Detecting Keystrokes In JavaScript

David Bidner

May 7, 2018

Abstract

Timing attacks and side-channels have been used to bypass security measures and attack users for some years now. It has been shown that these attacks can be used to break cryptographic algorithms and to cause information leaks about a victim system. Keystrokes and interacting with a touch screen triggers interrupts. These interrupts cause timing differences. Within this project, we demonstrate how to measure these differences and use them for side-channel attacks. Typically, these attacks require native code execution, which is less likely to be possible than executing JavaScript on a victim's system. We use the JavaScript engine of modern web browsers to implement the timing attacks without native code. By doing so, we successfully bypass modern security models like the same-origin policy, HTTPS and sandboxing.

We started by implementing a known timing side channel for interrupts in native code and then implemented the same functionality in JavaScript. This allows us to gain information about the victim by simply making him open a website or view a malicious advertisement.

We show that the results of the JavaScript implementation are as accurate as the results from the native code one and that our attack affects personal computers and mobile phones alike. As a countermeasure, we present randomly occurring interrupts to make timings less predictable.

Contents

1 Introduction

JavaScript is a scripting language supported by modern web browsers. It is used to execute code on the user's computer. This code is sandboxed, and it is not possible to execute particular CPU instructions or access the file system. It is well known for years, that JavaScript and timing attacks implemented in JavaScript harm users privacy [6]. More recently, other attacks via JavaScript have been published, e.g., Rowhammer [2] and attacks against address-space layout randomization [1]. Additionally, multiple side channels have been implemented, e.g., a DRAM covert channel by Schwarz et al. [11].

Interrupts change the way programs get scheduled on a system. It may sound like a problem only real-time operating-systems would suffer from but in fact, these timing differences can be measured with simple programs and, in this way, leak information to an attacker, such as keystrokes and network usage.

Keystroke timings have been exploited before to gain knowledge about typed words, passwords, and phrases [5]. Schwarz et al. [12] showed how accurate these timing measurements can be in native code. For this project, we implemented the same functionality in JavaScript. We show that this implementation is accurate enough to deliver the same results as the native code implementation.

For our attack, we implemented our own timestamp functionality to provide a resolution accurate enough to detect interrupts. This was done by increasing a counter over a given timeframe. Depending on the value of the counter after a frame, we can track back the amount of time it was scheduled. This makes it possible to track interrupts, as those lower the amount of time our code was running.

With this simple algorithm, we were able to bypass countermeasures by browser vendors who made JavaScript internal timestamps less accurate. We show that our attack works against every I/O-triggered interrupt and affects users of personal computers and mobile phones alike. Besides being able to spy on users via a website, our attack is also hard to detect as it has a very low impact on the performance of the browser and the memory footprint of the code is very low.

This project is based on previous work which includes JavaScript timestamps and side-channel attacks based on them [3]. Results and conclusions from this project were later used for a paper published by Lipp et al. [13] at the ESORICS conference in 2017.

2 General

Side-channel attacks are based on information leaked by side effects of a computational process. Many sources can be used to measure these effects, e.g., power consumption, timing, or temperature change, of a device. Kelsey et al. [8] showed how little information was needed to break cryptographic cipher implementations. They used time, processor flags, and power consumption as their sources to break various implementations.

While first only being used to break cryptographic implementations, researchers also gathered general information about users using similar techniques. Weinberg et al. [17] showed that it is possible to gather information about a users browser history by using side channels.

Gruss et al. [4] showed how side-channel attacks on caches can be used as a powerful source for information leakage and how to automate attacks on last-level caches.

Side-channel attacks might not only be applied to desktop computers or embedded systems but also be used in cloud environments to exploit shared hosting platforms. Kaniyasu et at. [10] showed how memory deduplication leaks memory content to a second guest system via a timing side-channel.

2.1 Timing Measurement

In native code, it is possible to use the time stamp counter (TSC) of the CPU to measure timings. The TSC returns the number of CPU cycles since the last CPU reset. This instruction allows us to get a very accurate timing measurement. The `RDTSC` instruction is implemented on all x86 CPUs since the first Pentium CPU [7].

