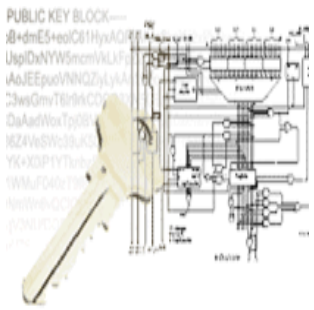
Graz University of Technology
IAIK
Institute for Applied Information Processing and Communications
Inffeldgasse 16a
A–8010 GRAZ

Bachelor Thesis



Cache.pdf: Cache Covert Channels using JavaScript

Version 1.0

Christian Gaisl

Cache.pdf: Cache Covert Channels using JavaScript

Christian Gaisl

November 23, 2017

Abstract

Covert channels are a way of communication between multiple parties that evade isolation mechanisms and are hard to detect. Cache covert channels use the CPU cache as a shared medium in order to establish a communication between two processes running on the same machine. They can evade sandboxing isolation mechanisms imposed by cloud virtualization, app permission systems, browser windows and many more. However, most existing channels require that both the sender and the receiver run native code.

In this paper, we show that it is possible to utilize cache covert channels using only JavaScript inside a modern web browser. We demonstrate that it is possible to send messages over a cache covert channel from basically any execution environment. Our covert channel even allows us to send messages from a PDF file and receive them using only a web browser on the receiving end. Our demonstration widens the applicability of this kind of attack greatly.

Contents

1 Introduction

Covert channels are unauthorized communication channels between two processes, that are supposed to be isolated from each other. There are instances where a process should not be able to communicate with another process for security reasons. The concept of sandboxing is a very widespread security mechanism, typically used in order to be able to run untrusted code without jeopardizing the security of the entire machine. Web browsers heavily use that concept, with the Google Chrome browser even instantiating each tab as a separate process. Security policies of virtual machines running on the same machine build on the assumption that they are isolated. A covert channel allows isolated parties to evade this sandboxing and communicate with each other, possibly exfiltrating sensitive data. Such a channel could also be used as a confirmation that the two parties are in eachs others proximity. An example for that would be the the confirmation of co-residency of 2 virtual machines running on the same machine in the cloud [1], which opens the possibility for further attacks.

Another motivation behind covert channels are collusion attacks [2]. Mobile operating systems implement an app permission system, which restricts each app from certain actions. In order to enforce this permission system, each app has to be isolated from each other. However, if unprivileged apps can communicate with each other over a covert channel, they can indirectly escalate their privileges by offloading the privileged task to another app, which may have different permissions.

Side channel attacks are a way to gather information from systems that unintentionally leak some information through the hardware or software implementation. A given device may involuntarily give away information through its timing, power consumption, electromagnetic leaks or even sound. An attacker could, for example, measure the power consumption of an encryption device in order to gain necessary information to break the encryption [3]. If side channels can be controlled, they can be used to create a covert channel.

CPU caches are a form of fast memory located directly inside the CPU that is used by every process run by that CPU. Every time the CPU fetches data from the main memory, it stores it in the cache in order to speed up subsequent accesses to that data. The time it takes to fetch data from main memory is significantly higher than from the cache. Cache side channel attacks abuse these timing differences that arise by accessing data that is either in the cache or in main memory. Cache covert channels, such as the ones described by Maurice et al. [4], use these cache timing differences in order to establish a high performance communication channel that is capable of a transmission speed of around 46 KBps.

If one has access to physical memory addresses, it is possible to target specific parts of the cache [5] using physical memory addresses. However, many programming languages, like JavaScript, do not even have access to virtual addresses.

In this paper, we want to give a basic overview of cache-based covert channels and show that it is possible to use a cache covert channel from environments which do not provide neither physical nor virtual addresses and do not provide high precision timers.

In section 2, a basic overview over CPU caches in Intel CPUs is given. It also describes current cache side channel attacks and how they work and how we used these techniques in order to build our cache covert channel

Section 3 of this paper will show that it is possible to apply cache attack techniques outside of native code. We describe how to deal with the limitations imposed by JavaScript running in a standard Web browser and created a cache covert channel running inside a browser. Furthermore, the PDF standard allows for embedding a restricted form of JavaScript inside a PDF file. We show that it is possible to utilize cache covert channels even from a PDF that is being viewed by a typical PDF viewer. The techniques used in that process are not limited to PDFs alone but to a wide range of execution environments and programming languages.

The results of our research implicate, that it is possible to utilize cache covert channels not only with native code, but from basically any execution environment, including JavaScript in webbrowsers and JavaScript in PDFs. This opens the possibility to evade many isolation mechanisms, such as the same origin policy in web browsers, cloud virtualization or app permission systems without having to run native code.

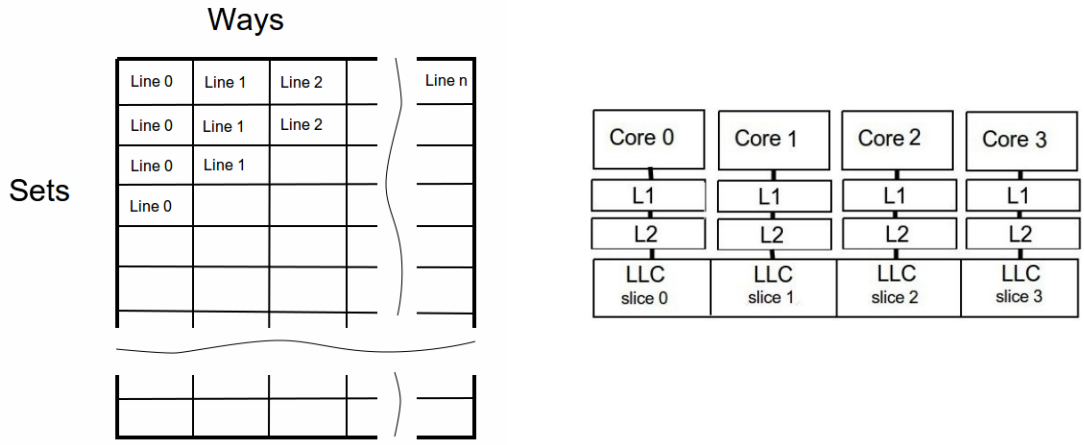
2 Background

2.1 CPU Caches

CPU speeds have increased greatly over the years and have long outrun the speed of typical main memory. In order for the CPU to not be bottlenecked by how fast it is able to access the memory, CPU caches were implemented. CPU caches are a type of small and fast memory that is significantly faster than typical main memory. It is used to store frequently used data in order to reduce the number of accesses to the main memory, which speeds up the system significantly.

The size of typical main memory today is in the order of tens of gigabytes but because the price of SRAM used in CPU caches is significantly more expensive, the size of a typical CPU cache is usually only in the order of a few megabytes. Broadly speaking, there is a direct relation between speed and cost of memory chips. Faster memory chips are simply more expensive to produce. Because of this, typical Intel consumer CPUs use 3 layers of cache, with the first layer being the fastest and smallest and the third layer being the largest and slowest. The first layer is called L1, the second L2 and the third is called L3 or LLC, which stands for "Last Level Cache" [6].

As the size of the LLC cache is in the order of megabytes, it is impractical to search through the whole cache in order to check if a specific variable is currently in the cache or not. The time it would take to search the whole cache every time a variable is accessed would diminish the point of the cache. That is why modern Intel CPUs are "set-associative" [6], which means, that they divide their cache into a fixed number of sets. Those sets contain a fixed number of ways that contain the actual cache lines which store the actual data. As shown in Figure 1, we think about this like a two-dimensional array with the height being



(a) Architecture of a Cache. Physical addresses map to a single cache set and data only ever gets stored in one of the lines of that particular cache set.

