A Javascript Side-Channel Attack on LLC

Alan Chiao, Donghyun Choi, Jeffrey Sun

May 14, 2015

Abstract

Abstract: A side-channel attack is an attack that utilizes information gained from the physical implementation of a security scheme, as opposed to the algorithmic implementation. For example, such information could come from power consumption or hardware timing. In this project, we focus on side-channels that utilize information leaked by the hardware cache implementation. The project and implemented attack are inspired by a recent paper on Javascript side-channel attacks [1].

1 INTRODUCTION

A side-channel attack bypasses the security of a theoretically sound cryptographic algorithm by taking advantage of information leakage in the physical implementation of a system. A cache-based side-channel attack is a subset of these attacks that exploits the difference in access times of a cached value and a uncached value. In section 2, we discuss modern cache architecture, the main cache attack algorithm, and properties of Javascript, all of which are necessary to understand the implementation of the attack. Section 3 provides a brief summary of our implementation of the attack, as well as a link to the source code. In section 4, we discuss the results we observed as well as challenges encountered. Section 5 lists some possible veins of future investigation and/or countermeasures.

2 Background

2.1 Modern Cache Architecture

Modern processors use caches that are set-associative. These caches have S cache sets, each with N cache lines that hold B bytes each. When a program attempts to access data from the main memory, the program first checks to see if the data is already in the cache, known as a cache hit. Accessing data in a cache is preferable because it is faster than accessing data from the main memory.

If the data is not in the cache, a cache miss occurs. Upon a cache miss, B bytes are pulled from the main memory and placed into a cache line of the appropriate cache set, determined by a hashing function. If all N cache lines of that cache set are filled, the least recently used cache line is evicted. We use this eviction process to gather information from the cache.

The cache system has multiple layers, typically with L1, L2, and L3 caches. The higher the level of the cache, the bigger it is and the closer it is to RAM. This means that access times are slower. Typically, a program first tries access the L1 cache and upon failure, tries the next level caches one by one.

With respect to variable accesses, the nature of multilayer cache system will add noise because cache hits may happen at any of three levels, each with have different access times. Another source of noise to variable accesses is that modern compilers make several cache-related optimizations to the source code, all which affect the timing of variable accesses. For instance, cache prefetching will preload a variable in the cache, even if it was not before, before a variable access.

2.2 Prime+Probe

The Prime+Probe attack described in [2] is used for the cache attack in this project. For our system specs, the attack takes advantage of the 12 cache line limit on the amount of data that can be stored in a particular cache set. When the system attempts to store a 13th cache line in a cache set, a previously stored cache line will be evicted. With this property, the attack follows four key steps:

1. The attacker creates an eviction set, a set of 12 cache lines that map to the same cache set as a target cache line . A target cache line is one that a specific user action depends on, such as mouse movement and keyboard typing. The specific user action is what we attempt to monitor through the attack.

2. The attacker "primes" the cache set by iteratively accessing each element in the eviction set. This fills the cache set to its max capacity, 12 cache lines, with our data. Note that if we immediately iterated through each element in the eviction set again, each element access will be fast because the access will be from the cache instead of RAM.

3. The attacker waits an arbitrary amount of time, enough time so that the specific user action could be performed. The attacker "probes" the cache set again by re-accessing the elements of the eviction set.

4. The time it takes to "probe" given that our 12 cache lines remained in their cache set is designated as the threshold time. If the "probing" process takes longer than the threshold, then we know that at least one of our cache lines has been evicted from the cache. The source of this eviction would be the specific user action, so we know the user action has happened.

2.3 Security Characteristics of Javascript and Web 2.0

The security considerations of Javascript brings challenges in our attack implementation. The main challenge that we face in Javascript is that there is no notion of memory addresses. In C, we can grab the physical memory address of any variable, which provides us with valuable information.

For instance, in a particular cache attack in [3], the attack takes advantage of the fact that the 5 - 16 bits of a physical memory address are directly used to determine the cache set that the data at the address would be cached in when necessary.

2.4 Related Work

There have been previous research on the study of cache-based attacks. For our attack, we reference the procedures described in [1] and [3].

3 Implementation

The implementation is based on the described algorithms in [1]. The source code can be found in https://github.com/alanchiao/6.857-Javascript-SC-Attack.

Paper [2] suggests the following process for creating an eviction set data structure.

