The ISC Int'l Journal of Information Security

ISeCure

January 2022, Volume 14, Number 1 (pp. 27–46)
http://www.isecure-journal.org

A Study of Timing Side-Channel Attacks and Countermeasures on JavaScript and WebAssembly [**]

Mohammad Erfan Mazaheri [1], Siavash Bayat Sarmadi [1,*], and Farhad Taheri Ardakani [1]
[1] *Sharif University of Technology, Department of Computer Engineering, Tehran, Islamic Republic of Iran.*

ARTICLE INFO.

Article history:
Received: December 24, 2020
Revised: May 6, 2021
Accepted: May 10, 2021
Published Online: September 6, 2021

Keywords:
Timing Side-Channel Attacks, JavaScript, WebAssembly, Malicious Code Detection, Timers

Type: Research Article

doi: 10.22042/isecure.2021.263565.599

dor: 20.1001.1.20082045.2022.14.1.3.3

ABSTRACT

Side-channel attacks are a group of powerful attacks in hardware security that exploit the deficiencies in the implementation of systems. Timing side-channel attacks are one of the main side-channel attack categories that use the time difference of running an operation in different states. Many powerful attacks can be classified into this type of attack, including cache attacks. The limitation of these attacks is the need to run the spy program on the victim's system. Various studies have tried to overcome this limitation by implementing these attacks remotely on JavaScript and WebAssembly. This paper provides the first comprehensive evaluation of timing side-channel attacks on JavaScript and investigates challenges and countermeasures to overcome these attacks. Moreover, by investigating the countermeasures and their strengths and weaknesses, we introduce a detection-based approach, called Lurking Eyes. Our approach has the least reduction in the performance of JavaScript and WebAssembly. The evaluation results show that the Lurking eyes have an accuracy of 0.998, precision of 0.983, and F-measure of 0.983. Considering these values and no limitations, this method can be introduced as an effective way to counter timing side-channel attacks on JavaScript and WebAssembly. Also, we provide a new accurate timer, named Eagle timer, based on WebAssembly memory for implementing these attacks.

1 Introduction

Among the attacks on hardware security, side-channel attacks are one of the most powerful attacks. These attacks are used to exploit the deficiencies in the implementation of systems, regardless of their theoretical flaws. These attacks are based on leaked information from system implementation. One of the most common side-channel attacks is timing side-channel attacks that use the time difference of running an operation in different states. These attacks are used to break cryptographic algorithms, read the victim's secret information, and fault injecting. Cache attacks are a kind of timing side-channel attacks in which the attacker steal the victim data by CPU cache [1–3]. Although these attacks have high power in extracting the victim's cryptographic key, there is an important limitation to them that the attacker program must run on the victim's system. For this reason, since 2015, various studies have implemented these attacks remotely, on JavaScript and WebAssembly platforms [4, 5].

* Corresponding author.
**This article is an extended/revised version of an ISCISC'17 paper.

Email addresses: mazaheri@ce.sharif.edu, sbayat@sharif.edu, farhadtaheri@ce.sharif.edu

ISSN: 2008-2045 © 2020 ISC. All rights reserved.

Different countermeasures are proposed at three levels of hardware, operating system, and software to counter timing side-channel attacks on JavaScript and WebAssembly. The hardware-level countermeasures [5–8] will generally be very costly because they are usually associated with a change in architecture and structure, but they will be very powerful. The operating system level countermeasures [4, 8–10] are also usually associated with structural changes. Also, some methods of this level will deprive the user of the benefits of some JavaScript features. The software level's countermeasures are very flexible and practical so in many related work, several countermeasures are introduced [5, 9, 11–13]. Despite the benefits of software methods, no effective method has been introduced so far that it does not prevent the performance of JavaScript. Such a countermeasure can be implemented with a detection-based approach.

This work is the first survey and comprehensive evaluation on timing side-channel attacks on Javascript and WebAssembly platforms. The various aspects of these attacks are investigated in this paper. Moreover, in this word, we introduce: 1) a new accurate timer, based on WebAssembly memory timer, named Eagle timer, and 2) a method to detect timing side-channel attacks on JavaScript and WebAssembly, named Lurking Eyes.

Lurking Eyes is presented as an effective way to counter timing side-channel attacks on JavaScript and WebAssembly. To implement our approach, we have investigated timing side-channel attacks implemented in different ways on JavaScript and WebAssembly. We then have extracted the features and functions that lead us to detect these attacks. The evaluation results show that Lurking Eyes has high accuracy and does not impose any restrictions on the JavaScript. Note that our detection method (Lurking Eyes) first introduced in [14] and this work is an extended version of [14].

The contributions of this paper can be summarized as follows:

- The first comprehensive evaluation on Javascript and WebAssembly timing side-channel attacks
- Presenting a new accurate timer, based on WebAssembly memory, named Eagle timer
- Introducing Lurking Eyes, a detection based countermeasure for timing side-channel attacks on JavaScript and WebAssemly. The Lurking eye has an accuracy of 0.998, precision of 0.983, and F-measure of 0.983.

In the following, first, we explain background such as CPU cache, memory deduplication, speculative execution, and JavaScript and WebAssemly platforms in Section 2. Side-channel attacks, especially timing side-channel attacks, are investigated in Section 3. In Section 4, we perform an overview on the side-channel attacks implemented on JavaScript and WebAssembly. The challenges of timing side-channel attacks are discussed in topics timers and eviction strategies in Section 5 and Section 6. We introduce the countermeasures for timing side-channel attacks on JavaScript and WebAssembly in Section 7. Then, in this section, to overcome the weaknesses of the previous countermeasures, we present and evaluate our detection method, named Lurking Eyes. Finally, we conclude and introduce future works in Section 8.

2 Background

In this section, a brief review of caches in modern CPUs as a structure used in many side-channel attacks is made. The concepts of memory deduplication and speculative execution are discussed, and finally, an overview of JavaScript and WebAssembly languages is described.

2.1 CPU Cache

Calling data from memory is considered a bottleneck. To prevent this bottleneck, modern systems are created hierarchically. This hierarchy is made up of memories at different types, speeds, and capacities.

Modern processors use hierarchical caches. The highest cache level (L1) is the nearest memory to the CPU and is smaller and faster than the two others. L1 usually consists of two caches: Instruction Cache and Data Cache. Typically, in modern x86 CPUs, an intermediate cache, named L2 is used. The third level of cache (Last Level Cache: LLC) is shared between all the cores of a multi-core chip and maintain both instructions and data. The CPU to fetch data from main memory, first, access the L1 cache, and if the data were not at the L1, the request would be sent down in the memory hierarchy to hit in the cache or the main memory. L1 cache is usually indexed with virtual addresses, and other levels are indexed with physical addresses. To fetch data from memory, the CPU checks the existence of data in the cache. If the result is positive, it is called cache hit, otherwise, it is called cache miss. Figure 1 shows the cache hierarchy. The cache structure is the basis of many timing side-channel attacks [1–3, 15–17].

2.2 Memory Deduplication

In many cases, especially when we have the same operating system and several virtual machines, the contents that are mapping from these virtual machines to the main memory are the same. Memory deduplication merges these similar contents in the main memory. The way memory deduplication works

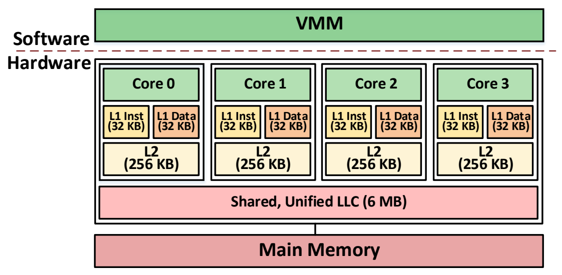

Figure 1. Cache hierarchy in a quad-core processor that each core [2, 3]

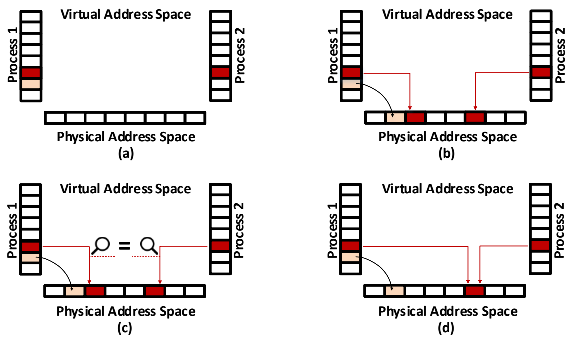

Figure 2. The process of memory deduplication

are to share memory by scanning through the main memory and finding duplicate pages. Each duplicated page is merged into one page and mapped into both original locations. The process of memory deduplication is shown in Figure 2. When one of the processes shared in one place tries to write into the shared memory region, the Copy-on-Write page fault will happen. Thus the operating system makes a copy of that region [4, 18–20]. This page fault causes a time difference in the run time and is used by the attacker to steal private information. It is the basis of memory deduplication attack.

