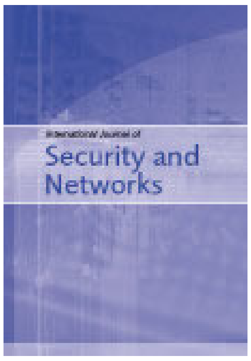


International Journal of Security and Networks

ISSN online: 1747-8413 - ISSN print: 1747-8405
https://www.inderscience.com/ijsn

Enabling secure modern web browsers against cache-based timing attacks

Sangeetha Ganesan

DOI: 10.1504/IJSN.2023.10058114

Article History:

Received:	11 September 2022
Last revised:	14 June 2023
Accepted:	14 June 2023
Published online:	12 March 2024

Enabling secure modern web browsers against cache-based timing attacks

Sangeetha Ganesan

Department of Artificial Intelligence and Data Science,
R.M.K College of Engineering and Technology,
Tiruvallur District, Tamil Nadu, India
Email: gsangeethakarthik@gmail.com

Abstract: Web applications have grown to be the foundation of any kind of system, ranging from cloud services to the internet of things (IoT) systems. As a huge amount of sensitive data is processed in web applications, user privacy shows as the most important concern in web security. In the virtualisation system, cache side channel (CSC) attack techniques have become popular to retrieve the secret information of other users. This paper presents a run-time detection and prevention mechanism, called browser watcher (BW), for time-driven CSC attacks. The computation overhead of the proposed BW java script engine is monitored and tabulated for the different domains. The average cache miss rate is measured from 23% to 89%. Once the BW system identifies the attacker, then it prevents stealing the secret information of the victim. This makes it very hard for the attacker to find the memory access pattern of the victim.

Keywords: cache side channel; CSC attack; timing attack; BW system; cache attack prevention; internet of things; IoT.

Reference to this paper should be made as follows: Ganesan, S. (2024) 'Enabling secure modern web browsers against cache-based timing attacks', *Int. J. Security and Networks*, Vol. 19, No. 1, pp.43–54.

Biographical notes: Sangeetha Ganesan received her BE degree in Computer Science and Engineering from Periyar University Salem, ME degree in Computer Science and Engineering from Anna University, Chennai, and PhD in Faculty of Information and Communication Engineering, Anna University, Chennai. She is currently working as an Associate Professor in R.M.K. College of Engineering and Technology, Puduvoyal. Her research interests are distributed computing, cloud computing, security, data science and machine learning.

1 Introduction

Side channel analysis is the famous intelligent part of the cryptanalytic attack (Kocher, 1996; Osvik et al., 2006; Yarom and Falkner, 2014). The secret information is emitted to the attacker from secure devices by analysing its physical signals (temperature, power, radiation, heat, laser, etc.) as it achieves any secure operation (Mangard et al., 2007). A particular type of side channel (SC) attack which is related to personal computers is the cache side channel (CSC) attack. The CSC attack utilises the use of cache memory as a shared memory set between different processors and it releases secret information (Hu, 1992a; Osvik et al., 2006). The CSC attack is a part of SC attacks that utilise the difference in access times of a cached value and an un-cached value (i.e., RAM value).

To secure the data of the cloud computing customer, CSC attack detection and prevention is essential. Most of the CSC attacks use one of the critical operations such as accessing timestamps on the shared cache of the physical CPU. Two of the famous CSC attacks are Prime + Probe and Flush + Reload. Both techniques calculate the access time slots to read a particular location from the memory. The read operation will be done either by cache hit or cache miss events.

Over the past 25 years, JavaScript (JS) has grown to be the biggest language on the web. Out of the 10 million majority popular websites, 94.7% use the JS language (W3Techs, 2017). There are security and privacy issues as a result of allowing every website to access APIs like the battery status API and the Geolocation API (Popescu, 2016; W3C, 2016a). Websites can utilise the sensors to get the finger print of the user by identifying more sensors. It also updates secure device values like battery level or geo-location. Moreover, these data can increase the CSC attacks on user input (Mehrnezhad et al., 2016; Cai and Chen, 2011).

Micro-architectural attacks include both SC attacks and covert channel attacks. These attacks are entirely implemented in software. Microarchitectural attacks are also feasible in JS. The timing side-channel attacks have been established and observed to collect the browser history of a web user (Felten and Schneider, 2000; Jang et al., 2010; Weinberg et al., 2011), geolocation (Jia et al., 2015) of the user, whether a user is logged in to any other websites (Bortz and Boneh, 2007), and CSRF tokens (Heiderich et al., 2012).

The suggested detection mechanisms rely on hardware performance counters for detection and presumptively find more cache hits and cache misses from all cache attacks than from benign applications. Since the first use of clflush is obviously a system function and the second assumes that no other process has access to the data, we advise restricting its use to memory pages that the process has write access to and that the system permits clflush access to. Despite being very efficient at recovering exponent bits, the attack has some limitations. Another limitation is the length of the secret key. A total of 2,200 cycles are required on the Dell system to probe three memory locations. Time periods that are shorter than that cannot therefore be used for the attack. For shorter key lengths, time slots of 2,200 cycles or more do not provide enough resolution to identify the victim. This tactic's primary flaw is that the attacker can create high resolution clocks using alternative methods. Utilising network data and running a 'clock' process in a different execution core are two examples.

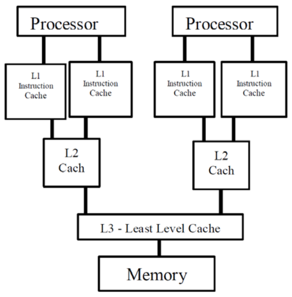
Usually, web browsers cannot get the browsing information on one website from being directly accessed by another website. Nevertheless, any web applications executed in the same browser share the same runtime environment with the attackers. Such shared information leads to SCs for malicious websites to find out information of other origins. The proposed browser watcher (BW) system identifies the attacker and secures the data of the victim from the attacker. The BW system analyses the modern web browser behaviour of the user and it classifies the user as an attacker or a normal user. Once it identifies the attacker, the attacker will be prohibited to fetch the memory pattern of the victim.

