Practical Keystroke Timing Attacks in Sandboxed JavaScript

Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner,
Clémentine Maurice, and Stefan Mangard

Graz University of Technology, Austria

Abstract. Keystrokes trigger interrupts which can be detected through software side channels to reconstruct keystroke timings. Keystroke timing attacks use these side channels to infer typed words, passphrases, or create user fingerprints. While keystroke timing attacks are considered harmful, they typically require native code execution to exploit the side channels and, thus, may not be practical in many scenarios.
In this paper, we present the first generic keystroke timing attack in sandboxed JavaScript, targeting arbitrary other tabs, processes and programs. This violates same-origin policy, HTTPS security model, and process isolation. Our attack is based on the interrupt-timing side channel which has previously only been exploited using native code. In contrast to previous attacks, we do not require the victim to run a malicious binary or interact with the malicious website. Instead, our attack runs in a background tab, possibly in a minimized browser window, displaying a malicious online advertisement. We show that we can observe the exact inter-keystroke timings for a user's PIN or password, infer URLs entered by the user, and distinguish different users time-sharing a computer. Our attack works on personal computers, laptops and smartphones, with different operating systems and browsers. As a solution against all known JavaScript timing attacks, we propose a fine-grained permission model.

Keywords: JavaScript, side channel, interrupt, keystroke, fingerprint

1 Introduction

Keystroke timing attacks are side-channel attacks where an adversary tries to determine the exact timestamps of user key presses. Keystroke timings convey sensitive information that has been exploited in previous work to recover words and sentences [39, 49]. More recently, microarchitectural attacks have been demonstrated to obtain keystroke timings [15, 25, 32, 36] in native code. In particular, the interrupt-timing side channel leaks highly accurate keystroke timings if an adversary has access to a cycle-accurate timing source [36].

JavaScript is the most widely used scripting language and supported by virtually any browser today. It is commonly used to create interactive website elements and enrich the user interface. However, it does not provide access to native instructions, files, or system services. Still, the ability to execute arbitrary

code in the JavaScript sandbox inside a website can also be exploited to perform attacks on website visitors, e.g., timing attacks [12].

JavaScript-based timing attacks were first presented by Felten et al. [12], showing that access times to website elements are lower if a website has recently been visited. Besides attacks on the browser history [12, 21, 46], there have also been more fine-grained attacks recovering information on the user or other websites visited by the user [8, 16, 22, 40, 41]. Vila and Köpf [43] showed that shared event loops in Google Chrome leak timing information on other browser tabs that share worker processes responsible for rendering or I/O.

Previous work has shown that timing side channels which are introduced on the hardware level or the operating system level, can be exploited from JavaScript. Gruss et al. [14] demonstrated page deduplication attacks, Oren et al. [30] demonstrated cache attacks to infer mouse movements and network activity, and Booth [6] fingerprinted websites based on CPU utilization. Gras et al. [13] showed that accurate timing information in JavaScript can be exploited to defeat address-space layout randomization. Schwarz et al. [37] presented a DRAM timing covert channel in JavaScript.

In this paper, we present the first generic keystroke timing attack in sandboxed JavaScript. Our attack is based on the interrupt-timing side channel which has previously only been exploited using native code. We show that this side channel can be exploited from JavaScript without access to native instructions. Based on instruction throughput variations within equally-sized time windows, we can detect hardware interrupts, such as keyboard inputs. In contrast to previous side-channel attacks in JavaScript, our channel provides a more accurate signal for keystrokes, allowing us to observe exact inter-keystroke timings. We demonstrate how this information can be used to infer URLs entered by the user, and distinguish different users time-sharing a computer.

Our attack is generic and can be applied to any system which uses interrupts for user input. We show that our attack code works both on personal computers and laptops, as well as modern smartphones. An adversary can target other browser tabs and browser processes, as well as arbitrary other programs, circumventing same-origin policy, HTTPS security model, and both operating system and browser-level process isolation. With a low impact on the overall system and browser performance, and a code footprint of less than 256 bytes of code, the attack can easily be hidden in modern JavaScript frameworks and malicious online advertisements. Our attack code utilizes new JavaScript features to run in the background, in a background tab, or on a locked phone. Hence, we can spy on the PIN entry used to unlock the phone.