2.3 Speculative Execution

Speculative execution can be considered as a key to optimizing run-time efficiency used by modern CPUs. Speculative execution results in increased efficiency by guessing the probable execution path. The basis of speculative execution is to predict the following path of the program. The code of this path is executed before it is identified whether it is the path that must be executed now to prevent the delay of waiting for doing the work after it is known that it is needed. If it is determined that the work was not needed, after all, the changes made by this path are reverted. Given this feature and misdirecting the branch prediction unit, we will leak some information through CPU cache [7]. This is done by executing a specific code in the victim program and misleading the branch unit.

2.4 JavaScript and WebAssemly Overview

JavaScript is a lightweight, dynamic, and object-oriented language with a run-time evaluation that empowers the modern web on the client-side. JavaScript codes, once received by the browser, will be compiled and optimized by a just-in-time mechanism.

Today, JavaScript can be executed not only in the browser, but also on the server, and generally on any device that has a JavaScript engine. JavaScript engine is a computer program that executes JavaScript codes. The function of the JavaScript engine is reading the script, converting it to the machine language, and then running the machine code. JavaScript was designed to run in a single-threaded environment. Browser vendors introduced Web Workers to simulate multi-threading. A web worker is a JavaScript that is executed in the background, separate from other scripts [21–24].

WebAssembly is a low-level and assembly-like language that can be run in modern browsers, with performance near to native codes. By compiling some language codes like C/C++ and Rust to WebAssembly, they can be executed on the web. The Chrome 70 release supports multi-threading for WebAssembly, experimentally. JavaScript and WebAssebly platforms have been used to implement remote timing side-channel attacks due to their unique features.

2.5 WebAssembly Memory

WebAssembly memory is a resizable ArrayBuffer that contains a linear array of bytes that can be read or written. Note that the ArrayBuffer is an object used to represent a fixed-length raw binary data buffer [25]. The read and write process is performed by WebAssembly low-level memory access instructions. WebAssembly memory is accessed by the WebAssembly.Memory object. One difference between WebAssembly memory and JavaScript memory is the ability to access bytes in memory rows in web assembly directly.

3 Side-Channel Attacks

A bunch of common and powerful attacks in cryptography, that exploits the information obtained from the implementation of cryptography systems, is named side-channel attacks [26–30]. This type of attack does not have much attention to the theoretical defects of algorithms. The first purpose of most of these attacks is to find the secret cryptography key.

Side-channel information that is interesting in these attacks consists of power consumption, electromagnetic radiation, execution time, and in fewer cases, optic and acoustic information [26–28].

Generally, side-channel attacks are categorized into two categories: active and passive. In active side-channel attacks, we typically try to change the cryptography system's side entries, like power supply voltage and temperature, to generate errors in the system. The effect of this error will be investigated on the main and side circuits. On the contrary, passive side-channel attacks don't exploit side inputs and only consider the side outputs of circuit.

In the following, we will focus on timing side-channel attacks, especially cache attacks and Spectre attack. There will also be some discussion about fault attacks after that.

3.1 Timing Side-Channel Attacks

Timing channels are communication channels that exploit time difference, to transmit information to a receiver or decoder [31]. Implementations of cryptographic algorithms usually run the calculations at a fixed time. If this operation deals with secret parameters and results in a time difference, this time difference can lead to information leakage. With the precise statistical analysis of this time difference, the victim's secret information, including the private key, can be obtained. DRAM-based attacks and Cache attacks are two types of the most common side-channel attacks. What is important in these attacks is to recognize the presence or absence of certain data in memory. Two basic elements of these attacks to reach this goal are time measuring and cache eviction. Time measuring is necessary to recognize cache hit from cache miss in caches and row hit from row conflict in DRAM. Cache eviction is the process of victim data eviction from memory by calling several data with specified addresses [1–3].

One of the main disadvantages of cache-based timing side-channel attacks is the necessity of running the spy program on the victim's system. In cache attacks, the spy and victim should reside in one system or even in one physical core. To overcome this limitation, some researches have used JavaScript and WebAssembly as two appropriate platforms [4–6, 12, 18, 32–36]. With these platforms, the spy program can easily run in the victim system.

3.1.1 Cache Attacks

Cache Attacks are one of the well-known categories of side-channel attacks that search to find information about the memory locations that have been accessed by a victim program [1, 2, 37–44]. Cache attacks exploit this fact that the accesses of victim process to memory an be monitored by a spy process, if a cache is shared between them. These attacks initially used L1 cache, so there was a need for co-residency of attacker and victim on the same core, but recent attacks have shifted from L1 cache to LLC that is shared between all cores. So these attacks can be performed between virtual machines [2, 3, 38, 45–49].

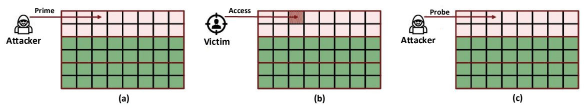

Figure 3. Prime+Probe Attack; (a) Priming the intended cache set by attacker with the data that is suspected to be accessed by victim, (b) Victim access to the intended cache set, (c) Probing the cache sets again by attacker [50]

The basis of many memory-based attacks is checking the time and detecting cache hit from cache miss; if execution time is less than one threshold, it shows the access of the victim to a specific data or code, and this matter can result in cryptanalysis or user behavior tracking.

One of the most fundamental cache attacks is the Prime+Probe attack that first was implemented on L1 cache in 2005 [1] and ten years later on LLC [2, 47]. The steps of this attack can be summarized as follows (see Figure 3):

(1) Creating an eviction set for one or more related set of cache and priming the cache set by an attacker with the data that is suspected to be accessed by the victim (Prime)
(2) Triggering victim operation by the attacker and wait (Wait)
(3) Probing the cache sets again (Probe).

In addition to cross-core and cross-vm cache attacks [29, 48, 51–53], cross-processor cache attacks constitute another field of research that has been increasing interest among researchers [54, 55].

3.1.2 Spectre Attack

Spectre is the serious security bugs found in modern processors [7]. This attack is a method of reading data from another program's address space that runs on the processor kernel. In the attack, by misleading the branch prediction unit, a specific code is executed in the victim's program, and the attack will occur. In this attack, the victim's confidential information will be leaked to the attacker.

Spectre is based on speculative execution. Given this feature and misdirecting the branch prediction unit, we will leak some information through CPU cache [7]. The attacker then read this information from the cache with a kind of flush+reload attack.

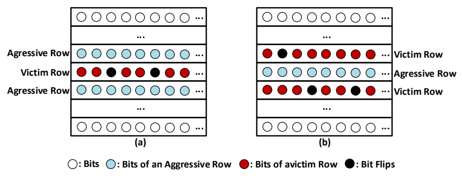

Figure 4. Rowhammer; (a) sigle-sided, (b) double-sided

So far, various forms of Spectre attack have been published [56–58].

3.2 Fault Attacks

The idea of Fault Attack is originated from the suggested methods of two papers [59, 60]. A fault attack typically needs physical access to the victim device and consists of stressing that device with an external mean to generate errors so that these errors results in a security fault in the system; in fact in the fault attacks we are facing some intentional modifications to expose the device, some out of specification physical conditions, like high or low temperature and radiation. However, software-induced physical fault attack is also possible [61].

Rowhammer is a software-induced hardware fault that is effective on DRAM chips. The increasing intensity of DRAM ICs conduces smaller memory cells that can save less charge. This is the cause of lower margins related to operational noise and increases the electromagnetic interaction rate between cells, making data loss more likely. As a result, some annoying errors have been observed, originated from interfering of cells, and arrival of random changes in bits values stored in the affected memory cells. In Rowhammer, as a hardware trust problem, the adversary accesses DRAM cells frequently to make unauthorized changes in contiguous locations in the memory [6, 62–64, 64, 65]. The single-sided and double-sided Rowhammer and their description are shown in Figure 4.