(b) Cache Architecture of a Quad Core Intel CPU. Each Core has its own L1 and L2 cache and the LLC is shared between all Cores.

Figure 1: Cache Architecture of an Intel CPU

equal to the number of cache sets and the width equal to the number of cache ways. Each piece of data is mapped to only one set, which is determined by the physical address of that data. This means, that if it is accessed from the main memory, it can only be cached in one of the cache lines of that particular cache set. Because of that, the CPU only has to search through that one set in order to determine if the data is already cached or not.

L1 cache typically has 8 ways, while the number of ways in the LLC cache varies between 12 to 20 ways, depending on the specific CPU model [6].

On modern multi-core Intel CPUs, each core has its own L1 and L2 cache, meaning that a quad-core Intel CPU has four L1 and L2 caches. The LLC cache is a special case because it is shared between the cores. Each core owns a so-called slice of the LLC cache where it chooses to store its own data, but each core can also read and write from the slices of the other cores without a significant performance penalty. That being the case, one can view the LLC cache as a shared resource between the cores.

Having multiple cores operating at the same time can lead to a situation where data, that is stored inside the cache, is simultaneously manipulated by another core. In this case, the data in the cache does not represent the actual data in the main memory anymore. CPUs vendors implemented cache concurrency protocols in order to avoid such situations. Different vendors have implemented different protocols. Designing cache concurrency protocols for performance is very difficult and are an ongoing research topic [7]. The following section will give a brief overview of some of the implementation details of modern Intel consumer CPUs from Ivy Bridge to the most recent Kaby Lake architecture [6]. It will

only describe the parts that are relevant for our implementations of cache attacks.

If a CPU core is searching for data, it starts looking for it first in it's L1 cache and then it's L2 and finally in the LLC. If some requested data is already stored in the LLC slice of another core, it will not be duplicated in the LLC slice of the requesting core, but the core simply uses the existing data from the other slice.

Each level of cache is inclusive, which means that everything that is stored in L1 or L2 cache is also stored in every higher level of cache. This is not a huge efficiency problem since with every higher level of cache the size of it also increases greatly, meaning that storing the entire L1 cache in the L2 cache only takes up 25% of the L2 on Intel Kaby Lake CPUs. This property is strictly maintained by the CPU.

What this entails, is that if a line is deleted in the LLC cache, it is also deleted in the L2 and L1 caches. Because the LLC is shared between the cores it is possible for a core to delete a line in the L1 and L2 cache of another core by deleting that very same line from the LLC. Targeting the LLC makes a cache attack independent of the core that it is running at. This property will be used extensively in the implementation of our cache attacks.

2.2 Cache Attacks

The main purpose of a CPU cache is to speed up data accesses by caching frequently used data, to reduce the amount of accesses to the main memory. Accessing data from main memory (cache miss) takes a lot more time than accessing data from the cache (cache hit). This difference in timing is the key component of most cache attacks. Using these timing differences the attacker can find out which cache sets get accessed by the victim. This can in turn be used to gain more information about the victim's processes, like finding out which website he is visiting [8] or partially logging his keystrokes [9].

There are 2 basic types of cache attacks, depending on whether the attacker shares memory with the victim process or not. The following attack descriptions assume that the attacker shares data with its victim, e.g., by using a shared library or with memory deduplication active.

Flush+Reload This attack, first defined by Yarom et al.[10], builds on the fact that the attacker is sharing memory with the victim. An example would be the attacker and the victim using the same shared library. This means, that if both the attacker or the victim access a part of that library, it will only be cached once. The cached data is therefore also shared. The attacker selectively uses the *clflush* instruction on certain parts of the shared memory, which removes them from the cache. The attacker then measures the time it takes to access the data. If it takes a long time, the attacker knows that the data is loaded from main memory. If, however, the victim had accessed the data in the meantime, then it would already be in the cache and would therefore be fast to access. With this, the attacker can determine which parts of the shared library the victim is accessing. This information can be used to extract private encryption keys.

Evict+Reload Gruss et al. [9] proposed to replace the *clflush* instruction in Flush+Reload by evicting the target cache set.. Eviction works by filling a cache set with different data and therefore replacing all data in a target cache set, eliminating the need for *clflush*. The *clflush* instruction is not directly accessible in all programming languages and therefore the use of Flush+Reload is restricted. An eviction function can be implemented in higher level languages, like JavaScript.

Flush+Flush This attack, defined by Gruss et al. [11], uses the fact, that the *clflush* instruction takes a different time to execute depending on whether the data to flush is present in the cache or not. If the data is not cached the instruction executes measurably faster than if the data has to be deleted from the cache. Using this principle, the attacker can determine if a victim has loaded a part of the shared memory by using the *clflush* instruction on that part and measuring the time it takes to execute the instruction. This has the advantage that this attack does not cause cache misses, which leads to better performance and is harder to detect.

If the attacker does not share any memory with the victim, he can still determine which cache sets are accessed by the victim process:

Prime+Probe If the attacker can find out to what cache set his data maps to he can proceed to fill a specific cache set with his own data (Prime) and then measure the time it takes to access that very same data (Probe). If no other process has accessed that particular cache set in the meantime, then the time it takes to access that data will be low. If, however, a victim process had accessed data that maps to the same cache set then at least one of the cache lines would be replaced by victim data. This would mean that the attacker would have to access the main memory during the probe step, which takes a considerable amount of time[12] [13]. With this, the attacker can determine which cache sets the victim is using. This information alone is enough to determine victim behaviour, like finding out which websites the user is visiting [8].

Prime+Abort All previously described cache attacks rely on a high precision timing source in order to be able to differentiate cache hits and cache misses. Prime+Abort from Disselkoen et. al. [14] works similar to Prime+Probe but instead of relying on a timing difference, to determine whether the target cache set has been accessed, it uses the Intel TSX hardware provided in many server and consumer grade Intel CPUs. The attacker can put his code inside a TSX transaction, which reports an abort signal when the victim accessed the targeted cache set.

2.3 Covert Channels

Covert channels are unauthorized communication channels between two processes, which do not have the necessary permissions to communicate with each other, or want their communication to be hidden by using unexpected and unconventional ways.

The motivation behind covert channels is to evade isolation mechanisms, that try to prevent processes from communicating with each other. Many systems are designed to be isolated and their security assumptions build on that assumption. An example for such an isolation mechanism would be operating system virtualization. Two virtual machines running on the same machine should be isolated from each other, but using a covert channel they are able to communicate with each other without authorization. One attack scenario would be a virtual machine, without network access, sending data to the host operating system, which can then forward that data over the internet.

The Google Chrome browser creates a new process for every browser tab that is opened in order to isolate them from each other. Using covert channels they are still able to communicate with each other. Many mobile operating systems implemented an app permission system in order to restrain the capabilities of particular apps. A malicious app could utilize covert channels in order to communicate with another app that may have more permissions that itself and therefore circumvent that permission system.

In order to communicate one needs a shared medium that both communication partners can observe. An example of a such a shared medium would be the volume level or screen brightness of a device [15]. The sender could increase or lower the volume of the device and another process on the same device could interpret this as '0's and '1's. This concept is not limited to just computers but is also applicable in the real world. One could, for instance, turn on the lights of a building in a particular interval and an observer could interpret that as a message. Brent [16] described Air-Gap covert channels as a distinct category of channels that try to establish a communication between systems that are physically and electronically separated from another. This is done by using sound [17], radio frequency [18], hard-drive LEDs [19] and many more.