This paper is structured as follows: Section 1 explains the introduction of the attacks. Section 2 shows the related works of the timing attack. Section 3 shows the working principle of Attack methodologies. Section 4 explains the proposed BW system model. Section 5 shows the experimental results and section 6 gives the conclusion of the work.

2 Related works of the timing attack

Modern processors use caches to retrieve data from the memory. The cache works in a set-associative (2-way, 4-way, 8-way, 16-way) manner. These caches have an 'S' number of cache sets, each with an 'N' number of cache lines that hold 'B' bytes in each cache line. When a program tries to fetch data from the memory, the program initially searches it in the cache. If the data is found the cache it is defined as a cache hit. If the required data is not there in the cache, then cache miss occurs. In the cache miss, with the help of a hashing function, the required 'B' bytes are pulled from the main memory and placed into a cache line ('N') of the suitable cache set. If all cache lines ('N') are filled, then the least used cache lines are ejected from the cache set and leave a free empty room to place the retrieved 'B' bytes. The access latency is less in cache memory when compared with the main memory.

Figure 1 Cache architecture



The cache system has three levels: Level1, Level2, and Level3 (lease level cache – LLC) caches (Figure 1). The higher level (L1) cache is bigger and closer to the RAM. The access time of L1 is less. A program initially attempts to fetch data from the L1 cache and upon failure then tries next-level caches one by one. For the variable access, the multilayer cache system includes noise because the cache hit may occur in any of the three levels. Based on the cache hit/miss rate, the CSC attacks are classified into four categories as shown in Table 1. This proposed solution applies to time-based SC attacks. This suggested fix is only applicable to time-based SC attacks in categories 1 and 3, not all time-based SC attacks.

The BW system focuses on web browser time-based probe attacks (Felten and Schneider, 2000; Jia et al., 2015; Kotcher et al., 2013; Agrawal et al., 2003). It retrieves private information such as cryptographic keys or the status of other virtual machines (Agrawal et al., 2002, 2003). Timing attacks are popular types of SC attacks. In the timing attack, the time required to perform a secret operation is observed and tried to fetch the private information on the attacked system. Kocher (1996) showed that accidentally the timing characteristics expose enough information to fetch the complete secret key from a vulnerable cryptosystem (Sangeetha and Ganesan, 2020). Oren et al. (2015) demonstrated that the attacker could use JS TypedArrays to calculate the instantaneous load on the LLC with the help of high-resolution reference clock-like performance.now(). Sangeetha introduced a new secure allocation technique to prevent cache side-channel attacks (Sangeetha and Ganesan, 2020).

Bortz and Boneh (2007) offered two types of web-based timing attacks targeting dynamic web pages. In the first web-based direct timing attack, an attacker can find the response time from a website to attain information about the website's state. The second web-based timing attack is a cross-site timing attack that permits an attacker to get details on the position of a user at a cross-origin website.

To minimise the timing attack, the Tor Browser developers decreased the resolution of the performance.now timer to 100 ms. In response, the W3C (2016b), browser vendors, and some major web browsers (Chrome, Firefox) have applied similar techniques to reduce the timer resolution to 5 µs to overcome Oren et al.'s (2015) cache timing attack. But it is not possible to deliver the exact timing to the user from this low-resolution timer. Hu (1992b) proposed 'fuzzy time' ideas to build trusted browsers. The fuzzy time system degrades all clocks either internally or externally and also it shrinks the bandwidth of all timing channels. Kotcher et al. (2013) identified two-timing probing attacks using CSS default filters. In the first timing probe attack, by exploiting the document object model (DOM) rendering time difference, we can check whether a particular user has an account on the website. In the second timing probe attack, with the help of stealing pixels, we can find the browsing history of the user or can read the next token.

Rokicki et al. (2021) have studied the evolution in the previous years and the current state of JS-based timers and timing attacks for Chrome and Firefox, evaluating the resolution and measurement overhead for the two most efficient timers: performance.now() and Shared ArrayBuffer. Timer-based countermeasures like clamping the resolution and adding jitter do not prevent attacks, but increase the time needed to exploit these attacks.

In order to protect JS from web concurrency attacks, Chen and Cao (2020) proposed JSKERNEL, the first general outline that does so. In order to strengthen security, the JS kernel, which was inspired by operating system concepts, mandates the execution order of JS threads and events. They put together a JSKERNEL prototype that could be installed as add-on extensions for the three most popular web browsers, Microsoft Edge, Mozilla Firefox, and Google Chrome.2.1 JS and timing measurements.

JS is an object-based scripting language used by every modern browser. The JS language is sandboxed and conceals the concept of addresses and pointers. The native code can provide a cycle-accurate timestamp from the timestamp counter; that is not possible in the JS. In native code, the user can obtain the timestamp counter through the unprivileged assembler rdtsc instruction. Nevertheless, JS cannot execute this arbitrary instruction. Instead, JS uses the performance.now timestamp function to provide high-resolution time in sub-millisecond. Based on the high-resolution time application programming interface (W3C 2016a), different attacks have been established. Van Goethe et al. (2015) demonstrated that from the cross-origin resources, the attacker can fetch the private data of the user by measuring their access time.

Figure 2 shows how injecting java script advertisement code permits an attacker to determine the victim's memory pattern. The victim's multi-tabbed web browser was compromised when the attacker's code was introduced from an untrusted website.

2.1 *Micro-architectural attacks*

Modern processors are optimised for computational efficiency and power. Conversely, optimisations commence side effects that can be oppressed in so-called micro-architectural attacks. Micro-architectural attacks include fault attacks and SCs on micro-architectural components or exploits of micro-architectural elements, e.g., caches, pipelines, DRAM and buses. Attacks on caches have been examined widely in the past 20 years, using cryptographic performances (Kocher, 1996; Bernstein, 2004). The time varies between a cache miss, and a cache hit can be suppressed to discover secret information from co-located virtual machines and processes. Modern attacks exploit either Flush+ Reload (Yarom and Falkner, 2014), if read-only shared memory is available or Prime+ Probe (Osvik et al., 2006) otherwise. In these attacks, the attacker operates the status of the cache and shortly checks whether the status has altered. Moreover, attacks on cryptographic executions (Osvik et al., 2006; Patil et al., 2011), these attacks can be utilised to make covert-channels (Maurice et al., 2017) or ASLR (Gras et al., 2017).