1. Access a variable x.
2. For an untested cache line y:
   (a) Iterate through the buffer without touching cache line y, which may be in the same cache set as x. Note that the buffer has exactly 12 cache lines in the same cache set as x, which is just the perfect amount to evict x. If we skip over cache line y in the same cache set as x, we only iterate over 11 cache lines, and x should remain in the cache.
   (b) Access x again. If x remains in the cache, then cache line y is in the same cache set as x. We decide if x remained in the cache by looking if the difference in access latencies for 1) and 2b) is low enough.
   (c) Do the above for each untested cache line, and if we find 12 cache lines that work, the algorithm has worked.

This algorithm has several ambiguities in the context of a large buffer, which the paper implies is the source of the untested cache lines.

The buffer has a total of 131k cache lines across 8192 cache sets. This is more than 12 cache lines per cache set, so step 3 of the above algorithm will more likely than not fail, finding more than 12 cache lines in the same set as X.

We tried another implementation from an algorithm in [3], which first searches for a conflict set of 8192 * 12 elements. From that, it divides the conflict set into the 8192 cache sets. To find the conflict set, we follow this process:

1. For each untested cache line, see if accessing the elements of the current conflict set evicts the cache line.
2. If eviction occurs, then there are already 12 cache lines in the conflict set that are in the same cache set as the untested cache line, and we ignore it. Otherwise, add the untested cache line to the conflict set. Note that this steadily leads towards have 12 cache lines in the pertinent cache set.

The implementation of this is in the nicta subdirectory of the repository.

4 Results

4.1 Flushed and Unflushed Variable Access Time

We first verified that through Javascript, we could successfully evict a variable from the cache by filling the cache up with 12 cache lines. Our attack requires a sufficient access latency difference between a variable that is flushed from cache and one that is not. This experiment verified that we could fill a LLC cache set from our unprivileged Javascript code, as necessary in the second step of Prime+Probe.

We allocated a 8MB buffer array in virtual memory in Javascript and used iteration over the array as a means of flushing out said variable from its cache set and replacing the data in the cache set with the data in the 8MB buffer array. The 8MB is not a size set in stone, but rather one large enough such that we found it would consistently flush out the entire LLC cache and succeed in the next steps.

The steps we followed were as following:

1. Flush out a variable from cache and record the next access time as flushed.
2. Access the same variable again and record the access time as unflushed.
3. Access the same variable again and record the access time as unflushed2.
4. Flush out a variable from cache and record the next access time as flushed2.

Steps 3 and 4 were done as precaution to make sure the flushing process worked. Figure 1 and Figure 2 illustrate the gap between access latencies of a flushed and unflushed variables in a MacOS and Ubuntu 14.04 OS, respectively, both with 2.2GHz Intel Core i7 Processor. The verification process was successful.

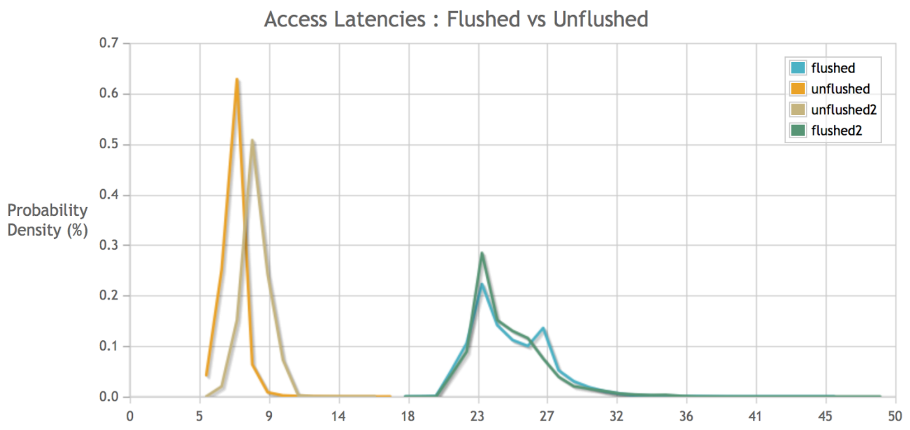


Figure 1: Probability distribution of access times for flushed vs unflushed variables on MacOS. We observe that the access latencies of a flushed variable is significantly higher than that of an unflushed variable.