Regardless of what shared medium is found, for a covert channel to work the sender has to be able manipulate it in a way that the receiver can observe it.

The sender of a covert channel can encode his message into binary bits and manipulate that shared medium. The receiver can then decipher the message by the use of some agreed to protocol.

Cache covert channels work by using the cache as a shared medium for communication. It is not possible for a process to simply read the contents of the cache from another process, but it is possible to utilize existing cache attacks to create a cache covert channel. As shown in Figure 2, the time it takes to access a simple Integer depends on whether it is already cached or not and these access times are easily distinguishable.

The receiver of a cache covert channel can utilize the Prime+Probe method. He puts some data inside an agreed to cache set and then measures the time it takes to access that data. The sender can access some data, that maps to the same cache set, in order to displace some of the receivers data. This induces a higher timing on the receivers side.

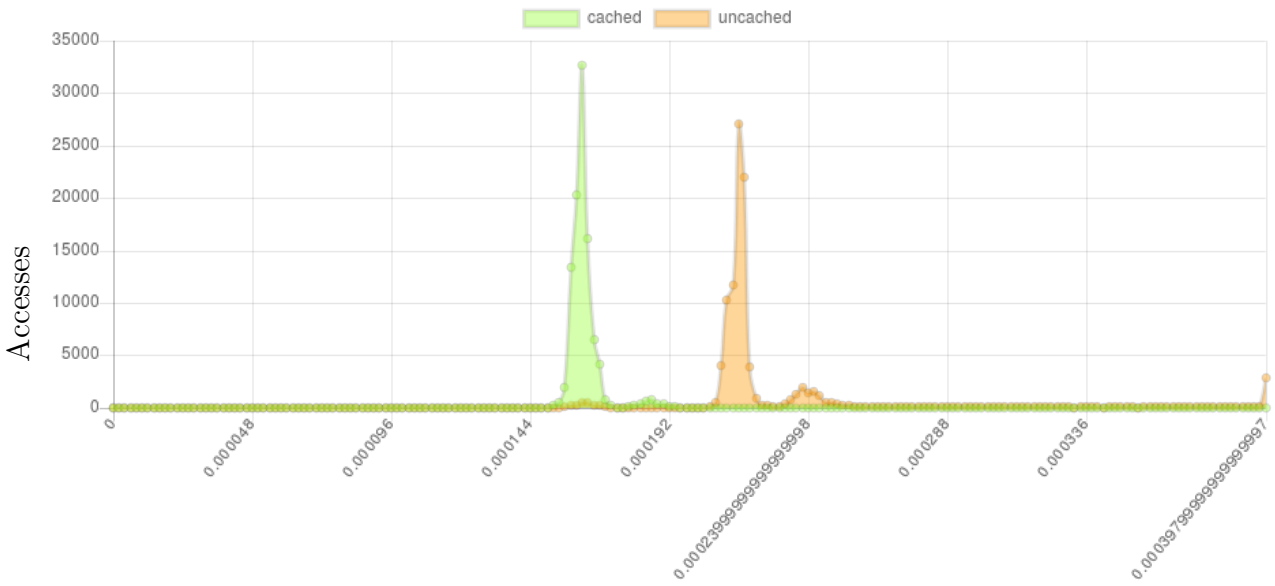


Figure 2: Histogramm of of access timing to cached and uncached data. The timings are easily distinguishable.

With that, the sender can encode his data into high and low timings by either accessing that cache set or not, representing '1's and '0's.

Because the cache is physical and shared by every process on the same CPU, it can be used for communication between any two processes on the same machine, without any permissions or network access, even if they are on different virtual machines [4].

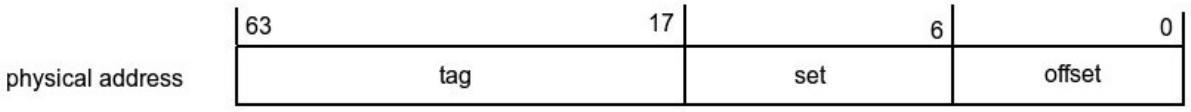


Figure 3: Complex Adressing Scheme for LLC

2.4 Prime+Probe

In order for a Prime+Probe attack to work one needs to be able to completely fill up a specific cache set with data that can be retrieved in the probe step. Because the data must be retrieved again, it is not sufficient to just fill the cache set with random data by accessing a huge array and hoping that it will fill the cache set by sheer volume. There are a few key problems to solve in order to be able to create a covert channel using this method:

- The receiver needs a way to target a specific cache set.
- The receiver needs a way to reliably fill that cache set with own data.
- The receiver needs to have a way to distinguish between cache hits and cache misses.
- The sender and receiver need a way to agree on specific cache sets.

2.4.1 Eviction sets

Every physical address in the main memory is mapped to a single cache set. In order to be able to fill a cache set with data, one needs to create a set of data, with a size of at least the number of cache ways, that maps to a single cache set. Such a set is called an eviction set.

The mapping function that maps a physical address to a specific cache set, inside a specific cache slice in the LLC, has been reverse engineered [5]. As shown in Figure 3, the 11 bits from bit 6 until bit 17 (starting from the LSB) determine which cache set a specific physical address maps to and a XOR function with specific bits, depending on the core count, determines which cache slice that address maps to.

One could build an eviction set for the L1 or L2 cache, but modern CPUs have multiple cores most of the time. It makes more sense to build an eviction set for the LLC since it is shared between the CPU cores. This makes it more versatile because it can be used cross-core.

In order to create an eviction set for a single cache set, one can create a large array of variables and search for those that map to the desired cache set.

In Native Code

In C code the virtual addresses of variables are known. With the Linux provided "/proc/self/pagemap" one can retrieve the physical addresses and calculate which

