larger and slower memory elements, placed in different levels between the CPU and RAM.

Figure 1 shows the cache hierarchy of Intel Haswell CPUs, incorporating a small and 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 the 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 the RAM.

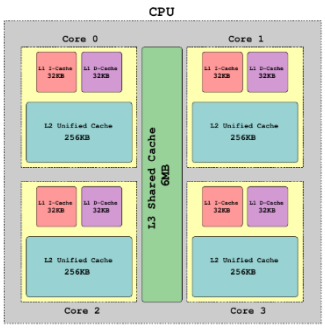

Figure 1: Cache memory hierarchy of Intel CPUs (based on Ivy Bridge Core i5-3470).

If the CPU needs something 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, called replacement policy, which has changed between processor generations.

Intel's cache micro-architecture is *inclusive*: all elements in the L1 cache exist in the L2 and L3 caches. Inversely, if an 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 developed for the Intel architecture are not directly applicable to that platform and vice-versa.

MICROARCHITECTURAL ATTACKS

The cache attacks are a well-known attack of the Microarchitectural side-channel attacks class. They are defined by Aciiçmez (5) 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, secure and insecure processes can interact through their shared use of the cache.

This allows an attacker to craft a “spy” program that can infer the internal state of a secure process. Results have shown how the cache side-channel can be used to recover AES keys (6, 7), RSA keys (8), or even allow one virtual machine to compromise another virtual machine running on the same host.

Another type of Microarchitectural side-channel attacks is covert channels. These attacks use the CPU cache to allow covert communications between different process and different VMs running on the same physical host. The cross-VM covert channels have been the focus of a lot of recent work, going from Ristenpart et al. (9) and their cache-based covert channel between two Amazon EC2 instances at a rate of 0.2bps, to the work of Maurice et al. (3) with an error-free covert channel between two Amazon EC2 instances of more than 360Kbps, allowing the building of an SSH connection through the cache.

CACHE ATTACKS

A JAVASCRIPT EXAMPLE

In the paper “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”, by Oren et al. (1), we are presented with a demonstration of “how a micro-architectural, side-channel cache attack, can be effectively launched from an untrusted webpage”. The authors explore, in this paper, a way to implement a side-channel cache attack in JavaScript with the goal of spying on the user’s behaviour.

PRIME+PROBE IN JAVASCRIPT

In modern browsers, JavaScript programs are executed by default when delivered by a webpage. To avoid potential damages caused by this property, JavaScript code runs in a sandboxed environment which severely restrict its access to the system. For example, it cannot execute native code or load native libraries. Another problem faced with JavaScript is that it has no notion of pointers, which make it impossible to determine the virtual address of a JavaScript variable.

There are four steps involved in a successful Prime+Probe attack:

- Creating an eviction set for one or more relevant cache sets;
- Priming the cache set;
- Triggering the victim operation;
- Probing the cache set again.

CREATING AN EVICTION SET

An eviction set is a “sequence of variables (data) that are all mapped by the CPU into a cache set that is also used by the victim process”. As JavaScript has no notion of pointers, it is arduous to provide a deterministic mapping of memory address to cache sets. To create an eviction set, you could “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”. However, as they explain in the paper, it would take an impractical amount of time to do so, thus rendering this attack useless.

To counter this problem, they came up with 2 optimisations for their algorithm:

- Starting with a subset containing all 131K possible addresses and attempt to shrink it by removing random elements and checking that the access latency of the victim address stays high.
- if physical addresses P1 and P2 share a cache set, then for any value v, physical addresses P1 XOR v and P2 XOR v also share a (possibly different) cache set.

With these optimisations they were able to gradually create eviction sets for most of the cache, except for the parts accessed by the JavaScript runtime.

PRIMING AND PROBING

Once the attacker finds an eviction set, the next goal is to replace all entries in the cache of the CPU with the elements of this eviction set. For the probe step, the attacker also has the goal of precisely measuring the time required to perform this operation. To perform these steps, they had to ensure the in-order execution of critical code parts by accessing the buffer as a linked list, and avoid the stride prefetching feature by randomly permute the order of elements in the eviction set and access it in alternating directions.

IDENTIFYING INTERESTING CACHE REGIONS

Since the eviction set created by the JavaScript code is non-canonical, they had to correlate the profiled cache sets to data or code locations belonging to the victim. To do this, they used machine learning methods to derive meaning from the output of the cache latency measurements.

TIMERS AND HOW TO CREATE THEM

As a response to the attacks shown in the first paper, through JavaScript, the W3C and browsers vendors eliminated fine-grained timers from JavaScript. This renders previous high-resolution Microarchitectural attacks non-applicable. However, we learn in the paper “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”, by Schwarz et al. (2), that this solution is not the right one as there are other ways to find or create new timers. They demonstrate, in this paper, “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”.

RECOVERING A HIGH-RESOLUTION TIMER

We know from previous papers that it is possible to recover a high resolution timer by observing the clock edges. They represent the time at which the timestamp is an exact multiple of its resolution.

As the underlying clock source has a high resolution, the time between two clock edges varies only as much as the underlying clock. That means that the time between two edges is always constant, which gives us an exact time base to build upon. With that in mind, we can interpolate the time between clock edges to get an accurate timestamp.