To verify our results, we implemented our attack also in Java without access to native instructions and only low-accuracy timers. We demonstrate that the same timing measurements as in JavaScript can be observed in our Java implementation with a lower noise level. Furthermore, we demonstrate that in a cross-browser covert channel two websites can communicate through network interrupts. These observations clearly show that the source of the throughput differences is caused by the hardware and not specific software implementations.

Our attack works in two phases, an online phase running in JavaScript, and an offline phase running on the adversary's machine. In the offline phase, we employ machine learning techniques to build accurate classifiers trained on keystroke traces gathered in the online phase. These classifiers enable an adversary to infer which website a victim opens and to fingerprint different users time-sharing the same physical machine (e.g., a family sharing a computer).

Our results show that side-channel attacks are a fundamental problem that is not restricted to local adversaries. We propose a fine-grained permission model as a solution against all known JavaScript timing attacks. The browser restricts access to specific features and prompts the user to grant permissions per domain.

Our key contributions are:

- We show the first generic keystroke timing attack in JavaScript, embedded in a website, targeting arbitrary other tabs, processes and programs.
- We demonstrate our attack on personal computers, laptops and smartphones, with different browsers and operating systems.
- We demonstrate that our attack can obtain the exact inter-keystroke timings for a user's PIN or password, infer URLs entered by the user, and distinguish different users time-sharing a computer based on their input.

Outline The remaining paper is organized as follows. In Section 2, we provide background. We describe our attack in Section 3. In Section 4, we present the performance of our attack on personal computers and smartphones. We discuss countermeasures in Section 5. Finally, we conclude in Section 6.

2 Background

2.1 Keystroke Timing Attacks

Keystroke timing attacks acquire accurate timestamps of keystrokes for input sequences. These keystroke timestamps depend on several factors such as bigrams, syllables, words, keyboard layout, and typing experience [33]. An adversary can exploit these timing characteristics to learn information about the user or the user input. Existing attacks use machine learning to infer typed sentences or recover passphrases [38, 39, 49]. Idrus et al. [19] showed that key press and key release events can be used to fingerprint users.

The Linux operating system exposes information that allows compiling accurate traces of keystroke timings [39, 49]. Zhang et al. [49] demonstrated that instruction and stack pointer, interrupt statistics, and network packet statistics can be used as side channels for keystroke timings. While Song et al. [39] demonstrated that SSH leaks inter-keystroke timings in interactive mode, Hogye et al. [17] showed that network latency in networks with significant traffic conceals these inter-keystroke timings in practice. Kamran et al. [3] showed that it is possible to detect keystrokes and classify the typed keys using Wi-Fi Signals. Jana and Shmatikov [20] showed that CPU usage is a much more reliable side channel for keystroke timings than the instruction pointer, or the stack pointer. Diao et al. [11] demonstrated high-precision keystroke timing attacks based on

Algorithm 1: Online phase of an interrupt-timing attack

```
input : threshold
now ← get_timestamp();
while true do
    last ← now;
    now ← get_timestamp();
    if now − last > threshold then
        report(now, diff);
```

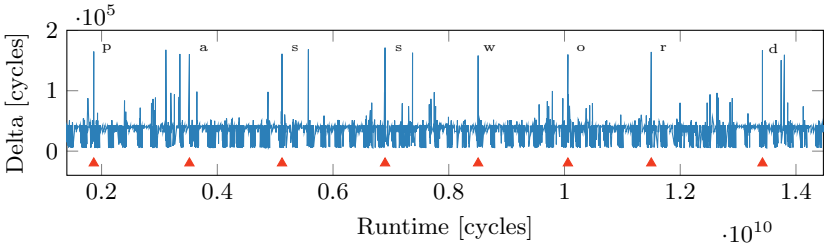

Fig. 1: Native interrupt-timing attack: The difference between consecutive timestamps is measured while a sentence is typed. Every keystroke leads to a significant deviation as the measuring program is interrupted by the keyboard.