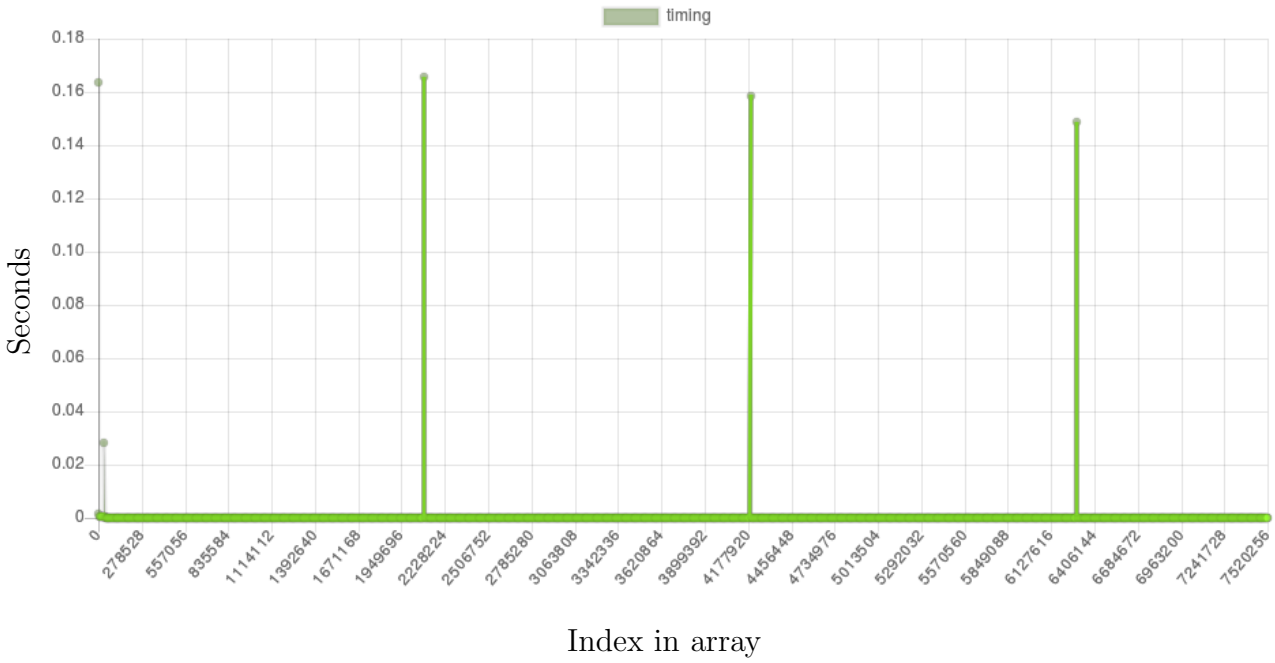


Figure 4: Time it takes to access data, when sequentially accessing a large typed Array with JavaScript. Creating a new 2MB page takes a considerable amount of time, which is easily identifiable on the chart. Those timing spikes indicate the beginning of a new page.

cache set and slice that variable maps to, using the complex addressing function reverse engineered by Maurice et al. [5].

In Web browsers using JavaScript

JavaScript hides the virtual addresses of variables to the programmer and has no access to physical addresses. In order to retrieve the physical addresses, we had to rely on a different timing side-channel, based on a similar timing attack performed by Gruss et al. [20]. When you allocate a large typed array in Linux with Firefox or Google Chrome, the browser allocates large typed 1MB aligned arrays and uses anonymous 2 MB pages when possible. Using that knowledge, we created a large uint8 typed array in JavaScript and accessed it iteratively and measured the time it took to access data inside the array. Once a memory page is filled, a new data access induces a page fault by the operating system that creates a new 2MB Page. As can be seen on Figure 4 that takes a comparatively long time and is easily measurable even in browsers with a reduced timer resolution. We now know that the data that induced that page fault is at the start of a new 2MB page and therefore has a known page offset. In the case of 2MB pages the last 21 Bits of the physical address represent the page offset, and therefore we know the last 21 Bits of the physical address of that variable. We used it as a point of reference to calculate the last 21 Bits of the other variables in the array, by using the known data offset of the typed array.

The last 21 Bits of the physical address are enough to calculate which cache set that variable maps to, but not enough to calculate which cache slice that cache set belongs to. In order to create an eviction set that targets a single cache set in a single slice we needed to add an extra step. We created an array, which includes a number of variables that map to the same set, and iteratively removed the variables that did not affect the first variable in that array using the following algorithm, based on the one described by Oren et al.[8]. We wanted to create an eviction set of size 12, our test CPU, an Intel i5-4200U, has a 12 way LLC. We first accessed the first variable in that array and then tried to evict it from the cache using the rest of the array as an eviction set. If the eviction was successful, we removed one variable from that array and tried again. If the eviction became unsuccessful after an iteration, we put the most recently removed variable back into the eviction set. We repeated this until the array reached the desired size.

MakeEvictionSet **Input :** *arr* : int[] //an array of indices to the large typed array
while *arr.length* > 13 **do**
| (a) access variable with the index of arr[0]
| (b) remove an index from arr
| (c) prime cache set with variables indexed by arr[1] and up
| (d) check if arr[0] is still cached, if yes put the previously removed variable back into the array
end
(e) remove a random variable from arr to get an eviction set of size 12

Algorithm 1: Algorithm that reduces an Array of data that maps to the same cache set but multiple slices to an Array that maps to only one cache set in one slice

Using this algorithm ensures that all variables inside the eviction set map to the same cache slice and set, but it is important to mention that the exact cache slice is still unknown.

If an address is cached or not can be determined by measuring the access time with the "performance.now()" function. In more recent versions of popular browsers the resolution of the "performance.now()" function has been reduced so that it is no longer easily possible to distinguish between a cache hit or miss. However, Schwarz et al. [21] have shown that it is possible to create a timer in the latest versions of popular Web Browsers with a resolution that is adequate for this task.

In environments without a high resolution timer

In order to distinguish between cache hits and cache misses, the programming environment has to have access to a high precision timer. Many environments, such as JavaScript in the most recent browser versions, do not provide a timer that has a resolution that is high enough to do that. However, as shown by Schwarz et al. [21], it is possible to create such a timer. One prerequisite for building such a timer is

having a way to perform asynchronous computing, in the form of threads for instance. If a given environment does not provide that, it is impossible for a process to create a timer that is able to measure the execution time of its own functions, because the timer and the function need to run in parallel.

An example for such a restricted environment is the JavaScript API that modern PDF viewers provide. We tried to create an eviction set in this environment but that API does not provide the "performance.now()" function or any other timing function that would provide the necessary accuracy. The most accurate timing primitive that is available is "Date.now()" which is accurate up to 1ms. This is not enough to differentiate between cached and uncached data. Using the techniques described by Schwarz et al. [21], in order to get more precision out of this function, did not result in a timer that is accurate enough.

JavaScript in PDFs does not provide a way for asynchronous computation. This means that, while it may be possible to create a timer with an accuracy of more than 1ms, this timer will not run in parallel to other actions in the same process. Without an accurate timer it is not possible to utilize cache attack techniques from such an environment.

We also tried modifying the PDF viewer "mupdf" in a way that the Date.now() function returns the *rtdsc* counter instead of ms. We found out that the overhead of interpreting JavaScript, using the JavaScript engine that mupdf uses, is orders of magnitude greater than the timing we were looking for. This makes the variance between the returned rtdsc values greater than the differences between cached and uncached data and therefore this approach is unusable for out purposes.

We further investigated whether this overhead also exists in other PDF viewers and constructed a simple test to see if a particular PDF viewer would provide the necessary performance. We created 4 integer variables and incremented each one of them one by one for a whole second to give us an estimation of the overhead of a particular PDF viewer. We found out, that out of the tested PDF viewers, only the one provided by the Chrome browser would be fast enough to create eviction sets, given that one could somehow get access to a high-resolution timer. This is most likely because it uses a similar JavaScript engine as the browser itself, Chrome V8. We did succeed to create eviction sets using the "performance.now()" function provided by the Firefox browser version shown in the table, as a point of reference.

Because manipulating the PDF viewer of a browser of a victim in order to create a cache covert channel is not a viable attack scenario, we did not investigate any further.

	mupdf 1.11	Acrobat Reader 2017.009.20058	Chrome PDF viewer	Chrome 59.0.3071.109	Firefox 34.0.5
global variable	2,372,790	282,347	16,113,269	16,191,227	11,307,132
global array	1,300,864	446,840	15,312,179	16,075,614	9,899,509
local variable	3,198,366	540,567	15,919,770	16,435,318	13,029,964
local array	1,459,943	466,377	15,737,726	16,104,796	10,492,527

Table 1: JavaScript performance estimation in different execution environments. Value represents the final value after incrementing a Number, starting from 0, for 1 second.