4 Timing Side-Channel Attacks on JavaScript and WebAssembly Performs

JavaScript language is an appropriate platform for performing timing side-channel attacks because it provides the possibility of performing attacks remotely and independent of the CPU and operating system. Also, JavaScript is enabled by default in modern browsers. It was suggested to use WebAssembly language to improve the level of timing side-channel attacks on JavaScript [4–6, 32].

Two common elements of timing side-channel attacks on JavaScript and WebAssembly are time measuring and cache eviction. These operations are performed in native attacks by native instructions like clflush and rdtsc, but it is impossible to execute native code in sandboxed JavaScript. On the other hand, there is no access to files and system services, and no notion of pointers in this language [4, 6]. The methods to overcome these limitations are investigated in Section 5 and Section 6. Before that, we will investigate timing side-channel attacks on JavaScript and WebAssembly.

Cache Attack on JavaScript. In 2015, [5] presented the first practical cache attack on the JavaScript platform. This attack is performed on LLC. They implemented the Prime+Probe attack on JavaScript and exploit it to track the user behavior, like the websites that user browses. Although the paper claimed that tracking user behavior is more related to the nature of the remote cache attack, it is important to know that this attack cannot break the cryptographic keys.

In 2018, [34] improved the level of remote cache attacks to cryptographic key extracting. This attack can extract cryptography keys from the ElGamal and RSA and even Curve25519 Elliptic Curve Diffie-Hellman. This attack needs to have higher accuracy than previous ones. To reach this accuracy to attack ElGamal and ECDH, they used PNACL and Web Assembly platforms. The attack consists of two phases:

(1) *Online phase*: The goal of this phase is to collect tracks of victim access to precomputed multipliers. To perform this phase, we must open a web page in browser for PNACL code and another page for running ecryption operation. Eight random cache sets must be selected and be monitored in parallel. Finally, the Prime+Probe cycle must be run in each cache set, and also, the process must be iterated.

(2) *Offline phase*: Processing the collected tracks to extract the key is the aim of the offline phase. At first, the adversary determines which tracks correspond with the memory accesses to precomputed multipliers. Then matches the tracks with the algorithm and recognizes the limitations on the tracks-based possible sequence. Finally combines the limitations and finds the order of accesses to multipliers.

In 2019, [36] presented a website fingerprinting attack through the cache occupy channel that works in two scenarios:

(1) *Cross-tab scenario*: A scenario like what be mentioned about [5] and [4] attacks that use a website with malicious JavaScript and the process of learning the intended websites, while the malicious website is open.

(2) *Cross-network scenario*: In this scenario, the adversary is considered an on-path attacker that can inject intended JavaScript code into the pages. To clarify, assume a user with a session over an unsecured connection for his ordinary and non-secret works and another session for over a secured connection for his secret and sensitive works. An attacker can alter the ordinary session's traffic and as a result understand what are being doing in other sessions.

This attack monitors the whole cache, despite the previous ones that focused on special sets of the cache. Though cache occupancy was later used in [45] in native code for covert channels, this paper is the first work that uses the cache occupancy channel with a degraded precision timer.

Another JavaScript-based cache attack was introduced in [66] that exploit the conflict of ASLR and caching. ASLR is the one important first line of defense against memory attacks and has an important role in many countermeasures. [66] showed that ASLR in modern cache-based architectures conflicts with caching, what they called AnC. Relying on initial access to memory, implementing AnC in JavaScript is practical. In this attack, an attacker can derandomize the virtual addresses of code and victim data by tracking the cache lines that store the inputs of page table for address translation within itself.

Page deduplication attack on JavaScript. In 2015, [4] presented the first practical page deduplication attack on JavaScript. By performing this attack, we can determine which programs are running and specify the special activity of the user, like determining the web pages that are now open in the user browser. This attack was eligible for any system supporting page deduplication.
The base of the mentioned attack is similar to the method presented in [67], but this time in JavaScript. The steps of the general native method of memory-deduplication attack are as follows:

(1) Filling a page with the data that an adversary suspects to find in the memory of victim's machine
(2) Waiting for deduplicating similar pages with the hypervisor of operating system
(3) Trying to write to the page again and measuring the time
(4) Detecting whether a copy-on-write page fault has been occur or not

Reproducing these steps in JavaScript includes the following steps:

(1) *Page filling with the data we expect to find in victim system*: Common browsers (chrome and firefox) run a call to their internal malloc when allocating a large array in JavaScript, so allocating a large array in JavaScript is done like native code.
(2) *Waiting until the operating system or hypervisor deduplicates our array*: As regards that we don't have any assumption of how much does it take to occur page deduplication, the proposed method is to write the same values in a similar location on destination page in a regular frequency.
(3) *Time measurement*: In the last step, it is necessary to measure the time to determine whether page deduplication has been done or not.

The concentration of the previously introduced memory deduplication attack on JavaScript and the similar ones in native code ([3, 16, 20, 48, 67, 68]) was to discover whether a certain page exists in victim system or not. In 2016, [18] showed that the power of the deduplication side channel is much more. This paper showed that memory deduplication could enable an adversary to expose arbitrary data from memory and to read/write arbitrary memory address.

Rowhammer attack on JavaScript. In 2016, [6] presented a pure implementation of rowhammer attack in JavaScript. It was the first hardware fault injection attack on JavaScript. It is plausible to perform both one-sided and double-sided rowhammer attack on JavaScript. The basis of the operation performed in this attack is to use a large array and change bits through this array.

Keystroke attacks on JavaScript. Implementing the idea of keystroke attacks on JavaScript is another interesting field, both for attacks and countermeasures. In 2017, [33] presented the first general keystroke attack on JavaScript that target other browser tabs, processes, and programs. URL and user classification, and also detecting touch screen interactions are feasible by using this attack. The basis of this attack is the possibility of keystroke detection through side channel, according to corresponding risen interrupt. This idea had been implemented formerly with native code. If a process falls into an interrupt, a significant difference will be observed in its execution time. This difference is attributed to the interrupt management by the operating system. Specially I/O interrupts (like our investigating interrupt: keystroke interrupt) cause peak observation in time tracking. The mentioned attack consists of two phases:

(1) *Online phase*: In this phase, the interrupt timing

attack on JavaScript (that explained above) will be run.

(2) *Offline phase*: The collected measurements will be processed and analyzed in this phase. During this time, an attacker can perform thousands of tracks to extract victim typing behavior. The approach of K-NN is used in the classification of this work.

Timing attack on shared event loops. Event-driven programming is a common pattern for GUIs, web clients, and server-side and network programming. Central components of event-driven programming are event loops. [12] in 2017, showed that shared-event loops are vulnerable to side-channel attacks. The method of attack is that an adversary process monitors the pattern of other processes' loops usage by entering the events to the queue and measuring their dispatch time.

GPU-based attack On JavaScript. The graphics processing unit (GPU), scaling the number of processor cores and memories, is an integral part of today's computers that is employed to accelerate a variety of applications such as image rendering [35]. Nevertheless, in 2018, [35] showed that GPU could be used to speed up microarchitectural attacks. The leaked information gives us the possibility of performing timing side-channel attacks are possible.

JavaScript template attack. In the following, in 2019, [32] presented the first fully-automated approach to track user behavior based on browser properties. For this purpose, a template for each environment must be built.

Spectre attack on JavaScript. In 2019, in the same paper that Specter attack was first published, the JavaScript-based implementation of Specter was also presented [7]. Like a number of other JavaScript-implemented timing side-channel attacks, this attack uses a large array for the eviction process and SharedArrayBuffer for time measuring.

Table 1 shows the timing side-channel attacks implemented in JavaScript and WebAssembly over time.

In the following, two main building blocks and challenges of these attacks, time measuring and eviction, will be explained in Section 5 and Section 6.

5 Timers in JavaScript and WebAssembly

To solve the challenge of timing in this platform, the first idea was to use Performance.now() function [4, 5]. After that, browsers reduced the precision of this function; thus, various researches have been carried out to introduce some alternative implicit timers [9, 11, 69]. These timers use several web features to measure time without having the primary timer function. They can be considered in either blocking or free-running mode. The first category runs independently of the rest of the code in parallel, and the second category, if implemented, blocks the rest of the code. As the most precious implicit timers, free-running message passing, using SharedArrayBuffer, and using WebAssembly memory can be mentioned.