–

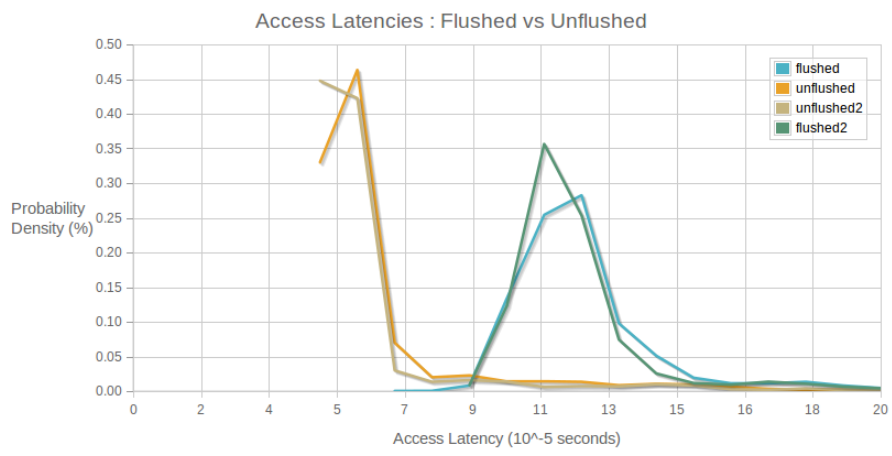


Figure 2: Probability distribution of access times for flushed vs unflushed variables on Ubuntu 14.04

4.2 Checking L3 Updates

In the previous section, we have verified that we were successfully able to flush the cache to alter variable access latencies on a single tab in a single browser. However, it is possible that these access may not alter down to the L3 cache, as the processes of a single tab may reside in a single core.

To check that our Javascript code actually flushes and updates the L3 cache, we perform the following test. We first prime the cache using a new language C. We note that this is much simpler to implement, as in C, we have full access to variable memory locations.

The following steps outline the test:

1. flush out a variable in C
2. access the variable in C
3. flush out the cache in Javascript
4. access the variable in C

If our Javascript code successfully flushes the L3 Cache, we would observe a difference in access time in steps 2 and 4.

4.3 Threshold Selection for Eviction Set Formation

As a recap, for forming the eviction set, we need to pick a threshold value for the time difference there should be between 1) and 2b) [from Section 2.2] above which we consider a variable to not be in the same cache set.

Note that while access time variation is low for an unflushed variable, access time variation is high for a flushed variable. This raises a problem because the suggested PRIME+PROBE attack relies on the difference between the time in 1) and 2b). No matter what threshold difference we pick, we will get either false positives or negatives. If we pick a small threshold difference, we will get false positives since 1) and 2b) may both be flushed, but they may have a large difference due to the large range. If we pick a large threshold difference, we will get false negatives (e.g. a low flushed time and high unflushed time may be smaller than our threshold.) In fact, because the range of the flushed data is wider than the gap between flushed and unflushed, there is no threshold value we can pick that will eliminate both false positives and negatives.

In Paper[2], this problem is also present in their graphs, displayed in Figure 3.

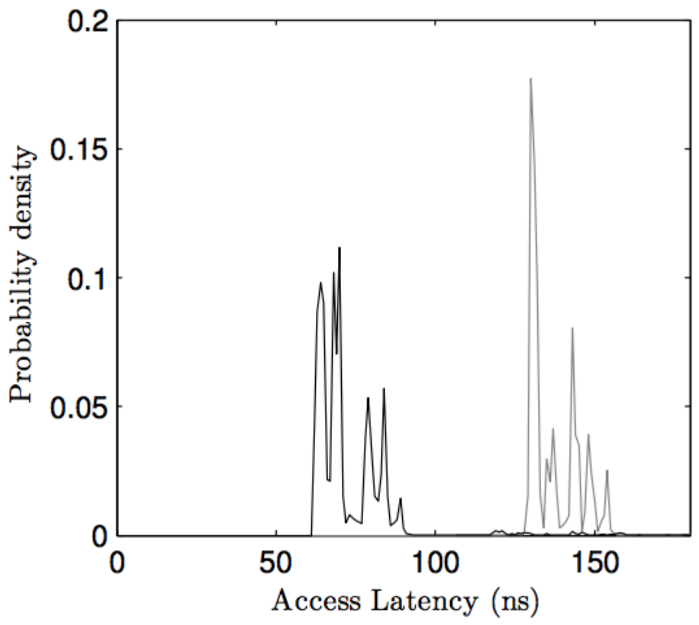


Figure 3: Probability distribution of access times for flushed vs unflushed variables, from Paper[2]