2.2 *Micro-architectural and side-channel attacks in JS*

Micro-architectural attacks were only recently suppressed by JS. As JS code is essentially single-threaded and sandboxed, attackers look like a definite challenge in contrast to attacks in native code. Recognised some requisites that are the origin of micro-architectural attacks, which is each attack, relies on at least one of these primitives. Furthermore, sensors initiated on many mobile devices, in addition to modern browsers, commence SCs that can also be suppressed by JS. Table 1 provides a summary of side-channel attacks and all 11 known micro-architectural in JS and their requirements. Oren et al. (2015) proposed 11 microarchitectural side-channel attacks in JS. All these attacks are categorised into five groups: shared data, memory addresses, multithreading, sensor API, and accurate timing (Table 2).

Table 1 Categories of cache side-channel attack

	Time-based attack	*Access-based attack*
Cache hit based attack	Category 1: If the victim gets more cache hits while executing their entire critical/security operations, due to the reusing of its cache lines. e.g., cache collision Attack	Category 2: If the attacker gets less memory access time while fetching the victim-fetched cache lines. e.g., flush-and-reload Attack
Cache miss based attack	Category 3: If the victim takes a longer execution time of their critical operations, due to access to attacker-evicted cache lines. e.g., evict-and-time attack	Category 4: If the attacker takes longer memory access time, due to the victim's eviction of the attacker's cache lines. e.g., prime-and-probe Attack

Figure 2 End-to-end web browser attack scenarios (see online version for colours)

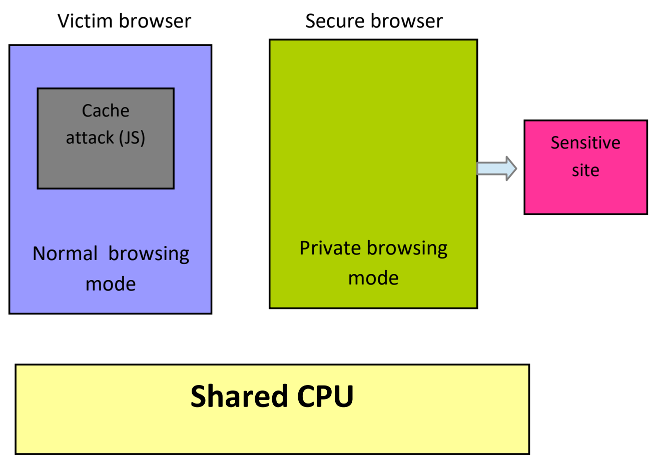


Table 2 Requirements of state-of-the-art side-channel attacks in JS

	Memory addresses	*Accurate timing*	*Multithreading*	*Shared data*	*Sensor API*
Rowhammer.js (Gruss et al., 2016)	3	3	1	1	1
Practical memory deduplication attacks in Sandboxed Javascript (Gruss et al., 2015)	2	3	1	1	1
Fantastic timers and where to find them (Schwarz et al., 2017)	3	3	2	2	1
ASLR on the line (Gras et al., 2017)	3	3	2	2	1
The spy in the sandbox (Oren et al., 2015)	2	3	1	1	1
Loophole (Vila and Kopf, 2017)	1	2	3	1	1
Pixel perfect timing attacks with HTML5 (Stone, 2013)	1	3	2	2	1
The clock is still ticking (Van Goethem et al., 2015)	1	3	2	1	1
Practical keystroke timing attacks in sandboxed JavaScript (Lipp et al., 2017)	1	2	2	2	1
TouchSignatures (Mehrnezhad et al., 2016)	1	1	1	1	3
Stealing sensitive browser data with the W3C Ambient Light Sensor API (Olejnik, 2017)	1	1	1	1	3

Note: 3 – high level requirements, 2 – medium level requirements 1 – low level requirements.

3 Attack methodologies

3.1 Prime + Probe attack

This technique involves two stages. In the prime stage: the attacker (‘A’) removes all the data of the victim (‘V’) iteratively from the targeted cache set by assigning an array of memory blocks to that set. The target cache set is fixed based on the action of the victim users such as keyboard typing and mouse movement. If the ‘A’ reads the same memory blocks again, the data will be accessed fast, because it is retrieved from the cache, not from RAM. Then the ‘A’ remains ideal for an interval before performing the probe step. In the probe stage: the ‘A’ again reads the same memory array which was read in the prime stage and also calculates the access time period. If the time period is greater than the certain threshold value, then the ‘A’ guesses that the cache set has been fetched by the victim during the interval time period. The ‘A’ repeats the same Prime and Probe actions until it collects the cache access pattern of the ‘V’. The ‘V’ cache access pattern can be used by the ‘A’ to extract the information about the ‘V’ operation. Refer to Algorithm 1 for Prime + Probe Attack. In the algorithm, the ‘A’ passes the created array address and a threshold value that finds the cache set details of the ‘V’.

Algorithm 1 Prime + Probe attack

```
1  Initialise address and threshold values
2  Initialise Boolean accessed [] to false
3  Read the address value
4  While(true) do
     a  Wait () //Till the victim accesses the cache lines
     b  Start = time () //Measure the time period
     c  Read the address value
     d  End = time () //Measure the time period
     e  AccessTime = End – Start //Measure the access time
     f  If (AccessTime > threshold) then
```

```
        accessed.push(true) //Cache Miss
  g  else
        accessed.push(false) //Cache Hit
  h  end if
  i  end while
5  Returned accessed []
```

Only with the help of the timer function, the ‘A’ can fetch the memory access pattern of the ‘V’. The proposed BW system detects the ‘A’ based on the usage of the timer function and secures the memory access pattern of the ‘V’.

3.2 Flush + Reload attack

The Flush + Reload (F+R) attack is an access-driven cache attack.