Message passing is a JavaScript mechanism based on setting message listeners up for one script. This method of time measuring uses a counter thread with postMessage function. The precision of this method in the free-running mode is 15 ms. The method of using SharedArrayBuffer is an extension of message passing by web workers. SharedArrayBuffer is the JavaScript object that presents a fixed-length data buffer, and its ownership can be shared between multiple web workers. In this method, a worker increases the counter value, and the main worker is responsible for reading the value of this counter as a timer. Note that the precision of this method in 2 ns. The last method is based on WebAssembly memory in the shared mode and has a similar function and precision to the previous method. A comprehensive review of represented timers is shown in the Table 2.

In addition to timing methods, some general techniques were presented in [9, 11]. These techniques are the basis for using these methods. The idea of one of these techniques, named Clock Interpolation, is to fine-tune the unit of measurement of an inaccurate timer (major clock) by a more accurate timer (minor clock). This process is done in two phases: Learning and Timing. Another technique, named Edge Thresholding, is used when there is no need for accurate time stamps and it is sufficient to distinguish between the time of a function with short running time (f_{fast}) and a function with long running time (f_{slow}). Figure 5 and Figure 6 illustrate these two techniques.

In the following, we will introduce our timer and show that this timer and SharedArrayBuffer are the most powerful and accurate timing solutions for executing side-channel attacks, and their accuracy is equal to each other.

Table 1. Timing Side-Channel Attacks on JavaScript and WebAssembly

Attack	Feature Used							
	Cache	Memory dedup.	DRAM	I/O interrupts	JS event loop	GPU	Web features	Spec. execution
Cache Attacks [6, 34]	✓							
Page Deduplication Attacks [18, 40]		✓	✓					
Rowhammer Attack [6]			✓					
Keystroke Attack [33]				✓				
Event Loop Based Attack in Event-Driven Programming [12]					✓			
GPU-Based Attack [35]						✓		
Fully Automated Attack [32]							✓	
Spectre Attack [7]	✓							✓

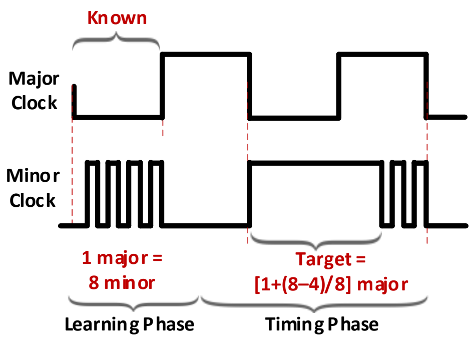

Figure 5. Clock-interpolation technique

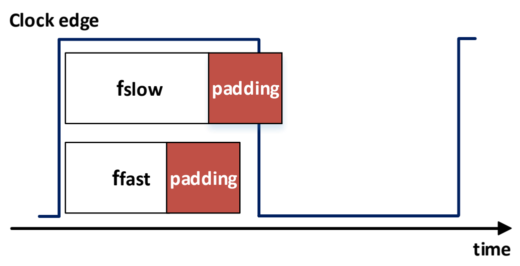

Figure 6. Edge-Thresholding technique

5.1 Eagle Timer: A New Precious Timer Based on WebAssembly Memory

The timing method presented in this paper is based on WebAssembly memory, named Eagle timer. WebAssembly memory can be shared by setting its shared flag. Eagle timer used WebAssembly memory is shared, and a web worker increments the counter uniformly, and at the same time, the main worker reads the value of this counter and uses it as a timer. Figure 7 shows the mechanism used in this timing method.

Table 2. A review of timers introduced in JavaScript and WebAssembly

Timer	Type		Feature Used
	Explicit	Implicit	
Performance.now() [5]	✓		A JS native API
clock-gettime() [34]	✓		A system API
TIME_ELAPSED_EXT [35]	✓		An OpenGL extension
TIMESTAMP_EXT [35]	✓		An OpenGL extension
Video Parsing [69]		✓	Parsing a document
ApplicationCache [69]		✓	File size
Service Worker [69]		✓	A web feature
Script Parsing [69]		✓	Script src attribute
Timeouts [9]		✓	A JS function
Message Passing [9]		✓	A JS API
Message Channel [9]		✓	A JS API
CSS Animation [9, 11]		✓	A CSS ability
SharedArrayBufer [9, 11]		✓	A JS API
Video Frame Data [11]		✓	A JS ability
WebSpeech API [11]		✓	A JS API
WebVTT [9, 11]		✓	A JS API
A Rate Limited Download [11]		✓	Download rate
Cooperating Iframes/Popups from Same Origin [11]		✓	Iframes/Popups
clientWaitSync() [35]		✓	A WebGL2's function
getSyncParameter() [35]		✓	A WebGL2's function

The accuracy of this timing method is the same as that of using SharedArrayBuffer. Despite the differ-

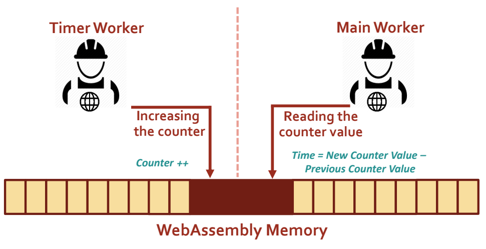
Timer Worker
Main Worker
Increasing the counter
Reading the counter value
Counter ++
Time = New Counter Value – Previous Counter Value
WebAssembly Memory

Figure 7. WebAssembly memory based timer

ences between WebAssembly memory and JavaScript memory, it works similarly to SharedArrayBuffer when shared. The timer provided is a method parallel to the method of using SharedArrayBuffer but in the WebAssembly platform. This is why it can be claimed that these two timers with an accuracy of 2 ns are the most accurate timers that have been used so far for timing side-channel attacks on JavaScript and WebAssembly.

With the introduction of WebAssembly, after several years, Google Chrome browser in its version 70, WebAssembly support for multi-threading capability was experimentally tested and exposed to the review and report of developers. Despite the explanations provided, there is no complete and reliable document on the details of how multi-threading works on WebAssembly, and even contradictory contents are published about it. For this reason, in order to understand how this feature works in WebAssembly, in order to implement a timer based on multi-threading capability in WebAssembly, it was necessary to examine the multi-threading capability in our work. What should be briefly mentioned here is that it is not currently possible to use this feature for timing. Further details will be provided in Section 5.2.

5.2 Results

In this section, some results obtained from various studies on timing methods in this paper are presented. First, the precision of the most precious timers and the status of different browsers' support for these timers are provided. Next, the possibility of measuring time using the multi-thread feature in WebAssembly will be discussed, and then, the possibility of running the system's APIs in WebAssembly will be examined.

To summarize the results of the investigations on the accuracy of the timers, pay attention to this section. The precision of the most precious timers is provided in Table 3 and the status of different browsers' support for these timers are shown in Table 4.

To evaluate the feasibility timing based on multi-threading capability in WebAssembly, we performed various studies. After investigations and studies of various documents about this feature, as well as several implementations of timing, based on multi-threading capability in the WebAssembly, the result is that this feature is now in the form of an initial version with full multi-threading capability available. This means that one cannot expect accurate time measurements by two threads running in parallel. However, the evidence suggests that full multi-threading capability is likely to be introduced in future versions, and if that happens, given the risks ahead, the timer based on it will be one of the most accurate timers available that can be used for timing side-channel attacks.

We will now look at the possibility of executing system's APIs in WebAssembly. In 2018, the article [34] claimed that there is no access to the system's APIs on WebAssembly. For this reason, system timing instructions such as clock_gettime cannot be accessed on this platform. However, our research shows the opposite. In this way, in WebAssembly, such instructions can be accessed without any restrictions. However, it is obvious that the accuracy of these instructions, due to browsers' policies, is reduced to the timers' accuracy in JavaScript. Thus, although the accuracy obtained from these instructions is not such that they can be used to implement timing side-channel attacks on JavaScript and WebAssembly, the claim made in the paper [34] will be rejected with the experiments performed in our work.

6 Eviction Strategies in JavaScript and WebAssembly

Cache Eviction is the process of filling memory with new data to evict the data that we expect to flush from the cache [9]. We face the challenge of cache eviction in JavaScript because we do not have access to processor ISA like clflush, and we have no notion of pointers.