In JavaScript, there is no way to get the TSC of the CPU. However, it provides the function `performance.now()`, which returns a floating point number with millisecond precision. In the past, this function even allowed nanosecond precise timestamps, but this behavior changed as it was used for side channel attacks. As this also had an impact on our measurements, we implemented another way to create precise timestamps in JavaScript.

2.2 Timing Attacks in JavaScript

Previous work showed that it is possible to implement attacks using the JavaScript timing interface. Gruss et al. [3] showed that it is possible to detect page deduplication and exploit it to gain information about open websites or running programs. Oren et al. [16] implemented cache attacks in JavaScript. These attacks used the `performance.now()` function to gain sufficiently enough timestamps for their attacks. JavaScript can be used to leak information about a user in many ways. As Mowery et al. [9] showed in their work, JavaScript execution characteristics can be used to identify a user.

As browser vendors introduced countermeasures against these attacks, by making the timestamps less accurate new methods were introduced by attackers to provide highly accurate timestamps again, e.g., by Schwarz et al. [11]. Those methods were also used by us to reintroduce precise timing measurements in JavaScript.

2.3 Interrupt-timing Attacks

A key press on a PS/2 keyboard triggers a hardware interrupt and a second one when the key is released again. Both actions require the kernel to trigger an individual interrupt handler. Those handlers take time to fulfill their actions. With precise timestamps, it is possible to detect if the running process was interrupted by such a handler.

It has already been shown that such interrupt timings can leak information to an attacker in native code [12]. Especially interrupts which take a longer time, such as I/O or network interrupts are easy to detect with accurate timing measurements.

Keystroke timings are well-known for leaking information about a user [15]. Keystrokes may not only change timings inside a program but also affect other side channels such as energy usage, network traffic, or cache behavior.

3 Interrupt-timing Attacks in Native Code

In this section, we describe how timing measurement is implemented accurate enough to detect keypress interrupts in native code. We therefore look at already known measurement algorithms and verify that those are accurate enough for our needs. We implement the exploit described in previous work and show that it works as expected for our use case.

3.1 General Algorithm

The algorithm for measuring timings in native code is based on previous work by Schwarz et al. [12]. We want to adapt and use their C code as a base for our implementation.

To store timings between instruction calls they use the CPU instruction `RTDSC` to get highly precise timestamps. The gap between two consecutive timestamps is then stored. This recording step can later be used to determine a normal difference and a threshold to indicate that an interrupt occurred between the `RTDSC` calls.

When looking at recorded data, peak times can be seen for measurements where an occurred interrupt. These peak values make it possible to define a threshold to distinguish between interrupts.

If the program was not interrupted two subsequent `rtdsc` calls show just a difference of around 27 CPU ticks. For timer interrupts it is higher, around 19 000 ticks. Also, these numbers are slightly lower than for key presses, which were in the area of 80 000 ticks on the hardware we used.

For further investigation, we now can look at the reported events. As mentioned in Chapter 2, a keystroke should be indicated by two events following each other, namely the pressing and the releasing of the key. Other interrupts may not have the second event or need a longer or shorter time and can be filtered in post-processing steps.

3.2 Improving the Accuracy of the Attack

On a typical GNU/Linux system, interrupts are handled on different CPU cores. This may even switch at runtime of the system, depending on the system load and configuration. As this could affect our measurements, we need to enforce the interrupt handling to be always handled by the same core. This can be done via the `/proc/irq/<INPUTIRQNR>/smp_affinity` file. Where the number it contains describes the CPU core which handles the interrupt. After assuring that the interrupts are always handled by the same core, we also need to make sure that the application measuring the timings runs on the same core. This can be done using the `taskset` command. As attackers usually cannot change these parameters, they can instead run multi-threaded to increase the chance to hit the same core as the interrupt handler. On the mobile devices we tested, the screen touch interrupt was always handled by CPU core 1. Hence, the simplified attack works out of the box.

A changing system load or different activities on the system may also affect the timings and the threshold, therefore, an attacker may have multiple profiles for systems and gain some more knowledge about it before applying post-processing steps on the gathered data.

3.3 Measurement Results

These measurements were made on a Laptop with a PS/2 hardware keyboard running on an Intel i7-2620M CPU.