The first thing the adventure does is flush a specific memory line that the victim frequently uses. Following that, the adventure waits until the victim has completed its cryptography operation. The same memory lines are loaded once more by the adventure. A longer reload time shows that the victim used the memory line during the second step of the adventure. In Algorithm 2, the Flush+ Reload attack steps are listed.

Algorithm 2 Flush + Reload attack

```
Input: Memory line address addr, Plaintext PT, key K
Output: Time period
1  Flush(addr); // Which memory lines are mostly accessed by
   the victim.
2  Wait ();
3  $T_1$ ← Time ();
4  Encryption (PT, K);
5  Reload(addr);
6  $T_2$ ← Time ();
7  Return ($T_2 – T_1$);
```

- *Flushing stage:* in this first stage, the adventurer applies the clflush instruction to flush block ‘b’ from the cache, therefore making sure that it has to get back from the memory the next time it is required to be accessed. The clflush instruction does not flush only the memory block, but it is flushed from all the levels of the caches of all the cores. The attack would perform only if the attacker and victim processes were co-residing on the same core. This would have needed a stronger hypothesis than just being in the same physical machine.
- *Target access stage:* the adventure waits until the target executes a part of the code, which might utilise the block ‘b’ that has been flushed in the initial stage.
- *Reloading stage:* The adventure reloads once more the previously flushed block ‘b’ and calculates the time it gets to reload. Using this rdtsc command counts the hardware cycles tried in a process. Before reading the block ‘b’ and the cycle counter value, make sure that the mfence and lfence barrier instructions have completed the load and store operations respectively. Based on the reloading time, the adventurer decides whether the victim has used the memory block or not.

4 Proposed BW system model

The BW system detects the runtime CSC attack and prevents the attacker from fetching the memory pattern of the victim. At time T_i, the secret variable is accessed from memory and the memory access time is measured through the performance.now() function. At time T_{i+1} the same variable is accessed again and the memory access time is measured. If any time-based cache attack has not occurred in the interval of T_i and T_{i+1}, then at time T_{i+1} memory access time will be lesser than the memory access time at T_i. The TIMER_COUNT variable value will be increased based on the usage of the performance.now() function.

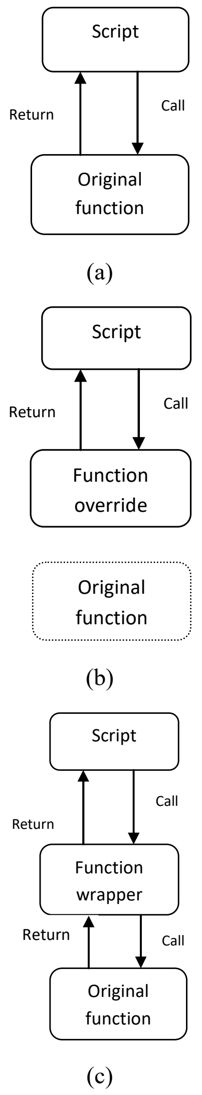
Algorithm 3 Access time measurement

```
1  Set AccessTime_FlushedLLC = AccessTime_UnFlushedLLC = 0
2  Set AccessTime_UnFlushedLLC_Test = AccessTime_FlushedLLC_Test = 0
3  Repeat K times // k → Number of Rounds
   a  Start = window.performance.now();
   b  Access the ith page
   c  end = window.performance.now();
   d  Diff_flushed = start-end // Retrieval time of data from memory
   e  Start = window.performance.now();
   f  Access the same ith page
   g  end = window.performance.now();
   h  Diff_Unflushed = start-end     // Retrieval of data from LLC
   i  Start = window.performance.now();
   j  Access the same ith page
   k  end = window.performance.now();
```

```
    l   Diff_Unflushed_Test = start-end     // Testing of retrieving data from Unflushed LLC
    m   Flush out LLC cache to Prime eviction set
    n   Start = window.performance.now();
    o   Access the same ith page
    p   end = window.performance.now();
    q   Diff_flushed_Test = start-end       // Testing of retrieving data from flushed LLC
    r   AccessTime_FlushedLLC = AccessTime_FlushedLLC + Diff_flushed
    s   AccessTime_UnFlushedLLC = AccessTime_UnFlushedLLC + Diff_Unflushed
    t   AccessTime_UnFlushedLLC_Test = AccessTime_UnFlushedLLC_Test + Diff_Unflushed_test
    u   AccessTime_FlushedLLC_Test = AccessTime_FlushedLLC_Test + Diff_flushed_Test
4   Find Fulshed_Avg = AccessTime_FlushedLLC/k
5   Find UnFulshed_Avg = AccessTime_UnFlushedLLC/k
6   Find UnFulshed_Avg_Test = AccessTime_FlushedLLC_Test/k
7   Find Fulshed_Avg _Test = AccessTime_FlushedLLC_Test/k
```

The BW detection system will detect the attack with the help of the TIMER_COUNT variable value. If the TIMER_COUNT variable value is greater than MAX_COUNT, then the average cache miss rate (CMR) is again calculated by using our existing model (Sangeetha and Ganesan, 2020). If the average CMR value is greater than the lower bound (LB) threshold value and lesser than the upper bound (UB) threshold value, then the modified performance.now() function will be called (Listing: 1) to reduce the efficiency of the attacker. If the average CMR is greater than the UB then the user accepts that the cache flush function will be called by the BW system to prevent the attacker. Then the attacker could not find the next memory pattern of the victim in their next iteration. Figure 3 shows how the normal, blocked, modified and with user permission wrapper java script function works in the BW system.