The L1 cache specifies the set assignment of the lower bits of its virtual address [1]. Assuming that the attacker knows his own variables' virtual addresses, [5] in 2015, it is straightforward to create cache set in the L1 cache. However, it was a problem that the set assignment for variables in the LLC is done by reference to their physical addresses that many processes do not have access to them. [2] in the same year, proposed to assume that the system is running in large page mode that 21 lower bits of physical and virtual addresses are similar and it is possible to decide higher bits by using an iterative algorithm. However, it still was not a comprehensive solution because it does not work on 4K page mode that only 12 bits of physical and virtual addresses are similar.

The mapping method of physical addresses to cache set indices was discovered by [66], in 2013. According

Timer	Message Passing	Message Channel	SharedArrayBuffer Beased	WebAssembly Memory Based
Precision	15 μs	30 μs	2 ns	2 ns

Table 3. The Precision of Different Timers

PC						Mobile						Execution Environment
Safari	**Opera**	**Internet Explorer**	**Firefox**	**Edge**	**Chrome**	**Samsung Internet**	**Safari on iOS**	**Opera for Android**	**Firefox for Android**	**Chrome for Android**	**Android webview**	**Node.js**
Enabled	47 Enabled	10 Enabled	Enabled	12 Enabled	Enabled	Enabled	Enabled	44 Enabled	10	Enabled	Enabled	Enabled
Message Channel												
5 Enabled	10.6 Enabled	10 Enabled	41 Enabled	Enabled	4 Enabled	1.0 Enabled	5.1 Enabled	11 Enabled	41 Enabled	18 Enabled	4.4 Enabled	Enabled
SharedArrayBuffer												
10.1-11 Disabled	Not Supported	Not Supported	57 Need to be configed	16-17 Disabled	68 Enabled	Not Supported	10.3-11 Disabled	Not Supported	57 Need to be configed	60-63 Disabled	60-63 Disabled	8.10.0 Enabled
WebAssembly.Memory												
11 Enabled	44 Enabled	Not Supported	52 Enabled	16 Enabled	57 Enabled	7.0 Enabled	11 Enabled	43 Enabled	52 Enabled	57 Enabled	57 Enabled	8.0.0 Enabled

Table 4. Final conclusion on supporting state of different browsers from the most precious timers for timing side-channel attacks on JavaScript and WebAssembly

to this research, by accessing to 8MB contiguous of physical memory, we can invalidate all of cache sets. To localize it in a JavaScript environment, [5] proposed to an 8MB array in virtual memory. Also, they present an iterative algorithm to identify each set of the cache.

After that, [6] in 2016, showed that it is possible to trigger hardware faults by performing fast cache eviction on all architectures. It was the first comprehensive probe on cache eviction strategies. A cache eviction strategy accesses to the addresses of an eviction set in a special access pattern. The introduced method of this paper consists of two phases:

(1) *Offline Phase:* In this phase, the attacker finds the most consistent eviction strategy for his accessing machines, between many strategies for different platforms, considering the criteria eviction rate, execution time, hit rate, and miss rate.
(2) *Online Phase:* Attacker targets an unknown system.

In windows, physical pages are handed out based on cache sets. The addresses that are 128KB apart usually map into the same cache set. [18] in 2016, used this feature to find the cache eviction sets for memory locations that we expect to hammer.

7 Countermeasures

In this section, we review presented countermeasures for timing side-channel attacks on JavaScript and WebAssembly. These countermeasures can be generally categorized into three levels of hardware, operating system, and software. These three levels of attacks are described in Section 7.1 to Section 7.3. After explaining and evaluating the previous methods in section Section 7.4, we present our detection-based method for these attacks, called Lurking Eyes in section Section 7.5.

7.1 Hardware Level Countermeasures

These methods are mainly concerned with the lowest level of abstraction used to confront timing side-channel attacks on JavaScript and WebAssembly. These countermeasures try to changes the structure and architecture of processors. These methods have more cost and overhead than the other two-level methods, but in many cases, if implemented, they have a high ability to counter attacks [5–8]. In the following, we will introduce these methods:

1. Changes to the way physical memory addresses are mapped to cache lines. To confound the attacks like what was mentioned in [5], that are based on direct use 6 of the lower 12 address bits to select a set of cache, as an impressive countermeasure, we can change the way that physical memory addresses are mapped to cache lines [5].

2. Move to an exclusive cache microarchitecture. It will be infeasible to evict entries from the L1 cache when we move to an exclusive cache [5]. Therefore, using exclusive Caches will be more difficult to implement cache attacks on JavaScript and WebAssembly.

3. Increasing the refresh rate. Increasing the memory refresh rate is a solution for Rowhammer attacks. However, it is not in a good status from the point of view of performance, also is inadequate to protect against attacks on all modules of DRAM [6, 62].

4. TRR and pTRR. The process of refreshing neighboring rows when a row get accessed more than a threshold number (what the features Pseudo Target Row Refresh (pTRR) and Target Row Refresh (TRR) do) is a countermeasure with good condition in terms of overhead for Rowhammer attacks [6]. Note that this is a good overhead compared to other hardware methods. Otherwise, the overhead of hardware-level methods is generally higher than other level methods.

5. Using ECC memory. Using Error-correcting code memory (ECC memory) is another way (though unreliable) to protect against Rowhammer attacks [6]. ECC memory is a type of memory that can detect and correct n-bit data corruption in memory.

6. Isolated cache. It means to separate caching PT (Page Table) entries from data cache relieve AnC (As mentioned earlier, AnC is the title for conflict of ASLR in modern cache-based architectures with caching). It is very expensive and in conflict with the goal of ASLR to provide an inexpensive first line of defense [8].

7. Time warp. Time Warp idea is originated from [70] and was mentioned in [8] as a countermeasure for timing side-channel attacks. It reduces the timers' accuracy, and distinguishing between different microarchitectural events becomes hard. An algorithm that schedule CPU instructions indifferent to timing differences from hardware components protects us from timing cache attacks as a result [71].

8. Halting wpeculative execution. If the speculative execution gets halted, the vulnerability of the conditional branch will be avoided. This method is applicable to Spectre attack, but there are some serious problems, for example a method that works for all processors or system configurations, and it greatly reduces the performance level [7].

7.2 Operating System Level Countermeasures

These countermeasures are related to the operating system and have a higher level of abstraction and less performance overhead than hardware-level countermeasures [4, 8–10]. In the following, introduce these methods will be explained:

1. Disabling memory deduplication. The only general effective way to prevent memory deduplication attacks is Disabling memory deduplication. Other ways somehow hurt the efficiency or are not practically implementable [4]. It should be noted that because of memory deduplication applications, disabling it can hurt performance, so this countermeasure is not taken seriously by developers and private clouds.

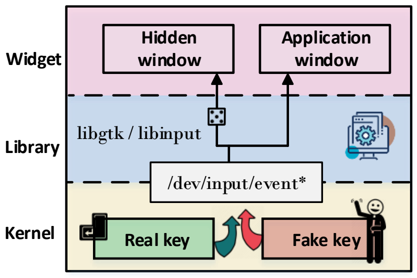

Figure 8. KeyDrown countermeasure layers

2. Cache coloring. To quarantine an application from the rest of the system, we can partition the LLC. This countermeasure imposes performance limitations on both operating system and application [8].

3. KeyDrown. KeyDrown is a 3-layer defense mechanism against keystroke timing attacks. The mechanism is represented by [72] and is implemented on X86 and ARM by responding to three requirements and a very low performance overhead:

(1) *Minimizing the accuracy of side-channel:* The F-Score (a measure of a test's accuracy) has to be reduced to the amount that the adversary does not benefit from side-channel.
(2) *Reducing statistical features in password input:* The adversary must need a huge and impractical number of tracks to access the F-Score of current attacks.
(3) *Implementation security:* The countermeasure must not have an identifiable code path or data access pattern to assure not to leak the information.

As shown in Figure 8 we are faced with three layers. The first layer job is injecting fake interrupt. Each real key interrupt gets combined with many fake interrupts. As a result, the real key will be unidentifiable. This makes the keystroke interrupt density uniform and independent of the real key. The first layer manages two types of interrupts: Hardware interrupts from input devices and Input interrupts. The second layer protect the input library of the user against Flush+Reload attacks. For every keystroke event received from the kernel, a random keystroke will be sent to a hidden window. In the third layer, the real password input field will be protected against Prime+Probe attacks by accessing the base buffer when a real or fake key arrives.