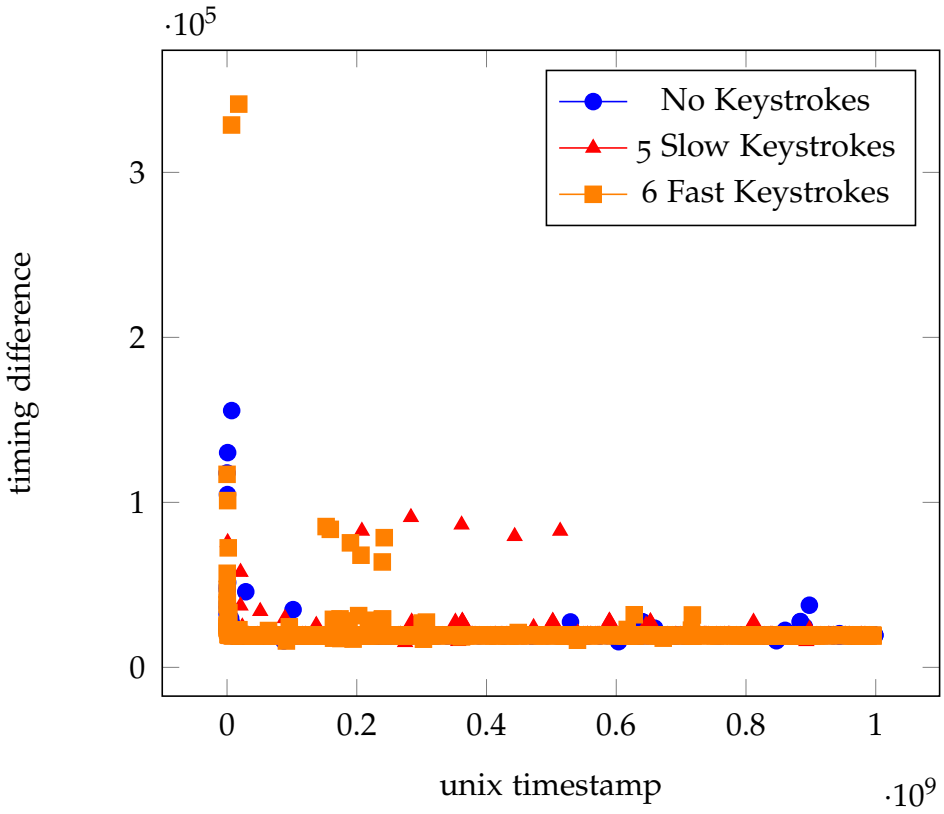

Figure 3.1: Native timing measurement for different inputs

Surprisingly, only one event can be seen in the plot and not two, as expected. This is likely the result of a key release being handled a lot faster than a keypress event. For us, a single peak is still sufficient as a clear threshold can be determined from the plots.

There is some noise in the plot at the beginning. This is not a problem in a real-world attack as such algorithms run for a longer time and, therefore, this noise can be ignored. This noise goes away due to the CPU optimizing code at runtime.

Figure 3.1 shows the results of the different measurements. Additionally, we show that it does not matter how fast a user types, each keypress can be detected. This allows attackers to recover typed sequences by their keypress timing differences.

4 Interrupt-timing Attacks in JavaScript

In this section, we describe how the timing of instructions is measured in a way accurate enough to gain similar outcomes as in the native attack. This was the main part of the attack algorithm that had to be changed from the native approach.

4.1 General Implementation and Adaptions from Native Code

For the attack to work in JavaScript, we implemented our own timestamp functionality as browser vendors changed the resolution of the `performance.now` function as a countermeasure to side-channel attacks.

As shown in Listing 4.1, we implemented a simple tick counter which just reports the number of increments for a given period, in our case four milliseconds. The counter value indicates how many times the function has been scheduled. JavaScript engines in modern browsers block the execution of endless loops. Therefore, we had to bypass this by continuously calling the loop with the `setTimeout` function. Our function would count for four milliseconds and store the counter value afterward. The reason for these values is the minimum timeout for `setTimeout`, which is four milliseconds. Still, we get an acceptable resolution for the timestamp to detect keypress interrupts.

```
results[];
func timing(index)
{
  setTimeout(timing, 0, index+1);
  cnt = 0;
  start = performance.now();
  while((performance.now() - start) < 5)
    cnt++;
  results[index] = cnt;
}
```

Listing 4.1: Receiving highly accurate timestamps in JavaScript

In native code, we did not suffer problems with changing CPU clock frequencies as `RTDSC` runs with the same frequency all the time. In JavaScript, we work with time and counter values which depend on the core frequency and might change over time. For our measurements, we enforced the governor via the `/sys/devices/system/cpu/cpu<CORENR>/cpufreq/scaling_governor` file by setting it with `performance` to always run at the highest CPU frequency.