Figure 3 Workflow model for BW system, (a) normal function, (b) blocked function, (c) modified function, (d) function with user permission

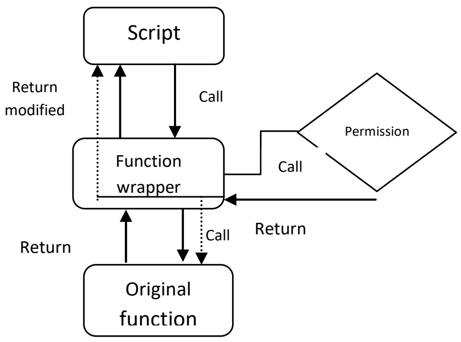


4.1 *Cache access time measurement*

The attacker can find the memory access blueprint of the victim only if the attacker has accessed the cache repeatedly after the operation of the victim. If any other user has accessed the cache memory in the meanwhile of the wait function of the attacker (Algorithm 1: Step4a), then the attacker will not be able to find the memory access pattern of the victim. So, calculate the access time of the flush and unflushed cache memory with the help of Algorithm 3. In Figure 4, the proposed BW protected system model, the permission system performs as an abstraction layer between the interfaces offered to a JS developer and the JS engine.

Created a buffer with a total size of 131 k cache lines across 8,192 cache sets. There are more than 12 cache lines in a cache set. Therefore, we could emit a variable from the cache by filling it with 12 cache lines. As per the second step of the Prime + Probe attack, we could fill-up the LLC cache set by JS code. Created an 8 MB buffer array in JS and iterated over the array to replace the data in the cache set with the data in the 8 MB buffer array.

Figure 3 Workflow model for BW system, (a) normal function, (b) blocked function, (c) modified function, (d) function with user permission (continued)



(d)

Algorithm 3 is used to measure the access time of a variable from the flushed cache set and unflushed cache set. Repeated tests have been taken to confirm the access time of the flushed and unflushed cache LLC.

Figure 4 Proposed BW protected system model (see online version for colours)

JavaScript engine	
Conditions, Loops, Arithmetic,…	Objects, functions, · · ·
	Permission system
(Malicious) JavaScript Code	

4.2 *Detection of cache based timing-attack*

A BW system is developed to detect timing-based probe attacks in web applications. The objective of the proposed system is to examine every web application run time behaviour and to identify timing-based attacks. Most CSC attacks happen with the help of the timestamp function which gives information to the attacker about the memory access pattern of the victim. In the prime + probe attack, the memory footage of the victim will be identified by invoking the timer function in the probe stage. Used the virtual machine layering (VML) methodology to provide security and enlarge it for objects using proxy objects. The VML was established for low-overhead dynamic monitoring of functions (Lavoie et al., 2014).

Listing 1 VML method to call the original function via reference

```
var actual_ref = window.performance.now;
window.performance.now = function () {return 0;};
alert(window.performance.now()); // call new function via
function name
alert (actual_ref.call(window.performance.now )); // call the
original function via reference
```

Listing 2 Modified performance.now function

```
(function (){
var timer = window.performance.now ();
window.performance.now= function () {
return Math.floor ((timer.call(window.
Performance))/1000.0) * 1000.0"}
};
})();
```

- Listing 1: show that the VML calls the window. performance.now() function through the reference variable. With the help of VML technology without modification of a web browser, we can call the function instead of the original function.
- Listing 2: show that the modified performance.now function. This function will be called if the TIMER_COUNT value is crossed by the MAX_COUNT value.

5 Experimental results

The BW system is implemented in Windows and Ubuntu14.04. The outline of the proposed BW system is explained in Pseudcode 1. Initially, The Prime + Probe and Flush + Reload attacks are implemented in JS as per Algorithms 1 and 2. In the prime stage, a buffered array is created as the size of an LLC and filled by the data of the attacker. In the probe stage, the attacker waits until the victim completes his operation. The attacker finds the current time by the function of performance.now(). While the attacker accesses any random page from the buffer, the access time is measured. The access latency shows whether the page was accessed by the victim.

Pseudocode 1 Outline of the proposed BW system

```
1  Monitoring and extracting the runtime behaviour of the web
   user.
2  TIMER_COUNT variable is incremented based on Listing:
   III
3  If(TIMER_COUNT < MAX_COUNT)
       Identified as normal user.
   else
       Average CMR is calculated
4  If (average CMR = = LB_Threshold )
       Identified the user as Attacker
       Implemented Listing II to reduce the efficiency of the
       attacker.
   else if ( average CMR = = UB_Threshold )
       The cache is flushed as per user conformation.
```

5.1 Stage 1: extracting runtime behaviours

The base of the proposed system is to monitor the behaviour of the web user by analysing every website at runtime. The BW system intercepts JS API calls to characterise the behaviour of a web application because most of the web application dynamic behaviours are written in JS. The BW system records the API together with its parameters. To recognise the user behaviour and distinguish APIs, the BW system extracts the runtime JS stack of the API. Need to extract the runtime behaviours without modifying the web browser, so build the BW system as a browser extension shown in Figure 5 and Figure 6. Since it focuses only on timing-related behaviours, the interception mechanism is flexible to allow the users to specify General JS APIs to monitor. Lavoie et al. (2014) proposed a VML technique to monitor JS applications at runtime.

Listing 3 Detection and prevention model

```
(function () {
var original_ref = window.performance.now;
window.performance.now = function (){
COUNT=COUNT + 1;
if (LevelI_TH < COUNT)
Low-resolution Timer ()
else if (LevelII_TH < COUNT)
return Clear_LLC;
else {return original_ref.call(window.performance.now);
};}
Clear_LLC = function (){ // we use this buffer to evict everything from LLC
var evictionBuffer = new ArrayBuffer (8192 * 1024); // 8MB buffer
var evictionView = new DataView (evictionBuffer);
var current,offset = 64;
for (var i = 0; i < ((8192 * 1023) / offset); i++) {
    current = evictionView.getUint32(i * offset);
})();
};}
```

5.2 Stage 2: detection of the attacker

The proposed BW system detects the attacker and avoids leaking any memory footage information about the victim. The BW system continuously monitors the behaviour of the user by watching the TIMER_COUNT variable. The TIMER_COUNT variable will be incremented as per the number of times the performance.now() function is invoked by the web user (Listing: 3). If the TIMER_COUNT value is lesser than the MAX_COUNT variable, then the user is treated as a normal user. Otherwise, the average CMR is calculated again. If it reaches the LB_Threshold variable value, then the user is treated as an attacker. The modified performance.now() function will be invoked and it returns a low-resolution timestamp (Listing 2) to reduce the attacker's efficiency.