This technique, clock interpolation, needs a calibration before it can be used to return accurate timestamps. To do that, “we repeatedly use a busy-wait loop to increment a counter between two clock edges”. This gives us the number of steps we can use for the interpolation. The time it takes to increment the counter once is the resolution we can recover. That resolution can be approximated by dividing the time difference of two clock edges by the number of incrementations of the counter we were able to do.

To use this timer we simply busy-wait until we observe a clock edge, we then start the operation we want to time, and then, after our timed operation has ended, we busy-wait for the next clock edge while incrementing the counter. Multiplying the number of incrementations by the time it takes to make one gives us the interpolated time. Adding this time to the measured one increases the timer’s resolution again.

This method allows us to recover a high-resolution timer, for example, “for a timer rounded down to a multiple of 100 ms, we recover a resolution of 15 µs”.

ALTERNATIVE TIMING PRIMITIVES

When we cannot use the High Resolution Time API, like on Tor browser, we need to find other timing primitives. As JavaScript is mono-threaded and doesn’t have true concurrency, we need to either base our primitive on recurring events or non-JavaScript browser features.

HOW TO COMMUNICATE BETWEEN PROCESSES WITH SIDE-CHANNEL ATTACKS

A covert channel is an unauthorized communication between a sender and a receiver. It can be used to communicate between different processes and even between different virtual machines. It uses the difference in latency for memory accesses, whether the data is cached or not. Caches are used to make covert channels between VMs as they are not virtualized, and thus, shared across VMs on the same physical host. They also are shared across the cores of a CPU and coherent between CPUs of the same machine, which makes it possible to have a cross-core or cross-CPU covert channel.

In the paper “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”, by Maurice et al. (3), we learn about how to get a sustainable SSH connection from a covert channel. To do this, they characterized noise on cache covert channel, propose a protocol to handle the physical layer of the robust cache covert channel, evaluate their assumptions and demonstrate an SSH connection between two virtual machines on Amazon EC2.

MEASUREMENT ERRORS

As the cache is a shared resource with other programs and is small, it is prone to evict data from our covert channel attack. With the sender using eviction as a way to send bits to the receiver, a heavy use of the cache can create interferences on our covert channel as false positives. If the eviction is not successful, it can also create false negatives. These errors are called substitution errors.

Another type of errors come from the fact that the sender and the receiver have no way of telling the other if they are transmitting data, as they are not always running synchronously. This results in insertion and deletion errors on the receiver’s end.

These synchronisation errors can be prevented by creating a common clock for both ends of the covert channel.

ROBUST CACHE COVERT CHANNELS

For this robust covert channel, they developed a communication protocol, on the physical and then on the data-link layer. This protocol is as follows: the data-link layer gets a buffer of data to be transmitted, this data is then divided into equally sized chunks to which we had error correction bits, resulting in a packet. After that, each packet is divided into small enough words to be transmittable in between scheduling, before sending these words, we encode them to prevent synchronization errors.

On the physical layer, to send the words to the receiver, the sender needs to prime the last-level cache enough so that the receiver’s lines are evicted from its L1 cache. Depending on how many lines of the receiver’s L1 cache were evicted, it can infer the transmitted symbol, whether a ‘0’ or a ‘1’. The sender does not need to evict all lines of the last-level cache, as the L1 cache has fewer lines. However, it cannot know which lines of the last-level cache are present in the L1 cache of the receiver. Because of this, in practice, the optimal number of lines primed by the sender is all the lines from the L1 cache in the last-level cache.

To receive the information, the receiver probes its lines in the L1 cache. It needs to probe at least one line to get data, but can probe all the lines of the L1 cache to achieve the maximum bits per symbol, with a bigger risk of having interferences with lines being evicted by other programs.

CONCLUSION

Side-channel attacks present a huge yet almost non addressed threat. At first they were hard to use because you needed to get a physical access to the victim's machine and run a program locally on that machine, but since 2015 and the work of Oren et al. (1), we know how to perform them from a remote PC and on common user PCs through the use of JavaScript programs and timers.

This lead to a new interest for these attacks which resulted in a quick development of the field, as shown by the huge amount of paper published recently on cross-VMs covert channels.

Browsers vendors are trying to find a good solution to prevent these timer based side channel attacks, but the quick fix they tried did not work as planned and there is a high probability that whatever they do, attackers will always find a way to circumvent it and create other clever ways to get accurate enough timers to be able to continue attacking user's systems.

BIBLIOGRAPHY

(1) Y. Oren, V. P. Kemerlis, S. Sethumadhavan, A. D. Keromytis, The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications, October 2015.

(2) M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript, 2017.

(3) C. Maurice, M. Weber, D. Gruss, C. A. Boano, M. Schwarz, L. Giner, K. Römer, S. Mangard, Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud, In: NDSS'17 (2017), March 2017.

(4) P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of CRYPTO*, 1996.

(5) O. Aciiçmez. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proc. of ACM CSAW*, 2007.

(6) D. J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming, April 2005. [Online; accessed November-2017].

(7) 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.

(8) C. Percival. Cache Missing for Fun and Profit. In *Proc. of BSDCan*, 2005.

(9) 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 CCS'09, 2009.