4.2 Measurement Results

Figure 4.1 shows the results of the measurement for the keystrokes of `admin\n`. The blue dots show the measurements made in JavaScript, and the red squares show the actual time the keys were pressed. The multiple dots are getting to a lower value when a keystroke occurs, are a good indicator for a keypress. The rising dots at the beginning of Figure 4.1 indicate that the code runs more performant after some time, as the JavaScript just-in-time (JIT) compiler will optimize the code.

We used the same hardware as for the native attack. The JavaScript was running in Firefox 52. With our implementation, we achieved an average of about 84 000 increments per four milliseconds. This is equivalent to a resolution of 48 nanoseconds. This is a major improvement to other

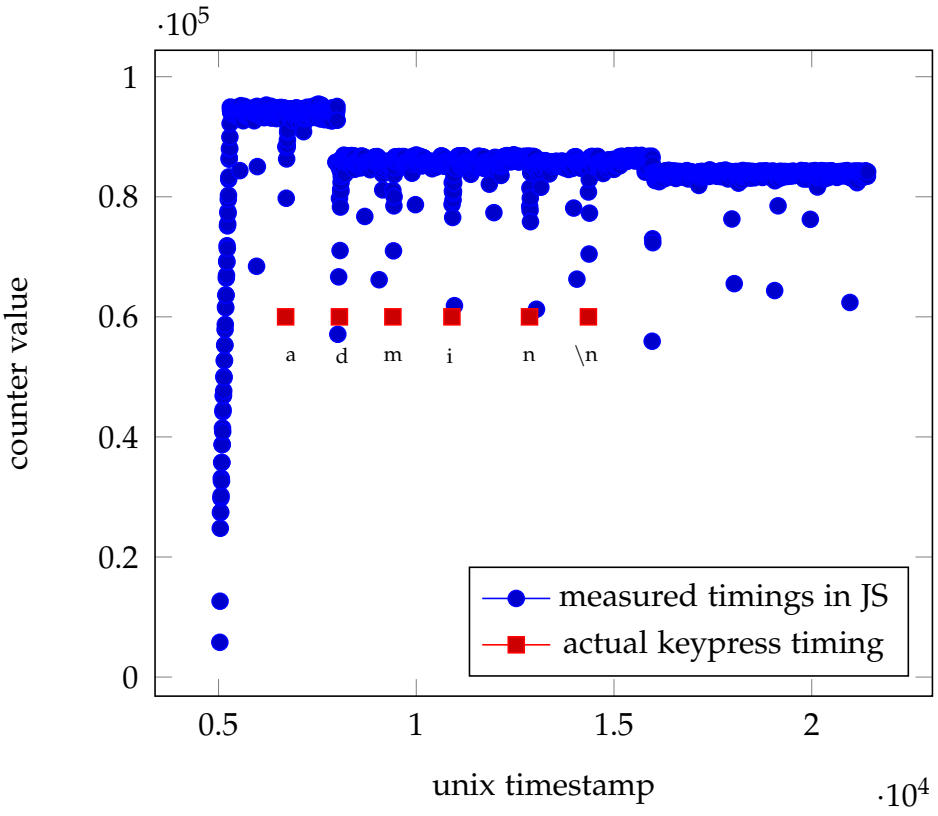

Figure 4.1: JavaScript timing measurement and timestamp of the actual press

alternative timestamps, like Vila and Köpf [14], who reported a resolution from 25 to 100 microseconds.

5 Interrupt-timing Attacks on Mobile Devices

The main difference between the attack in JavaScript working on desktop computers and the one for mobile phones is the kind of interrupts we aim to detect. A touch or swipe of the screen results in multiple interrupts for the operating system, whereas only two occur for a keystroke on a hardware keyboard.