5.3 Stage 3: getting user confirmation

If the average CMR value is crossed over the UB_Threshold value, then the user confirmation is invoked to flush the cache. The attacker has to get opportunities to access the LLC to find the memory access pattern of the victim after the immediate use of the cache memory of the victim. Nevertheless, the BW system flushed out the cache memory to remove the memory footage of the victim before the attacker accessed the LLC. Evict the corresponding data from all cache levels by using the clflush instruction. But the clflush instruction implementation is not possible in JS. The BW system reads the same 8 MB buffer array when needed to execute clflush instruction. If the attacker gets the cache miss in the probe stage, then the attacker assumes that it was accessed by the victim.

Algorithm 3 states that the access time of the average flushed test rate is larger than the average unflushed test rate. Due to the cache clflush instruction, the CMR will be high for the victim. The average memory access time (AMAT) is calculated as follows,

$$AMAT = HT + MR * CMP \tag{1}$$

where the hit time, miss rate, and cache miss penalty are denoted as *HT*, *MR*, and *CMP* respectively. CMP is defined as the additional processor halt caused by the next-level of cache access or memory access. The BW system is implemented on the Chrome web browser and evaluates the effectiveness of the BW system approach using malicious probing websites. The BW system efficiency is measured in terms of the cache MR of the attacker.

Figure 7 shows the Avg_CMR of different data sizes at Ti. The same data is accessed at T_{i+1} and Avg_CMR is calculated. If none of the timing attacks have occurred in the interval of T_i and T_{i+1}, then the T_i(Avg_CMR) > T_{i+1} (Avg_CMR).

As per Algorithm 2, the average of flushed and unflushed variable access time is measured. Algorithm 2 illustrates that the CMR is high in the flushed cache memory. Figure 8 shows the access latency between the flushed and unflushed variables. In comparison, the flushed variable access latency is higher than the unflushed variable.

Once the attacker is identified with the help of the BW system, the LLC cache memory is flushed out and the memory footage of the victim is removed. So, both the attacker and victim CMR will be higher. Figure 9 demonstrates the protected and unprotected mode Computation overhead of the JS engine. The result of the proposed BW system with the different domains is shown in Table 3. The BW extension is enabled while visiting Alexa Top websites and measuring protection levels. Figure 10 shows the CMR and cache hit rate in the chrome web browser. Even though the victim cache performance was low due to the exploitation of the attacker, the secret data of the victim is secured.

Figure 5 BW developer system (see online version for colours)

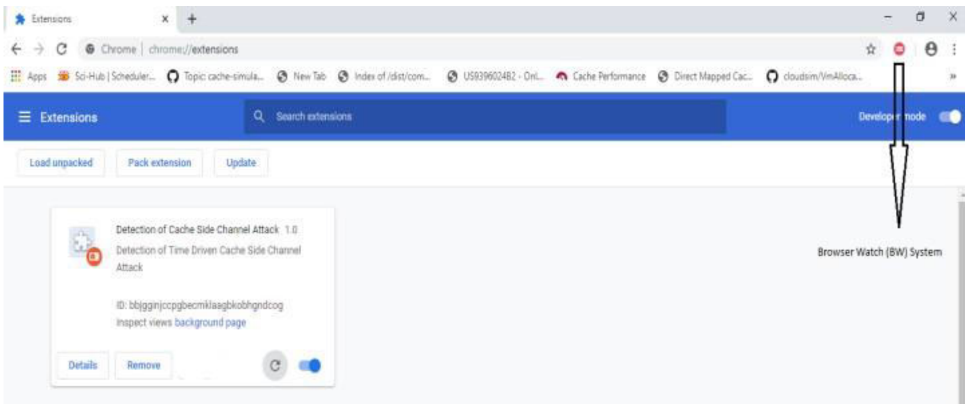

Extensions
Search extensions
Developer mode
Load unpacked
Pack extension
Update
Detection of Cache Side Channel Attack 1.0
Detection of Time Driven Cache Side Channel Attack
ID: bbjgginjccpgbecmklaagbkobhgndcog
Inspect views background page
Details
Remove
Browser Watch (BW) System

Figure 6 BW developer system (see online version for colours)

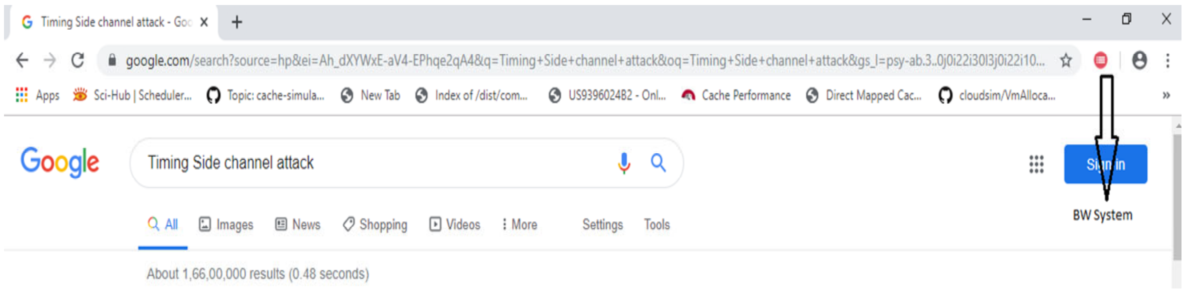

Timing Side channel attack - Goo
Google
Timing Side channel attack
All
Images
News
Shopping
Videos
More
Settings
Tools
About 1,66,00,000 results (0.48 seconds)
Sign in
BW System

Figure 7 Average CMR in timing attack (T_i, T_{i+1}) (see online version for colours)