4. Thread affinity to the same CPU core. It is a countermeasure to prevent parallelism and asynchronous time measurement innovations, and it means to have thread affinity, for threads with shared memory, to the identical core of CPU. Note that this method has a severe destructive effect on performance, also by using it, the web workers application will become fruitless [9].

5. Lessening the level of aggressiveness of OS allocator in near-out-of memory. This countermeasure has been represented for Rowhammer attacks. For optimization, the large physical memory frames being allocated by the operating system. The same frame is not allocated for page tables, kernel pages, and user pages, except in near-out-of memory (the last few kilobytes of physical memory). If the level of aggressiveness of OS allocator in near-out-of memory become less, performing double-sided hammering will become infeasible. However, it is more difficult to prevent single-sided hammering, as, among the borders of regions, the hammering is feasible [9].

7.3 Software Level Countermeasures

This level of countermeasures is the highest level of abstraction. These methods can be divided into two categories: Browser and JavaScript architecture related [12, 13] and Timer related [5, 9, 11, 12].

7.3.1 Browser and JavaScript Architecture Related Methods

What we will study in this part are the countermeasures that are dependent on the features of browser and JavaScript architecture.

1. Full isolation. We should look for providing cross-process navigations, cross-process interactions of JavaScript and out-of-process iframes with low overhead to satisfy the effort of multi-process architecture of Chrome to use a different renderer for cross origins. However, there is an unanswered question: how to handle the process limit of the system [12]?

2. CPU throttling. It is a method that Chrome v55 represented in the form of an API to throttle the functions of a background page when they exceed a predefined threshold that determines how much CPU a background page is permitted to use [12]. Although affecting background tabs that wants to spy on the main thread of renderer, it cannot prevent spying on iframes, popups, I/O thread of the host process among shared workers, and background tabs with the audio task. Also the time that pages will be throttled after that (10 seconds) is too long to prevent the attacks mentioned in [12].

3. JavaScript Zero. One of the most important software level countermeasures is JavaScript Zero that was presented in [13]. This method is, in fact, a generic fine-grained permission model that can operate at several levels of rigidity, including off, low, medium, high, and paranoid. Each of these levels subject to the permission of user, restrict, or disable some features of JavaScript. These features are: memory addresses, accurate timing, multithreading, shared data, and sensor API. The design of JavaScript Zero is based on an abstract layer between the JavaScript engine and the provided interface for JavaScript developers. The function of this layer is to protect functions, interfaces, and object features. The middle layer in the face of each transaction can perform four different actions: block, allow, modify, or allow after user permission. JavaScript Zero seems to be the most appropriate way to deal with timing side-channel attacks on JavaScript until now. Table 5 shows the policies correspond to the protection levels of JavaScript Zero for Google chrome browser.

7.3.2 Timer Related Methods

One of the most fundamental topics in implementing timing side-channel attacks of JavaScript is representing alternative timers instead of a native timer, therefore dealing with these timers to prevent such attacks, is one of the most interesting types of countermeasures against timing side-channel attacks on JavaScript and WeAssemly. In this part, we will have a glance at the timer related countermeasures.

1. Access limitation to the timers. One of the basic initial reactions to the timer primitives in JavaScript is restricting access to these timers. We can achieve this in two ways: Catching User Permission or Validation By a Third-party (for instance downloading from authentic stores) [5].

2. Heuristic profiling. It is another approach to detect and prevent the investigated attacks. This idea has come from [73] and is adapted to timing side-channel attacks by [5]. It means the detection of the profiling-like behavior from executing code and modifying its response by JavaScript runtime. It is effective because the attack's measurements access memory in an especial template, and modern JavaScript runtimes split a hair the runtime performance of code.

3. Reducing the explicit timer's resolution. After publishing the paper [5] and introducing the first cache attack on the JavaScript platform, browser vendors lowered the resolution of performance.now() [8, 9, 12]. Although this reaction had an immediate effect on performing such attacks, it leads to many other timer primitives to overcome this limitation. Somehow we can consider the timers reviewed in this paper as a reaction to this limitation.

4. Fuzzy time. The concept of Fuzzy Time was brought up in [11] as an idea like [74] that can be

Table 5. The policies correspond to the protection levels of JavaScript Zero in Google chrome browser [13]

Prevention Level / Feature	Memory addresses	Accurate Timing	Multithreading	Shared data	Sensor API
Off	No Action	No Action	No Action	No Action	No Action
Low	Buffer ASLR	Ask from user	No Action	No Action	No Action
Medium	Array preloading	Timestamp with low resolution	Delay on message	Slow SharedArrayBuffer	Ask from user
High	Non-deterministic array	Fuzzy time	Web worker polyfill	Disable	Fixed value
Paranoid	Array index randomization	Disable	Disable	Disable	Disable

adapted to the browsers to make them trusted, to reduce the clocks resolution and the bandwidth of timing channels. As a proof of possibility, the Fuzzyfox (a Firefox with the implemented fuzzy time idea on it) was introduced in this paper. They claimed that this concept could mitigate all clocks.

5. Fermata is what they described it as a theoretical design for browsers to reduce the resolution of all timers. The accuracy of all clocks in Fuzzyfox should be lowered to provide a resolution no less than a predefined threshold. In the ideal case, the effectiveness of this idea is significantly obvious [11].

6. Increasing the latency of message passing. It is a compromise to overcome the problems of sThread Affinity countermeasure. It should not have a destructive effect on applications with low- to moderate-bandwidth [9].

7. Rate limiting. [12] introduced this approach for one of its counters. It means to impose a threshold on the rate of posting tasks into an event loop. Because of the performance overhead that this approach imposed, we will have a problem with asynchronous code.

A comprehensive review of the countermeasures, their short description, and considerations are shown in the Table 6.

7.4 A General Evaluation of Countermeasures Up to Now

As mentioned, previous countermeasures can be categorized into three levels of hardware, operating system, and software. Among these three levels, the hardware level methods will generally be very costly to implement because they are usually associated with a change in architecture and structure. However, if these methods can be implemented in practice, they will have a more significant impact than two other methods. In general, these methods are not reasonable in many cases because of the high cost.

The methods introduced at the operating system level are also usually associated with structural changes. Also, some methods, like disabling memory deduplication, will make a negative impact on the system.

The countermeasures of software level are the most innovative for researchers and developers because these methods cost far less than the other two levels methods, and also they have relatively high flexibility. Because of the mentioned features, this level of confrontation with timing side-channel attacks can attract more focus than other levels. As mentioned before, JavaScript Zero, as a software method, can be introduced as the most appropriate countermeasure for timing side-channel attacks on JavaScript until now. Despite its considerable advantages that were mentioned in section 8.3, this method has several drawbacks:

(1) To counter timing side-channel attacks, JavaScript Zero focuses on restricting some JavaScript features. Note that the presence of these features is not a strong reason for timing side-channel attacks. To ensure that an attack occurs, several features must be examined together.
(2) Restricting and disabling JavaScript features deprives the user of the maximum performance offered on a web page. In many cases, such features are not the reason for a timing side-channel attack.
(3) In some cases, to prevent an attack, there is a need to ask the user. This solution does not provide good feedback on user satisfaction, especially for regular users.

To overcome these challenges, we have implemented a detection-based method in which there is no need to disable useful JavaScript features. In the following, we will explain this method in detail.

7.5 Lurking Eyes: a New Detection Based Method

One of the most common methods to deal with malicious attacks is detection-based methods. To prevent side-channel attacks at the architecture level, several detection-based methods have been proposed so far [75–78]. There are also several detection-based

Table 6. A comprehensive review of the countermeasures

Countermeasure	Level				Focused Features							
	H.W.	OS	S.W.		Memory arch.	Cache arch.	Timers' accuracy	Spec. execution	Mem. dedup.	OS allocator	Statistical features	Browser Policy
			Browser & JS Related	Timer Related								
Changes to The Way Physical Memory Addresses Are Mapped to Cache Lines [6]	✓				✓							
Move to an Exclusive Cache Micro-Architecture [5]	✓					✓						
Increasing The Refresh Rate [6, 62]	✓				✓							
TRR and pTRR [6]	✓				✓							
Using ECC Memory [6]	✓				✓							
Isolated Cache [8]	✓					✓						
Time Warp [8, 70, 71]	✓						✓					
Halting Speculative Execution [7]	✓							✓				
Disabling Memory Deduplication [4]		✓							✓			
Cache Coloring [8]		✓				✓						
KeyDrown [10]		✓					✓				✓	
Thread Affinity to The Same CPU Core [9]		✓										✓
Lessening the level of aggressiveness of OS allocator in near-out-of memory [9]		✓								✓		
Full Isolation [12]			✓									✓
CPU Throttling [12]			✓									✓
Javascript Zero [13]			✓									✓
Access Limitation to The Timers [5]				✓			✓					
Heuristic Profiling [5, 73]				✓			✓					
Reducing the Explicit Timer Resolution [5, 9, 12]				✓			✓					
Fuzzy Time [11, 74]				✓			✓					
Increasing The Latency of Message Passing [9]				✓			✓					
Rate Limiting [12]				✓			✓					

methods to deal with malicious JavaScript codes [79–83]. However, no detection-based methods for timing side-channel attacks on JavaScript and WebAssembly have been provided so far. The approach presented in this paper is a detection-based method, called Lurking Eyes, that is capable of detecting timing side-channel attacks on the JavaScript and WebAssembly platform within different web pages. As previously mentioned, in our method, there is no need to disable useful JavaScript features. In the following, we will explain more about how this method works.