2.4.2 Prime

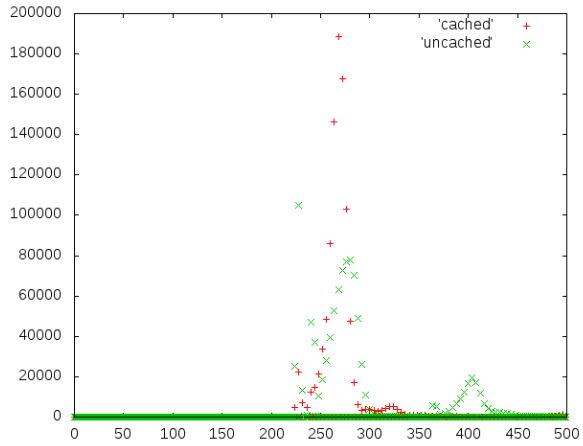
In the Prime step one tries to fill a target cache set with own data. Using an eviction set with the same size as the number of cache ways, one can fill a specific cache set with that data, considering the cache replacement policy of the specific CPU one is working with.

The cache replacement policy decides what data gets to stay inside the cache and what gets replaced if the cache set is full. Older Intel CPUs used a pseudo least recently used (LRU) replacement policy [22], meaning that if a cache set is full, the least recently used cache line is replaced by the new entry. With a LRU replacement policy in place, one could just access the whole eviction set once, in order to fill the whole cache set. However, newer Intel CPUs often use an undocumented adaptive replacement policy, which makes replacing the whole content of the cache set more difficult. Simply accessing the whole eviction set once could potentially lead to a situation where only a fraction of the data in the eviction set is actually inside the cache set, because its own entries were replaced by itself. If the cache set is not in a known state, one can not infer any information with access timings.

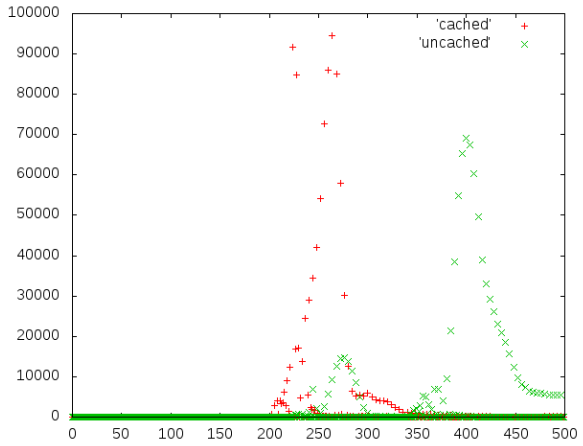
Adapting the access patterns described by Gruss et al. [23] we created an access pattern that replaces the content of a cache set with the contents of the eviction set in an acceptable amount of time. Using that access pattern works, but there may be room for improvement regarding performance. If one is able to fill the cache set in a shorter period of time, the performance of the cache covert channel could potentially be improved.

Being able to fill a cache set with data also means being able to evict any other data that was present in that cache set before. Because of the inclusive property of Intel CPU caches, data that gets evicted from the LLC also gets evicted from the L1 and L2 caches of the respective core. To verify this algorithm, we accessed some data to cache it and then tried to prime the cache set, that this data maps to, with different data. If we were to successfully fill that cache set with only our data, all previous data should be evicted from that cache set and should take longer to access.

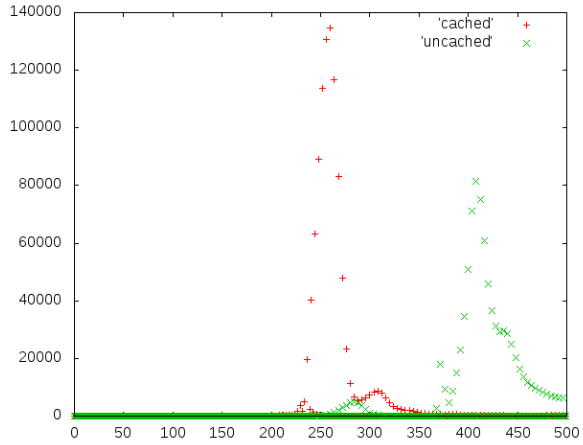
We measured the time it takes to access the initial data and compared it to the access timings of simply iteratively accessing it, which means that is constantly cached with a high probability. As shown in Figure 5, we found out that on our machine, running an Intel i5-4200U dual core Haswell, 4 runs were adequate to fully prime a 12 way cache set with a high probability.



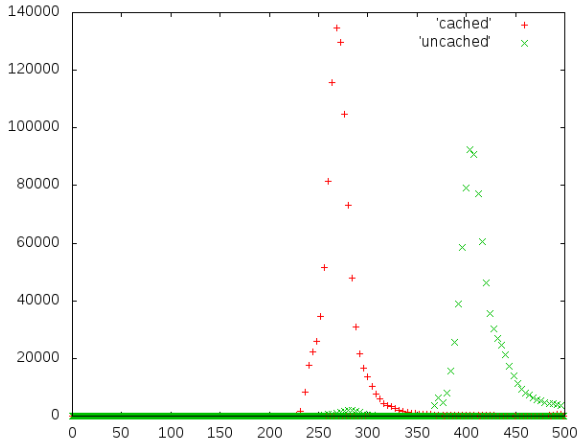
(a) runs = 1, eviction rate 10%



(b) runs = 2, eviction rate 84%



(c) runs = 3, eviction rate 96%



(d) runs = 4, eviction rate 99+%

Figure 5: Comparing access times to data in a cache set before and after using our prime algorithm on the same cache set. 4 runs of our prime function ensure an acceptable eviction rate. Having evicted all data in the cache set implies that the cache set is successfully filled with the data from our eviction set.

```
prime (runs, nr_ways, addr);
Input  : runs : int, nr_ways : int, addr : int[nr_ways]
for i = 0 to runs do
    for j = 0 to nr_ways - 1 do
        *addr[i];
         *addr[i+1];
         *addr[i];
         *addr[i+1];
    end
end
```

Algorithm 2: Prime step. Takes an eviction set with a size of the number of cache ways and fills the cache set with the data inside the eviction set.

2.4.3 Probe

Once the targeted cache set is in a known state, filled with the data pointed to by the eviction set, the attacker has to measure the time it takes to access that data. If no other process on the machine has accessed that specific cache set during that time, then all of the variables in the eviction set should still be inside that cache set and should therefore be fast to retrieve. If, however, another process or the sender has accessed that particular cache set, then at least one of the cache lines inside that set has been replaced by something else. Therefore the time it takes for the sender to retrieve his data rises significantly. This is because at least one part of the eviction set needs to be loaded from main memory again. When that part gets loaded into the cache set again, it potentially replaces another part of the eviction set that has not yet been accessed within the probe step. That part then also has to be loaded from main memory again. This process makes it very easy to distinguish whether or not that cache set has been accessed by a different process, using the timing it takes for the probe step to complete.