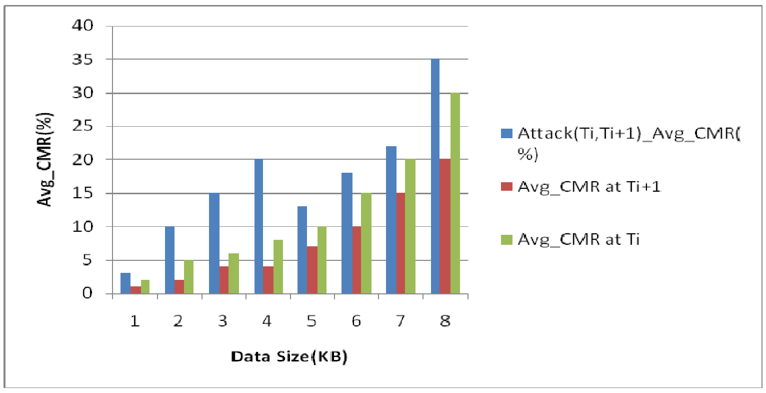


Figure 8 Access latency between flushed and unflushed variables in BW (see online version for colours)

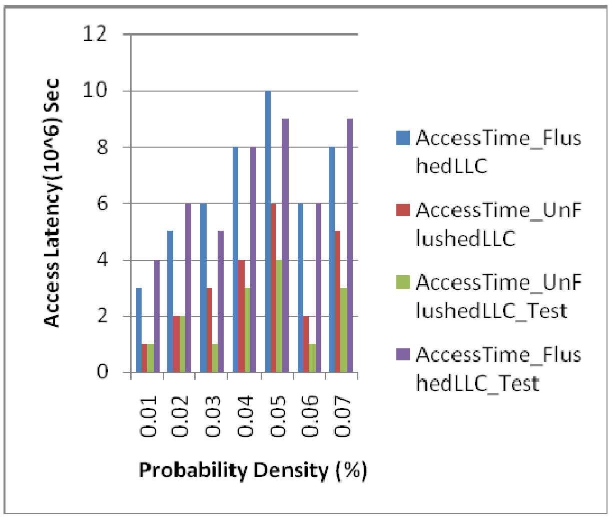


Figure 9 Computation overhead of the java script engine

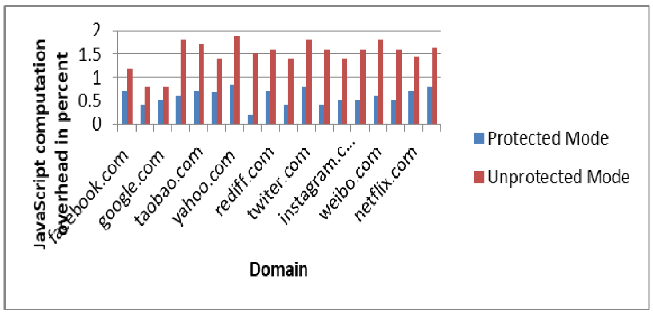


Figure 10 BW system – cache performance (see online version for colours)

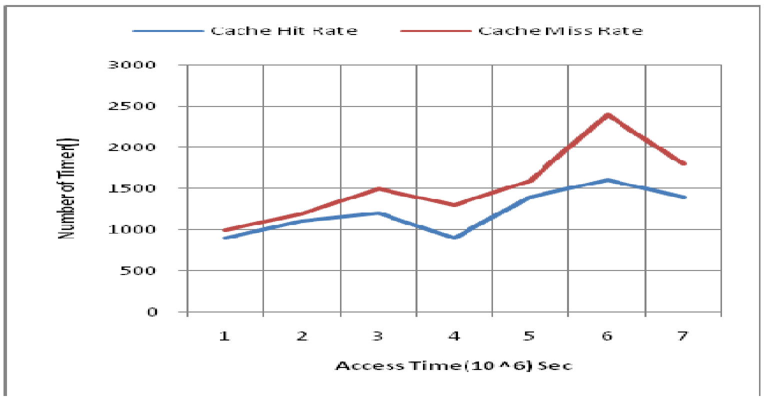


Table 3 Result of proposed BW system with different domain

Domain	*Avg_CMR %*	*TIMER_COUNT*	*Permission for wrapper function*	*Permission for original function*
facebook.com	47	57		√
google.co.in	87	84	√	
google.com	94	97	√	
reddit.com	69	35		√
taobao.com	78	59	√	
wikipedia.org	45	67		√
yahoo.com	84	76	√	
baidu.com	89	81	√	
rediff.com	23	32		√
amazon.com	94	96	√	
twiter.com	87	90	√	
live.com	85	84	√	
instagram.com	76	87	√	
sina.com.cn	28	24		√
weibo.com	67	79	√	
inkedin.com	74	83	√	
netflix.com	91	94	√	
ebay.com	89	84	√	

6 Conclusions

The proposed system monitors and avoids the time-based cache side-channel attack. The system avoids all timestamp-related attacks. Most of the existing defences against time-based attacker systems are providing a solution using a zero-timer function. Normal users also get affected by the solution of the existing system. The proposed work reduces the cache performance of the victim while the attacker frequently tries to access his/her memory pattern. Instant of concentrating on the cache performance, the BW system focuses on the security of the secret key of the victim user. Once the attacker starts to steal the data of the victim, then the cache LLC will be flushed out to protect the secret key of the victim. The CMR will be high for the victim, as the memory footage is saved from the attacker.

References

Agrawal, D., Archambeault, B., Rao, J.R. and Rohatgi, P. (2003) 'The EM side-channel(s)', in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, Shores, CA, USA, 13–15 August 2002, Springer: Berlin/Heidelberg, Germany.

Bernstein, D.J. (2004) *Cache-Timing Attacks on AES* [online] http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

Bortz, A. and Boneh, D. (2007) 'Exposing private information by timing web applications', in *WWW'07*.

Brier, E. and Joye, M. (2002) 'Weierstraß elliptic curves and side-channel attacks', *Public 42Key Cryptography*, Springer: Berlin/Heidelberg, Germany.

Cai, L. and Chen, H. (2011) 'TouchLogger: inferring keystrokes on touch screen from smartphone motion', in *USENIX Workshop on Hot Topics in Security – HotSec*.

Chen, Z. and Cao, Y. (2020) 'Jskernel: fortifying javascript against web concurrency attacks via a kernel-like structure', *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE.