7.5.1 Lurking Eyes Overview

As discussed in the evaluation of JavaScript Zero, the existence of certain features in JavaScript is not alone a strong reason for the occurrence of timing side-channel attacks on the web page being investigated. However, to ensure that an attack occurs, several features need to be seen together. This is one of the major disadvantages of JavaScript Zero because it restricts some of the functionality available in JavaScript. As a result, the user will be deprived of the optimal performance of a web page. Accordingly, Lurking Eyes considers several features together and identifies attacks based on them. The steps to investigate the occurrence of a timing side-channel attack on a web page are as follows:

(1) Investigating HTML code of the indented web page
(2) Extracting the JavaScript code used on the web page
(3) Searching for the features that may indicate the occurrence of a timing side-channel attack
(4) Announcing the result to the user

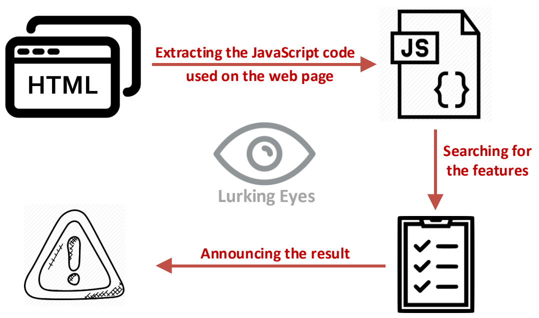
HTML
Extracting the JavaScript code used on the web page
JS
Searching for the features
Lurking Eyes
Announcing the result

Figure 9. The process of investigating the occurrence of a timing side-channel attack on a web page

This process is shown in Figure 9.

In this method, the JavaScript codes of the intended web page can be extracted, whether internal JavaScript codes or external ones. Our countermeasure can be implemented in a variety of languages. Also, we can implement this method on a browser extension for better efficiency. What we have used as a practical example to evaluate this method is C# language with a graphical environment for better use.

The attacks detectable in this method are memory deduplication and Spectre attacks. These attacks can be detected in the three most important timing methods: message passing in free-running mode, using SharedArrayBuffer, and using WebAssembly memory. These timers are thought to be most likely to use in attacks. The probability of using other timers is very low because the precision of other timers for use in these attacks is ineffective.

The method of detecting malicious code for each of these timers investigates the presence or absence of some features and functions required to execute a side-channel attack using that timer. The features and functions are investigated in the JavaScript code of the intended web page. Given the importance of each feature and function investigated in the intended attack, it is assigned a score. If the total score assigned to the features in a page's JavaScript code exceeds a predetermined threshold, that page is declared a malicious page. We obtain these features and functions by evaluating the attacks and replicating them in different ways. By examining the acceptable number of attacks, we ensured that using these features for attacks is unavoidable in these attacks. We set the threshold for each timing attack by investigating the main properties of timing attacks in a large number of web pages. With these thresholds, we can increase the accuracy of our detection method. Investigated features in our method consist of the below list. Different features in this list are related to different timers.

(1) Using a large array
(2) Allocating and flushing to/from array
(3) Time measuring
(4) Message passing using PostMessage()
(5) Using web workers
(6) Using message listeners
(7) Delay instructions
(8) Defining SharedArrayBuffer
(9) Defining WebAssembly.Memory
(10) Iterating the operation

The attack alert is displayed to the user on several levels due to the variety of features. Warnings to a certain level are not a reason for malicious code, but if they exceed a predetermined threshold, then the web page is considered malicious. There are several levels of certainty for introducing a page as a malicious page. In total, we propose Seven levels of alert for this method:

(1) Benign code and no timing side-channel attack
(2) Very very low probability of malicious code
(3) Very low probability of malicious code
(4) Low probability of malicious code
(5) Possibility of malicious code
(6) High probability of malicious code
(7) Definite probability of malicious code

The attacker can deceive the application in various ways due to the nature of detection-based methods that require the investigation of code on the web page. We have tried to identify as many different methods of deception as possible and to provide them with a solution. Examples of these methods of deception can be the use of upper and lowercase letters, the use of different commands that are equivalent, and the extra spacing between words.

7.5.2 Evaluation

In this section, we evaluate the detection method presented in this article, using different evaluation metrics. Since no such approach has been proposed so far to deal with side-channel attacks on JavaScript and WebAssembly platforms, and this paper presents the first detection-based approach to these attacks, There is no prior relevant work to compare. We provide a report on the evaluation metrics for this countermeasure.

Before reporting the result of our work, it is worth mentioning that if a page without a malicious code is correctly detected as a secure page, it is considered a true negative (TN) and otherwise a false positive (FP). Also, if the malicious page is correctly identified, it is a true positive (TP) and otherwise is called false negative (FP). F-measure is also one of the most essential and standard metrics used to eval-

Table 7. Statistics of pages reviewed by Method Lurking Eyes

Number of Web Pages Reviewed	3350
Number of Benign Web Pages	3230
Number of Malicious Web Pages	120
Number of True Negatives	3226
Number of True Positives	120
Number of False Negatives	0
Number of False Positives	4

Table 8. Evaluation metrics of Lurking Eyes

Evaluation Metric	Equivalent Expression	Value
False Positive Rate	*FP/(FP+TN)*	0.001
False Negative Rate	*FN/(FN+TP)*	0
Accuracy	*(TP+TN)/(TP+TN+FP+FN)*	0.998
Error Rate	*(FP+FN)/(TP+TN+FP+FN)*	0.001
Specificity	*TN/(TN+FP)*	0.998
Precision	*TP/(TP+FP)*	0.967
Recall	*TP/(TP+FN)*	1
F-measure	*2/[(1/Precision)+(1/Recall)]*	0.983

uate detection-based methods. F-measure is obtained from two other metrics: Precision and Recall. The formula for the metrics evaluated is available in Table 8.

The total number of pages examined in this project is 3350 pages. These pages are mainly selected from the top websites presented in Alexa and Majestic and have tried to select websites from different categories. The method of choice in Majestic is in order, and in Alexa, thematic. Of the total pages reviewed, there are 3230 benign and 120 malicious pages. Of the 3230 malicious pages, 3226 pages were correctly, and 4 pages were incorrectly classified. Also, 120 malicious pages (97 pages with memory deduplication attack code and 23 pages with Spectre attack code) were correctly identified as malicious pages. A summary of the presented statistics is shown in Table 7. A comprehensive evaluation of the proposed method, using seven different metrics, is shown in Table 8. This method gives acceptable results with an accuracy of 0.998, precision of 0.983, recall of 1, and F-measure of 0.983.

Note that our detection-based method can be improved to examine JavaScript events dynamically (online). Moreover, the attacker may recognize our detection-based method in the target system and use tricks to bypass this approach. Therefore, to overcome this challenge, machine learning based approaches can be applied as future work.

8 Conclusion and Future Works

In this paper, we explained side-channel attacks, especially timing side-channel attacks. We then looked at how the challenges of these attacks can be addressed by implementing them on JavaScript and WebAssembly. In the following, we had an overview of the side-channel attacks implemented on JavaScript and WebAssembly. The challenges of implementing these attacks on JavaScript were also discussed in two general sections: time measuring and eviction strategies. In the timers section, in addition to introducing and evaluating previous works, we presented a timer based on WebAssembly memory, named Eagle timer.