These multiple interrupts result in a very noisy plot for our JavaScript attack. Therefore, we cleared the measured values by applying a sliding window algorithm, to sum up multiple measurement points. Using this we receive clearer peaks in our plots which makes it easier to detect screen touches and swipes on mobile devices. As Figure 5.1 shows, the plots are noisier than those from desktop computers, but peaks for touches can still be detected.

These tests were running on a OnePlus One with a Qualcomm MSM8974AC Snapdragon 801 CPU running Android 6.0.1 and Firefox 54.0a1. With this hardware and the JavaScript implementation, we reach an average of 44000 increments which are equivalent to a resolution of 90 nanoseconds.

Lipp et al. [13] stated that this noise is due to redrawing of the screen after touches. Therefore, a PIN pad which does not trigger these events should be used to achieve a less noisy plot. Additionally, they stated that different phones have different behaviors for screen touches and swipes. Therefore more profiling may be needed to have automatic attacks working on multiple devices.

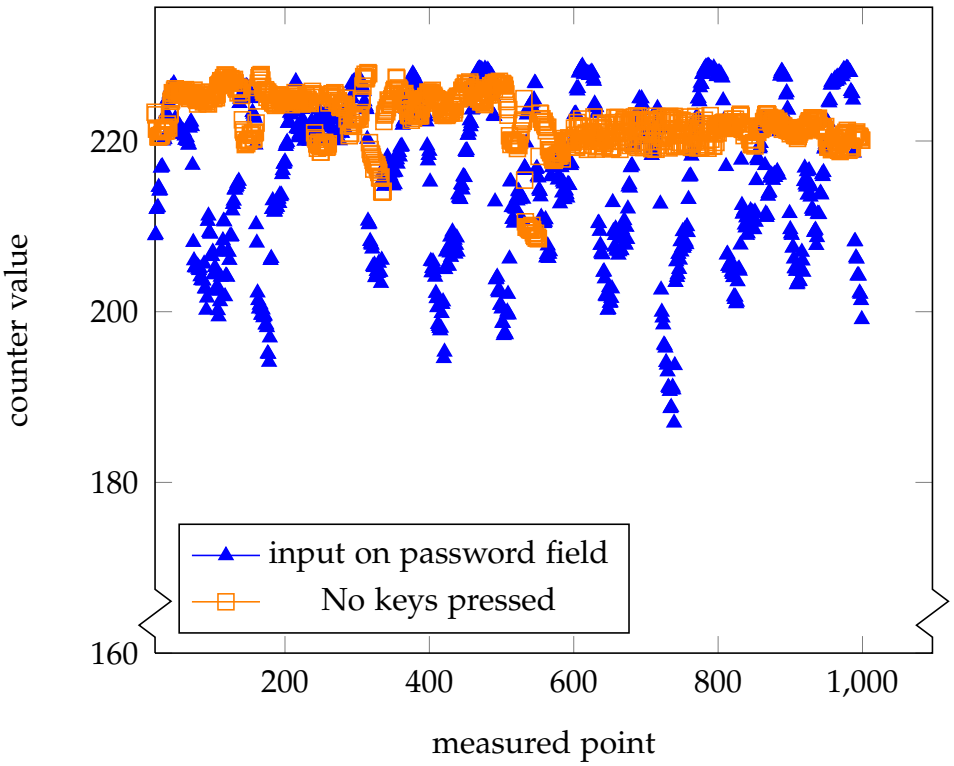

Figure 5.1: JavaScript measurement on a mobile phone using a sliding window algorithm

6 Countermeasures

Several countermeasures were proposed against this kind of attacks. Schwarz et al. [12] stated that randomly injected keystrokes could not be distinguished from real ones. These keystrokes are generated in the kernel driver and are filtered in the user application, in this way the keystrokes need the same processing time as real ones. Hence, they cannot be distinguished by an attacker.

Another countermeasure is to have a dedicated CPU core to handle interrupts. This may sound like a huge overhead on desktop computers, but on mobile phones, multi-CPU architectures already exist. Lipp et al. [13] state that on some phones screen touches and swipes are processed on another CPU than the running applications.

However, another countermeasure has been proposed by Lipp et al. [13] which targets the JavaScript permissions in modern browsers. Many websites do not require all the provided APIs. Especially online advertisements should only have access to a limited number of API calls. A fine-grained permission level would prevent the execution of uncommon API calls, like web workers.