Felten, E.W. and Schneider, M.A. (2000) 'Timing attacks on web privacy', in *CCS*.

Gras, B., Razavi, K., Bosman, E., Bos, H. and Giuffrida, C. (2017) 'ASLR on the line: practical cache attacks on the MMU', in *NDSS'17*.

Gruss, D., Bidner, D. and Mangard, S. (2015) 'Practical memory deduplication attacks in sandboxed javascript', in *ESORICS'15*.

Gruss, D., Maurice, C. and Mangard, S. (2016) 'Rowhammer.js: a remote software-induced fault attack in JavaScript', in *DIMVA'16*.

Heiderich, M., Niemietz, M., Schuster, F., Holz, T. and Schwenk, J. (2012) 'Scriptless attacks: stealing the pie without touching the sill', in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, pp.760–771.

Hu, W-M. (1992a) 'Lattice scheduling and covert channels', in *1992 IEEE Computer Society Symposium on Research in Security and Privacy*, IEEE Computer Society, Oakland, CA, USA, 4–6 May, pp.52–61.

Hu, W-M. (1992b) 'Reducing timing channels with fuzzy time', in *Proceedings of IEEE Security and Privacy ('Oakland')*, Jthenal of Computer Security, Vol. 1, Nos. 3–4, pp.233–254.

Jang, D., Jhala, R., Lerner, S. and Shacham, H. (2010) 'An empirical study of privacy violating information flows in javascript web applications', in *CCS'10*.

Jia, Y., Dong, X., Liang, Z., Saxena, P. (2015) 'I know where you've been: geo-inference attacks via the browser cache', *IEEE Internet Computing*, Vol. 19, No. 1, pp.44–53.

Kocher, P.C. (1996) 'Timing attacks on implementations of Diffe-Hellman, RSA, DSS, and other systems', in *Crypto'96*, pp.104–113.

Kotcher, R., Pei, Y., Jumde, P. and Jackson, C. (2013) 'Cross-origin pixel stealing: timing attacks using CSS filters', in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, Berlin, Germany, 4–8 November.

Lavoie, E., Dufthe, B. and Feeley, M. (2014) 'Portable and efficient run-time monitoring of javascript applications using virtual machine layering', in *European Conference on Object-Oriented Programming*.

Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C. and Mangard, S. (2017) 'Practical keystroke timing attacks in sandboxed JavaScript', in *ESORICS'17*.

Mangard, S., Oswald, E. and Popp, T. (2007) *Power Analysis Attacks – Revealing the Secrets of Smart Cards*, Vol. 31, Springer Science &Business Media.

Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S. and Romer, K. (2017) 'Hello from the other side: SSH over robust cache covert channels in the cloud', in *NDSS'17*.

Mehrnezhad, M., Toreini, E., Shahandashti, S.F. and Hao, F. (2016) 'Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript', *Jthenal of Information Security and Applications*, Vol. 26, pp.23–38.

Olejnik, L. (2017) *Stealing Sensitive Browser Data with the W3C Ambient Light Sensor API* [online] https://blog.lukaszolejnik.com/stealing-sensitive-browser-datawith-the-w3c-ambient-light-sensor-api (accessed March).

Oren, Y., Kemerlis, V.P., Sethumadhavan, S. and Keromytis, A.D. (2015) 'The spy in the sandbox: practical cache attacks in JavaScript and their implications', in Kruegel, C. and Li, N. (Eds.): *Proceedings of CCS 2015*, ACM Press, Oct.

Osvik, D.A., Shamir, A. and Tromer, E. (2006) 'Cache attacks and countermeasures: the case of AES', in Pointcheval, D. (Ed.): *Topics in Cryptology-CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, San Jose, CA, USA, 13–17 February, Proceedings, Vol. 3860 of Lecture Notes in Computer Science, Springer, pp.1–20.

Patil, K., Dong, X., Li, X., Liang, Z. and Jiang, X. (2011) 'Towards fine-grained access control in javascript contexts', in *31st International Conference on Distributed Computing Systems (ICDCS)*.

Popescu, A. (2016) *Geolocation API Specification 2nd Edition* [online] https://www.w3.org/TR/geolocation-API (accessed June 2016).

Rokicki, T., Maurice, C. and Laperdrix, P. (2021) 'Sok: in search of lost time: a review of javascript timers in browsers', *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE.

Sangeetha, G. and Ganesan, S. (2020) 'An optimistic technique to detect cache based side channel attacks in cloud', in *Peer-to-Peer Networking and Applications*, Vol. 14, No. 4, pp.2473–2486, https://doi.org/10.1007/s12083-020-00996-1.

Sangeetha, G. and Ganesan, S. (2021) 'A multi-objective secure optimal VM placement in energy-efficient server of cloud computing', *Intelligent Automation &amp; Soft Computing*, Vol. 30, No. 2, pp.387–401, DOI: 10.32604/iasc.2021.019024.

Schwarz, M., Maurice, C., Gruss, D. and Mangard, S. (2017) 'Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript', in *FC'17*.

Stone, P. (2013) *Pixel Perfect Timing Attacks with HTML5; Context Information Security*, White Paper, London, UK.

Van Goethem, T., Joosen, W. and Nikiforakis, N. (2015) 'The clock is still ticking: timing attacks in the modern web', in *CCS'15*.

Vila, P. and Kopf, B. (2017) 'Loophole: Timing attacks on shared event loops in chrome', in *USENIX Security Symposium*.

W3C (2016a) *Battery Status API* [online] https://www.w3.org/TR/battery-status/.

W3C (2016b) *High Resolution Time Level 2* [online] https://www.w3.org/TR/hr-time/.

W3Techs (2017) *Usage of JavaScript for Websites*, Aug. [online] https://w3techs.com/technologies/details/cp-javascript/all/all (accessed July 2017).

Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C. (2011) 'I still know what you visited last summer: leaking browsing history via user interaction and side channel attacks', in *S&P'11*.

Yarom, Y. and Falkner, K. (2014) 'Flush+Reload: a high resolution, low noise, L3 cache side-channel attack', in *USENIX Security Symposium*.