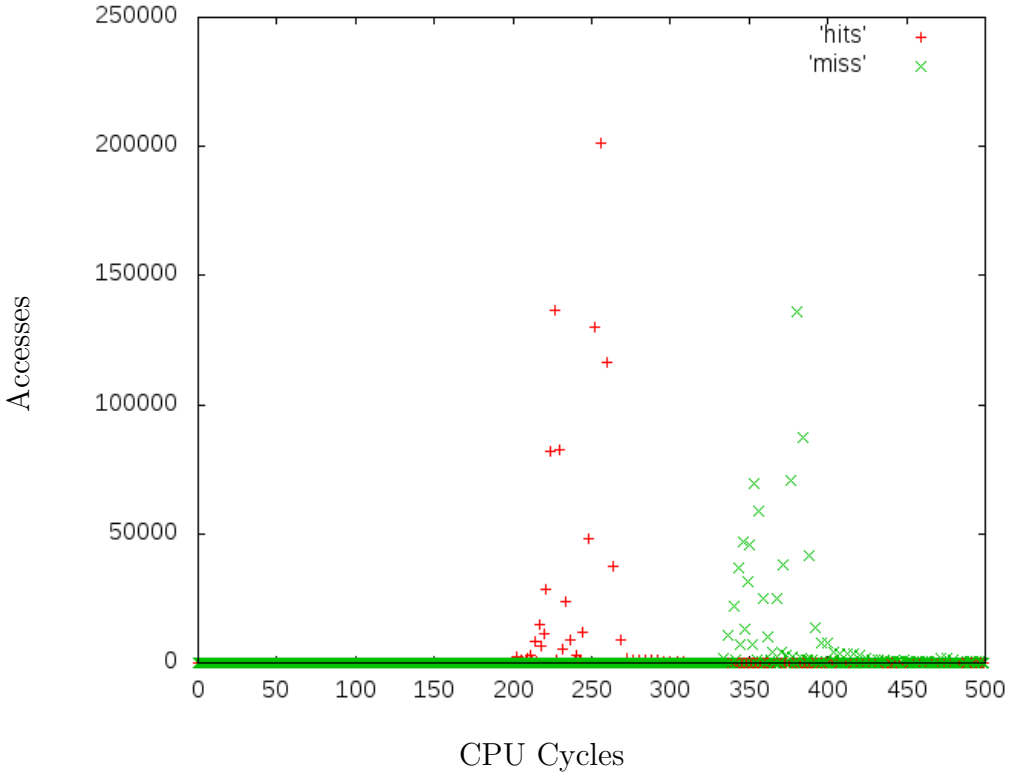


Figure 6: Histogram of access timings to data from a fully cached eviction set compared to accessing an only partially cached eviction set.

3 Cache Covert Channels in Restricted Environments

To build our cache covert channel, we decided to implement a simple transmission scheme that is portable across multiple execution environments. We use Prime+Probe as a foundation for our cache covert channel.

Receiver

: The receiver primes and probes a single cache set and registers whether it was a cache hit or miss. The receiver repeats this over the timespan of a millisecond, using a real-time clock. Depending on whether the majority of tries were a cache hit or miss, this millisecond will be treated as either a '0' or '1' respectively.

Sender

: The sender converts his message into binary bits and sends 1 bit per millisecond. If the bit to send is a '1', he continuously accesses a variable that maps to the agreed cache set during that millisecond. If the bit is a '0', the sender does nothing during the course of that millisecond.

A visualization of this protocol can be seen in Figure 7. This transmission scheme is naturally limited to 1000 bits per second and does not scale with a faster machine. It's advantage is, that it is easy to implement in a wide range of execution environments and that it only targets a single cache set, which makes agreeing on a target cache set easier.

3.1 Covert Channel in C

In order to test implementations in other environments, we created a reference implementation in C. Creating a cache covert channel using our transmission scheme in C language is very straightforward because it is possible to create an eviction set that targets a single cache set inside a single slice. In C the virtual addresses are known and with the Linux provided "/proc/self/pagemap" one can retrieve physical addresses. With physical addresses, one can find out the exact cache set and slice that a single address maps to [5]. By allocating a large array, one can pick out the indices that map to the same set and slice and create an eviction set.

If both the sender and receiver have full access to physical memory addresses, they can both agree to a cache set and slice and use our transmission scheme without any further complications.

3.2 Covert Channel in JavaScript

Creating a covert channel using JavaScript over a native programming language is very useful because the victim would only have to visit a website instead of executing a binary. As discussed earlier, it is possible to create an eviction set for a particular cache set in JavaScript, but it is not possible to find out for certain in which cache slice that cache set

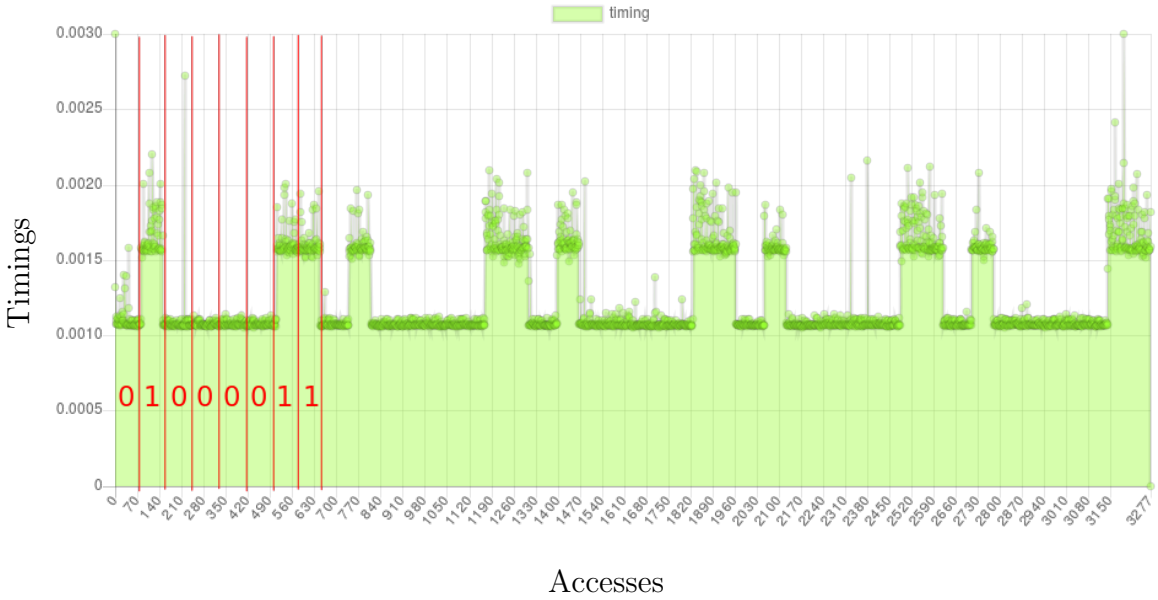


Figure 7: Simple Cache Covert Channel scheme visualized. Each point on the graph represents the access time from a Prime+Probe Probe step. For each millisecond, the sender induces a high or a low timings, represented by a '1' or '0' on the histogram.

resides in. It is, however, possible to create multiple eviction sets and run our transmission scheme parallel multiple times.

On a dual-core CPU, it is possible for the sender and receiver to agree on a particular cache set. The receiver then creates 2 eviction sets, one for each cache slice, and runs the receiving end of the covert channel twice in parallel. When the sender accesses data that maps to that set the receiver can quickly determine which of those 2 eviction set works. Our transmission protocol sends a single bit over a whole millisecond. This is a great amount of time in computing terms and creates very unnatural cache access patterns, as can be seen in Figure 7. This is easily distinguishable from random noise.

3.3 Covert Channel in PDFs

The real advantage of our transmission scheme is, that as long as the receiver of the covert Channel is able to identify the cache set that the sender is sending to, the sender does not have to create an eviction set or even target any cache set at all. The sender does not even need to have access to a high precision timer. The JavaScript API provided by most modern PDF viewers provides the function "Date.now()", which provides a real-clock time-stamp with an accuracy of 1ms. The sender could just simply access a piece of random data in his code, according to our transmission protocol.

As it turns out, it is not too difficult to find out to what cache set the data from the sender is mapped to. As seen in Figure 7, our visualized transmission scheme looks very artificial and not at all like the random noise that one would typically find in a cache set

under normal use.