7 Conclusion

Within this project, we implemented a keystroke timing attack in JavaScript. We show that just a small amount of code is used to leak keystroke information to an attacker. As the code can be hidden on any website or online advertisement and a victim will not detect a slight change in system load this attack is very practical.

We showed that this issue is a threat to users of computer systems and smartphones alike. Additionally, not only keystrokes can be detected, but also other I/O interrupts can be found, and an attacker can use this information to gather better profiles of his victims.

As browser vendors tried to prevent timing attacks by making internal timestamps less accurate, we had to build our own accurate time measuring functions in JavaScript. Our measurement methods are even more accurate than the ones which were provided in the past, and from the values achieved we can track back to other information like the CPU frequency.

Information leakages over interrupt timings can be prevented only by applying changes on operating system or CPU architecture level. Some mobile devices already prevent the leakage by using a designated co-processor to handle screen touches. Browser vendors can make it harder to spy on users by limiting the ability to run JavaScript from third parties on embedded advertisements.

Knowledge gained during this project was later used by Lipp et al. [13] to publish a paper at the ESORICS conference under the name of ”Practical Keystroke Timing Attacks in Sandboxed JavaScript”.

Bibliography

[1] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, Cristiano Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS'17*. 2017 (cit. on p. 1).

[2] Daniel Gruss, Clémentine Maurice, Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA'16*. 2016 (cit. on p. 1).

[3] Daniel Gruss, David Bidner, Stefan Mangard. "Practical Memory Deduplication Attacks in Sandboxed JavaScript." In: *ESORICS'15*. 2015 (cit. on pp. 2, 4).

[4] Daniel Gruss, Raphael Spreitzer, Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In: *USENIX'15*. 2015, pp. 897–912 (cit. on p. 3).

[5] Dawn Xiaodong Song, David Wagner, Xuqing Tian. "Timing Analysis of Keystrokes and Timing Attacks on SSH." In: *USENIX Security Symposium*. 2001 (cit. on p. 1).

[6] Edward W. Felten, Michael A. Schneider. "Timing attacks on web privacy." In: *CCS'00*. 2000 (cit. on p. 1).

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 253669-033US. Dec. 2009 (cit. on p. 3).

[8] John Kelsey, Bruce Schneier, David Wagner, Chris Hall. "Side channel cryptanalysis of product ciphers." In: *ESORICS'98* (1998), pp. 97–110 (cit. on p. 3).

[9] Keaton Mowery, Dillon Bogenreif, Scott Yilek, Hovav Shacham. "Fingerprinting information in JavaScript implementations." In: *W2SP'11*. Vol. 2. 11. 2011 (cit. on p. 4).

[10] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, Cyrille Artho. "Memory deduplication as a threat to the guest OS." In: *Fourth European Workshop on System Security*. ACM. 2011, p. 1 (cit. on p. 3).

[11] Michael Schwarz, Clémentine Maurice, Daniel Gruss, Stefan Mangard. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *FC'17*. 2017 (cit. on pp. 1, 4).

[12] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, Stefan Mangard. *KeyDrown: Eliminating Keystroke Timing Side-Channel Attacks*. 2017 (cit. on pp. 1, 5, 6, 16).

[13] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, Stefan Mangard. "Practical Keystroke Timing Attacks in Sandboxed JavaScript." In: *ESORICS'17*. 2017 (cit. on pp. 2, 14, 16, 17).

[14] Pepe Vila, Boris Köpf. "Loophole: Timing Attacks on Shared Event Loops in Chrome." In: *USENIX Security Symposium*. 2017 (cit. on p. 13).

[15] Syed Idrus, Estelle Cherrier, Christophe Rosenberger, Patrick Bours. "Soft Biometrics for Keystroke Dynamics: Profiling Individuals While Typing Passwords." In: *Computers & Security* 45 (2014), pp. 147–155. ISSN: 0167-4048 (cit. on p. 5).

[16] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, Angelos D. Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In: *CCS'15*. 2015 (cit. on p. 4).

[17] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, Jackson Collin. "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks." In: *SP'11*. 2011, pp. 147–161 (cit. on p. 3).