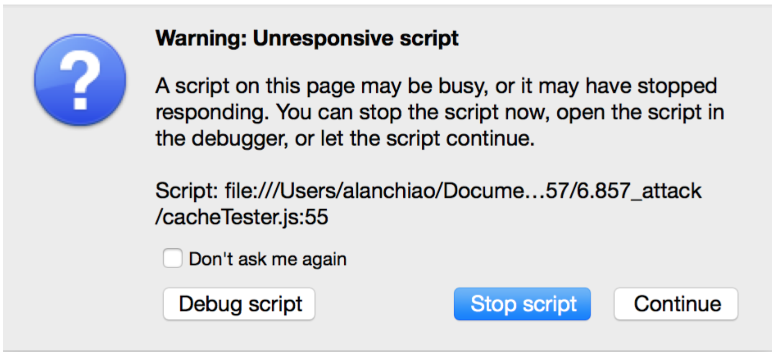
We found it more effective to simply use a different criterion, the time in 2b), as a threshold, as opposed to the difference between 1) and 2b). By observation, $16 * 10^-5$ seconds works well as a threshold value, being consistently between the flushed and unflushed data sets. If 2b)'s access time is above the threshold, we consider it to have been flushed. Else, we consider it to remain unflushed.

Another issue with threshold selection arises with different operating systems. From Figures 1 and 2, we find that the difference in access times of flushed and unflushed variables in Ubuntu 14.04 is significantly less than that of MacOS. Both of these CPUs had the same Intel 2GHz Intel Core i7. This suggests that Ubuntu somehow runs faster.

This difference may be problematic in determining the threshold, as it may result in multiple false positives, and render us unable to attaining our eviction set.

4.4 High CPU Usage

An issue we found with the attack is that a lot of browsers, such as Firefox, will display a warning message if the Javascript code that runs on a page consumes too much CPU processing power for a period of time. In Firefox, this time is eight seconds. In the paper, the suggested time to create the eviction set data structure is two minutes, more than enough to trigger the error message in Figure 4.



Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, open the script in the debugger, or let the script continue.

Script: file:///Users/alanchiao/Docume...57/6.857_attack /cacheTester.js:55

Figure 4: High CPU error message on Firefox

In practice, this error message will immediately tell the user that something is awry, causing the user to close the page running our attack program.

As an attempted remedy, we used the Javascript equivalent of a sleep function, to give the CPU time to rest from our program, which has a for loop that runs many iterations. In Python, this would be a simple time.sleep() call in each iteration of the loop. In Javascript, there is no time.sleep(). Instead, we have to nest some setTimeout() calls that run an iteration of the loop after a break. There are two problems with this remedy

1. It slows down a process which is already quite slow

2. Extensive usage of setTimeout (under linkedCacheTesterSleep.js), distorts the graphs that we had in Figure 1. This is quite possibly due to the fact that setTimeout is not a simple sleep function, that is, it may have side effects on the cache.

5 Further Investigations

5.1 Defenses Against Attacks

We considered defenses against this cache attack that were not mentioned in the original [1]. The paper suggests that any defense mechanisms would require significant changes in the cache system or the way that javascript is executed. We feel otherwise.

A simple defense could take advantage of two aspects of the javascript side channel attack

1. The attack only works on detecting interesting activities that operate on level low enough so that they are close to the caches, such as mouse and network activity. The number of interesting activities is highly limited.

2. By using the attack itself, we are able to detect which cache sets are affected by the mouse and network activity and manipulate them on the defender's side.

3. The accuracy of the timer provided by Javascript could be fuzzed. Most applications do not require timer accuracy on the nanosecond scale.

To defend against mouse activity monitoring, we can run a defensive program that randomly flushes out different cache sets associated with mouse activity, rendering the data that the attacker sees to appear randomized. This defense can extend to defending against other activity monitoring without being costly because the number of interesting activities is limited in number.

6 Conclusion

Although the attack outlined in [1] is theoretically quite powerful, in the current state the difficulty of implementation is considerable, and the applications are quite limited. Implementation is hard for several reasons: the noise from L1/L2, indirection introduced by Javascript, and memory access optimizations. An attacker would need to go to considerable lengths to surmount these obstacles. In terms of practicality, it is unlikely that an attacker could glean very useful data from this attack, as data can only be collected at a very low level (such as mouse movement). There are also natural barriers in place already, such as the warning messages that appear under CPU-intensive Javascript applications. Browser vendors should be wary of this kind of attack, but if there are significant tradeoffs, they probably do not need to rush to implement countermeasures.

References

[1] http://arxiv.org/pdf/1502.07373v2.pdf

[2] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. https://eprint.iacr.org/2005/271.pdf

[3] https://ssrg.nicta.com.au/publications/nictaabstracts/Liu_YGHL_15.abstract.pml