In order to be able to identify the sender's cache set, we create an eviction set for every possible cache set and look for access patterns that resemble our transmission scheme.

On our test CPU, an Intel Haswell dual-core i5-4200U, the search space in the LLC consists of 4096 sets, but we found out during testing that when the sender accesses a single integer, the receiver could receive the access patterns in any of 35 different cache sets, spread throughout the search space. That seems to be the case because the overhead from interpreting JavaScript may access different data when accessing a single variable. This makes the search-space orders of magnitude smaller. Figure 8 shows on which cache sets we could receive a 8 bit 'C' character when the sender only accessed a single variable. The chance of false positives is also very low because our transmission scheme creates patterns in cache usage that do not happen under normal use and because the readings are averaged over the course of a whole millisecond.

We tested cache sets for viability by trying to receive an 8 bit character. This way, testing a single cache set for viability takes 8 milliseconds. By testing multiple cache sets at once and by the sender accessing not just one, but multiple different variables when he wants to send a '1', it is possible to track down a candidate cache set in only a few seconds.

All of this combined makes it possible to create a PDF document that, when opened with a JavaScript-enabled PDF viewer, starts sending a message using a cache covert channel. In principle, this concept is not only limited to PDFs but to anything that is able to execute some form of code.

It is, however, not possible to receive data from an environment that is not able to create eviction sets, meaning that in the case of PDFs our covert channel is only unidirectional.

4 Performance Evaluation

In order for a cache covert channel using Prime+Probe to perform well, one has to consider that other processes may use that same cache set, with which one is trying to communicate, and therefore introduce unwanted noise. It would be standing to reason that one would try to look for specific cache sets that are less frequently used than others in order to avoid noise. However, which cache set is used by a particular program is depends only on the physical address of the variables a program uses. Because of virtual memory, normal programs do not care about what exact physical address they are given on start-up and therefore the addresses, and in turn, the used cache sets, are completely random for every process on the system. In fact, the concept of dividing the cache into different sets relies on the fact that every cache set gets used roughly equally, in order remain efficient. This means that we did not find any fixed cache sets that get used more than others.

The kind of noise that cache covert channels have to deal with is shown in Figure 9. A randomly selected subset of 120 different cache sets was chosen to illustrate the base noise when there are typical background tasks like a web browser running. Each cache set was primed and then a short waiting period, using the posix function *sched_yield()*, was performed. After that, the duration of the Probe step was measured. This was applied to

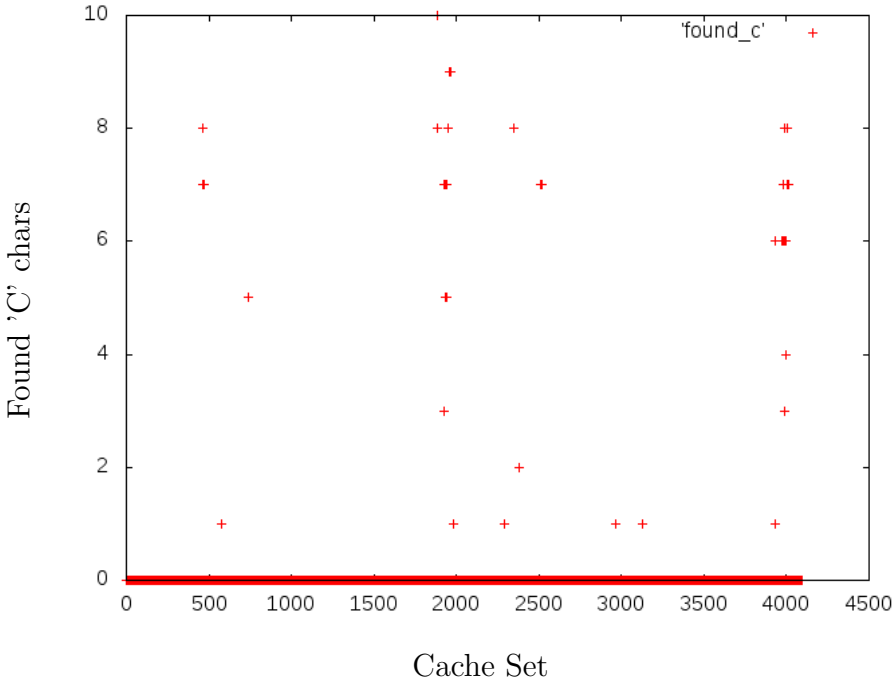


Figure 8: Cache sets on which the receiver can receive the senders message. The graph shows all cache sets found after scanning each cache set iteratively 10 times.

every selected cache set and repeated over a period of time. A black pixel on the graph indicates that the Probe step took less time than a certain threshold. The y-axis of this spectrogram represents a single cache set, and the x-axis represents time. If the probe step took longer than the threshold, then the pixel representing the measurement is gradually more red, depending on how much longer it took. With this, the activity of particular cache sets is clearly visibly.

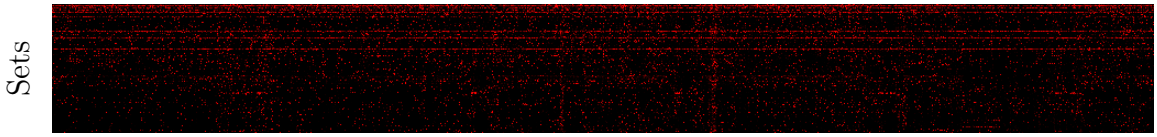
Maurice et al. [4] have shown that if a machine is under heavy load, the noise level rises uniformly across all cache sets.

4.1 Transmission Errors

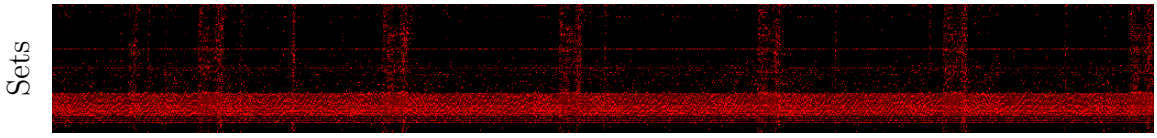
Cache covert channels have to deal with a multitude of problems [4], no matter in which way one chooses to send individual bits.

Substitution errors: Another process running in the background could access a cache set and therefore cause a '0' to read as a '1'.

Insertion errors: In order to send and receive bits correctly both the sender and the receiver have to be scheduled at the same time. If, for instance, the sender is not scheduled at the same time, the receiver could receive a long block of '0's.



(a) Background cache noise



(b) Background cache noise with 20 neighbouring cache sets being continuously accessed

Figure 9: Spectrogram showing the activity of 120 cache sets (y-axis) over time(x-axis). A more brightly lit pixel represents more activity.

Deletion errors: When the receiver becomes unscheduled during receiving, only a part of a message gets correctly received.

Those kind of errors are very common during a transmission and one needs to deal with them in order to achieve an error free high-performance channel.

A simple way to ensure that a message gets correctly received is to attach a checksum to the message and sending that message over and over again until it is correctly transmitted.

One good way to deal with substitution errors is to encode the message bytes with an error correcting code that can detect and repair bit flips. A textbook example of such a code would be hamming code.

Insertion errors and deletion errors are both caused because of the change in scheduling of the sender and receiver. There is no way to ensure that both stay scheduled at the same time. Those kinds of errors are called burst errors, and one way that they get dealt with is by the use of packets, that use a large amount of parity bits in order to be able to reconstruct the missing bits. If that is not possible, the packet needs to be retransmitted.