In the following, we discussed the countermeasures for timing side-channel attacks on JavaScript and WeAssembly. These countermeasures were introduced at three levels of hardware, operating system, and software. After that, we introduced and evaluated our method, named Lurking Eyes. This method gives acceptable results with an accuracy of 0.998, precision of 0.983, recall of 1, and F-measure of 0.983.

As future works, to improve the detection method presented in this study, we can supplement this method by using machine learning methods or by considering JavaScript events running on a web page, reducing the probability of error. Also, about timing methods, if the full multi-threading capability version of WebAssembly is introduced in the future, it may be possible to present an accurate timer based on it.

References

[1] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *RSA*, pages 1–20. Springer, 2006.

[2] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE*, pages 605–622. IEEE, 2015.

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

[4] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *ESRCS*, pages 108–122. Springer, 2015.

[5] 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 *CCS*, pages 1406–1418, 2015.

[6] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *DIMVA*,

pages 300–321. Springer, 2016.

[7] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE*, pages 1–19. IEEE, 2019.

[8] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[9] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *FCDS*, pages 247–267. Springer, 2017.

[10] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. 2018.

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

[12] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX*, pages 849–864, 2017.

[13] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *NDSS*, 2018.

[14] Mohammad Erfan Mazaheri, Farhad Taheri, and Siavash Bayat Sarmadi. Lurking eyes: A method to detect side-channel attacks on javascript and webassembly. In *ISCISC*, pages 1–6. IEEE, 2020.

[15] David A Patterson and John L Hennessy. *Computer organization and design MIPS edition: the hardware/software interface*. Newnes, 2013.

[16] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-vm attacks on xen and vmware are possible! *IACR*, 2014:248, 2014.

[17] Alan Jay Smith. Cache memories. *CSUR*, 14(3):473–530, 1982.

[18] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE*, pages 987–1004. IEEE, 2016.

[19] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *CSUR*, 47(1):11, 2014.

[20] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *DSN*, pages 1–12. IEEE, 2013.

[21] David Flanagan. *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[22] Robin Nixon. *Learning PHP, MySQL, JavaScript, and CSS: A step-by-step guide to creating dynamic websites.* " O'Reilly Media, Inc.", 2012.

[23] jsinfo. js information. https://Javascript.com/info, 2019. Accessed: Aug. 2019.

[24] Mozilla. javascript info. https://developer.mozilla.org/en-US/docs/web/javascript, 2019. Accessed: Aug. 2019.

[25] Mozilla. ArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, 2020. Accessed: Nov. 2020.

[26] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CC*, pages 388–397. Springer, 1999.

[27] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CC*, pages 104–113. Springer, 1996.

[28] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, pages 13–28. Springer, 2002.

[29] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, pages 305–316. ACM, 2012.

[30] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *IEEE*, pages 313–328. IEEE, 2011.

[31] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *CE*, 8(1):1–27, 2018.

[32] Michael Schwarz, Florian Lackner, and Daniel Gruss. Javascript template attacks: Automatically inferring host information for targeted exploits. In *NDSS*, 2019.

[33] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *ESRCS*, pages 191–209. Springer, 2017.

[34] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ICACNS*, pages 83–102. Springer, 2018.

[35] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *IEEE*, pages 195–210. IEEE, 2018.

[36] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website finger-

printing through the cache occupancy channel. In *USENIX*, pages 639–656, 2019.

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

[38] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *SSP*, pages 490–505. IEEE, 2011.

[39] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *DIMVA*, pages 279–299. Springer, 2016.

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

[41] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *TACIS*, pages 667–684. Springer, 2009.

[42] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[43] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the aes. In *RSA*, pages 271–286. Springer, 2007.

[44] Onur Aciiçmez. Yet another microarchitectural attack:: Exploiting i-cache. In *CSA*, pages 11–18. ACM, 2007.

[45] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, pages 199–212. ACM, 2009.

[46] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. {STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud. In *USENIX*, pages 189–204, 2012.

[47] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *SSP*, pages 591–604. IEEE, 2015.

[48] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *RAID*, pages 299–319. Springer, 2014.

[49] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *ADAC*, page 72. ACM, 2016.

[50] Gorka Irazoqui and Xiaofei Guo. Cache side channel attack: Exploitability and countermeasures. *Black Hat Asia*, 2017(3), 2017.

[51] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *DIMVA*, pages 46–64. Springer, 2015.

[52] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *USENIX*, 2017.

[53] Shahid Anwar, Zakira Inayat, Mohamad Fadli Zolkipli, Jasni Mohamad Zain, Abdullah Gani, Nor Badrul Anuar, Muhammad Khurram Khan, and Victor Chang. Cross-vm cache-based side channel attacks and proposed prevention mechanisms: A survey. *NCA*, 93:259–279, 2017.

[54] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *CCS*, pages 353–364. ACM, 2016.

[55] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *USENIX*, pages 565–581, 2016.

[56] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *arXiv*, 2018.

[57] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv*, 2018.

[58] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the security implications of speculative execution in cpus. *arXiv*, 2018.

[59] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *TACT*, pages 37–51. Springer, 1997.

[60] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *CC*, pages 513–525. Springer, 1997.

[61] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In *CHES*, pages 77–88. Springer, 2003.

[62] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *SIGARCH CAN*, volume 42, pages 361–372. IEEE Press, 2014.

[63] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[64] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX*, pages 19–35, 2016.

[65] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *SSTE*, page 5. ACM, 2017.
[66] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.
[67] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *SS*, page 1. ACM, 2011.
[68] Rodney Owens and Weichao Wang. Non-interactive os fingerprinting through memory deduplication technique in virtual machines. In *IPCCC*, pages 1–8. IEEE, 2011.
[69] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, pages 1382–1393, 2015.
[70] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *SIGARCH*, 40(3):118–129, 2012.
[71] Deian Stefan, Pablo Buiras, Edward Z Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESRCS*, pages 718–735. Springer, 2013.
[72] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSSS*. Internet Society, 2018.
[73] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *ESRCS*, pages 200–215. Springer, 2009.
[74] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.
[75] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *DASP*, 2018.
[76] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RID*. Springer, 2016.
[77] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *ASC*, 2016.
[78] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *HASSP*, 2018.
[79] Samuel Ndichu, Seiichi Ozawa, Takeshi Misu, and Kouichirou Okada. A machine learning approach to malicious javascript detection using fixed length vector representation. In *IJCNN*. IEEE, 2018.
[80] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. A machine learning approach to detection of javascript-based attacks using ast features and paragraph vectors. *ASP*, 2019.
[81] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX*, 2009.
[82] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Low-overhead mostly static javascript malware detection. In *USENIX*, 2011.
[83] JM Howe and F Mereani. Detecting cross-site scripting attacks using machine learning. *ISC*, 2018.

Mohammad Erfan Mazaheri received the B.S. degree in computer engineering from Buali Sina University, Hamedan, Iran, in 2018, and M.Sc. degree in computer engineering from Sharif University of Technology (SUT) in 2020. From 2018 to 2020 he was a member of Smart and Secure Systems (3S) Lab at SUT under supervision of Dr. S. Bayat-Sarmadi. His research interests includes system security, hardware security, side channel attacks, and software security.

Siavash Bayat-Sarmadi received the B.Sc. degree from the University of Tehran, Iran, in 2000, the M.Sc. degree from Sharif University of Technology, Tehran, Iran, in 2002, and the Ph.D degree from the University of Waterloo in 2007, all in computer engineering (hardware). He was with Advanced Micro Devices, Inc. for about six years. Since September 2013, he has been a faculty member in the Department of Computer Engineering, Sharif University of Technology, where he is currently a tenured associate professor. His research interests include hardware security and trust, cryptographic computations, and secure, efficient, and dependable computing and architectures. He is a member of the IEEE.

Farhad Taheri received the B.S. degree in computer engineering from Shahid Bahonar University of Kerman (SBUK), Kerman, Iran, in 2016, and M.Sc. degree in computer engineering from Sharif University of Technology (SUT) in 2018. From 2016 to 2018 he is a member of Data Storage, Networks, and Processing (DSN) Lab. at SUT where he researched on reliability of Solid-State Drives (SSDs). Currently he is a Ph.D. student at Smart and Secure Systems (3S) Lab at SUT under supervision of Dr. S. Bayat-Sarmadi. His research interests includes privacy-preserving machine learning, multi-party computation, computer architecture, system security.