High-performance cache covert channels like the one described by Maurice et al. [4] use even more techniques, sometimes similar to those used in the wireless communication industry, in order to achieve the best possible performance.

Our previously described covert channel circumvents a lot of those problems by not being a high-performance channel by design in favour of being able to be implemented in a wider range of environments. Because our design averages a lot of measurements over the course of a whole millisecond, many problems like the sender and receiver not being scheduled at the same time. Maurice et al. [4] have shown that the average time that a process is not scheduled is around 10µs until it gets scheduled again, which is a timespan that easily averages out over a millisecond.

4.2 Performance

Using native programming languages and relying on shared memory with Flush+Reload Gruss et al. [11] have achieved a transmission rate of 298 KBps and a 0% error rate. With Flush+Flush a transmission rate of 496 KBps with an error rate of 0.84% in a controlled lab environment could be achieved. Cache covert channels using Prime+Probe have a much greater area of applicability because they do not rely on shared memory. Using a Prime+Probe covert channel speeds between 67 KBps with a 0.36% error rate and 75 KBps [13] with a 1% error rate could be achieved.

Maurice et al. have created a relieable Prime+Probe covert channel with a transmission speed of around 46 KBps [4] that is capable of sustaining a SSH or Telnet connection across 2 virtual machine instances on Amazon EC2.

It is important to mention that these absolute numbers could possibly scale up with faster CPUs and memory in the future without having to make any changes to the procedure.

The transmission rate of our portable transmission scheme, that can send messages from environments that do not provide a precise timer, is capped by the resolution of the provided timer. In the JavaScript API provided by PDF viewers, for instance, the highest resolution available timer has a resolution of 1ms, meaning that it is theoretically capped at 1000 bits per second. Also, since this channel is unidirectional, it is not possible for the sender to know if his messages were received, but this could be solved by creating a second covert channel in the other direction.

The actual error-less transmission rate of our transmission scheme, without using error correcting code, is 898 bits per second. This number is so close to the theoretical maximum because our protocol averages measurements over 1ms, making it less prone to noise.

5 Countermeasures

The countermeasures against cache attacks that spy on user behaviour and against cache covert channels are the same because both build on the same techniques.

Timers

One of the most easily implementable countermeasures is the removal of timers that are precise enough to distinguish between cache hits and misses. This is difficult in native code, because *rdtsc* is a hardware instruction. Modern browsers have reacted to cache attacks by reducing the accuracy of the "performance.now()" down to 5 microseconds. Kohlbrenner et al. [24] in their work attempted to make it more difficult to retrieve a high-resolution timer and proposed various software fixes that would make it more difficult for someone to get access to such a timer.

However, Schwarz et al. have demonstrated that the proposed countermeasures do not prevent cache attacks from JavaScript equipped browsers by showing, that it is

possible to build a multitude of timers that are precise enough to be used for cache attacks.

Disselkoen et al. have shown a new type of cache attack, Prime+Abort [14], that does not rely on a timer at all.

Hardware performance counters

With the use of performance counters, it is possible to detect the amount of cache hits and misses. When a Prime+Probe attack is in use, the hits and misses deviate strongly from the norm, making it detectable. This detection is necessarily imperfect and prone to both false positives and negatives[25, 11, 26, 27].

Hardware Countermeasures

Cache attacks are only possible because of the architecture of the underlying hardware. Hardware fixes can only fix the problem for future hardware, upgrading existing hardware is not an option most of the time.

The Skylake-X generation of Intel chips has changed the cache hierarchy in such a way that the last level cache no longer has the inclusive property, meaning that the methods discussed in this paper no longer apply directly to that generation of Intel Chips. It is unlikely that this was done in order to mitigate cross core cache attacks but rather to improve performance. Wang et al. [28] have proposed a way to randomize the mapping between memory addresses and cache sets, which would make cache attacks more difficult. Their paper was published 10 years ago, and, as of now, no widespread hardware countermeasures against cache attacks have found their way onto todays CPUs.

Another approach would be randomizing the cache replacement policy, but this would generally only lower performance of cache attacks and not make them impossible.

Partitioning cache sets in a way, that cache sets are assigned to only a single process, or disallowing a single process to use up all ways in a cache set, would be a powerful defense against all kinds of cache attacks. Some efforts have been made to implement these ideas by Liu et al. [29] but not in a way that offers a general defense against all kinds of cache attacks.

One also has to take into consideration that any architectural change that negatively affects general computing performance, in non security-critical situations, is a tradeoff that many are not willing to make.

6 Conclusion

In this work, we demonstrated how current cache attack techniques can be applied to build an effective covert channel that can run inside a modern webbrowser. Using cache covert channels, it is possible to circumvent sandbox isolation mechanisms imposed by webbrowsers, virtual machines, app-permission systems on mobile operating systems and many more.

Furthermore, we demonstrate a PDF file that is capable to send data to a webbrowser using our cache covert channel. That file is compatible with a multitude of JavaScript APIs provided by modern PDF viewers, including the one provided by the Google Chrome browser. The principles used in the creation of that PDF are not limited to PDFs alone but should be able to be used in a wide range of execution environments, but only JavaScript was evaluated.

Our work demonstrates that the requirements in order to establish such a covert channel are very low. The receiving end of our covert channel can be implemented in JavaScript and the sender can be implemented in many more execution environments. The fact, that the sender and receiver have no need to run native code at all, broadens the scope in which cache covert channels can be used greatly.

List of Figures

List of Tables

References

[1] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 313–328. IEEE, 2011.

[2] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[3] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in cryptologyCRYPTO99*, pages 789–789. Springer, 1999.

[4] 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. *NDSS, San Diego, CA, US*, 2017.

[5] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Workshop on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[6] Intel. Intel® 64 and ia-32 architectures software developers manual. 3A, 3B, 3C, 3D.

[7] Xian-He Sun and Dawei Wang. Concurrent average memory access time. *Computer*, 47(5):74–80, 2014.

[8] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[9] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015.

[10] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[12] Colin Percival. Cache missing for fun and profit, 2005.

[13] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[14] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx. 2017.

[15] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. Towards a systematic study of the covert channel attacks in smartphones. In *International Conference on Security and Privacy in Communication Systems*, pages 427–435. Springer, 2014.

[16] Brent Carrara. *Air-Gap Covert Channels*. PhD thesis, Université d'Ottawa/University of Ottawa, 2016.

[17] Mordechai Guri, Yosef Solewicz, Andrey Daidakulov, and Yuval Elovici. Diskfiltration: Data exfiltration from speakerless air-gapped computers via covert hard drive noise. *arXiv preprint arXiv:1608.03431*, 2016.

[18] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, pages 58–67. IEEE, 2014.

[19] Mordechai Guri, Boris Zadov, Eran Atias, and Yuval Elovici. Led-it-go: Leaking (a lot of) data from air-gapped computers via the (small) hard drive led. *arXiv preprint arXiv:1702.06715*, 2017.

[20] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*, pages 108–122. Springer, 2015.

[21] 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 21th International Conference on Financial Cryptography and Data Security (FC17)*, page 11, 2017.

[22] Wong, h.: Intel ivy bridge cache replacement policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/. Accessed: 2017-09-26.

[23] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[24] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480, 2016.

[25] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[26] N Herath and A Fogh. These are not your grand daddy's cpu performance counters: Cpu hardware performance counters for security. *Black Hat*, 2015.

[27] Mathias Payer. Hexpads: a platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[28] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[29] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, pages 549–564, 2016.

[31] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.