`/proc/interrupts`. Mehrnezhad et al. [27] used the JavaScript sensor API to detect touch, hold, scroll, and zoom actions on mobile devices using built-in sensors such as accelerometer and gyroscope.

Cache attacks have also been used to obtain keystroke timings. In a cache attack, the adversary observes effects of the victim's operation on the cache and can then deduce what operations the victim performed. Ristenpart et al. [34] demonstrated a keystroke timing attack using a Prime+Probe cache attack. Gruss et al. [15] demonstrated that Flush+Reload cache attacks can be used for keystroke timing attacks.Similarly, Pessl et al. [32] showed a keystroke timing attack on the Firefox address bar using the DRAM as a side channel.

Recently, it was shown that keystroke interrupt timings can be obtained in a timing attack which continuously measures differences between consecutive `rdtsc` calls [36]. However, this is not possible if the adversary only controls a website that is visited by the victim. Sandboxed JavaScript running on a website cannot utilize any native instructions such as `rdtsc`.

2.2 Interrupt-timing Attacks

Interrupt-timing attacks have recently been demonstrated in native code to recover keystroke timings [36]. The basic idea of interrupt-timing attacks is to continuously acquire a high-resolution timestamp and to monitor differences between subsequent timestamps, *i.e.*, how much time has passed since the last measurement, as outlined in Algorithm 1. Significant differences occur when-

ever the measuring process is interrupted. The more time the operating system consumes to handle the interrupt, the higher the measured differences are. Especially interrupts triggered by I/O devices—such as keyboards—lead to clearly visible peaks in the measured trace. Figure 1 shows a trace from a native attack implementation while a user typed in a sentence. The exact timestamp where the user pressed a key is clearly visible and can be distinguished from other events. However, the trace does not only contain keyboard interrupts and, thus, allows spying on user input but also on every other event that causes one or more interrupts, e.g., network traffic or redraw events. An adversary can filter relevant peaks by means of post-processing algorithms to monitor entered keystrokes.

2.3 Timing Attacks in Sandboxed JavaScript

JavaScript has evolved to be the most widely supported scripting language, notably because it is supported by virtually every modern browser. With highly-optimized just-in-time compilation, modern JavaScript engines deliver a performance that can compete with native code implementations. The timestamp counter provides a cycle-accurate timestamp to user programs in native code, but it is not accessible from JavaScript. Instead, JavaScript provides the High Resolution Time API [45] (`performance.now`) for sub-millisecond timestamps.

Based on this timing interface, various attacks have been demonstrated. Van Goethem et al. [41] were able to extract private data from users by measuring the differences in the execution time from cross-origin resources. Stone [40] showed that the optimization in SVG filters introduced timing side channels. He showed that this side channel can be used to extract pixel information from iframes. Booth [6] fingerprinted websites based on CPU utilization—interfering with the execution time of a benchmark function—when loading and rendering the page.

Gruss et al. [14] showed that page deduplication timing attacks can be performed in JavaScript to determine which websites the user has currently opened. Oren et al. [30] showed that it is possible to mount cache attacks in JavaScript. They demonstrated how to perform Prime+Probe attacks in the browser to build cache covert channels but also to spy on the user's mouse movements and network activity through the cache. This attack caused all major browsers to decrease the resolution of the `performance.now` method [1, 7, 10]. The W3C standard now recommends a resolution of 5 µs while the Tor project reduced the resolution in the Tor browser to a more conservative value of 100 ms [28]. Gras et al. [13] showed that accurate timing information in JavaScript can be exploited to defeat address-space layout randomization. Vila and Köpf [43] showed that shared event loops in Google Chrome leak timing information about other browser tabs sharing worker processes for rendering and I/O operations. They exploit this side channel to identify web pages, to build a covert communication channel, and to infer inter-keystroke timings.

Recently, several works investigated timing primitives in JavaScript that allow recovering highly accurate timestamps [13,24,37]. We use these timing primitives to build highly accurate keystroke timing attacks in sandboxed JavaScript.

3 Sandboxed Keystroke Timing Attacks without High-resolution Timers

Our attack follows the same idea as interrupt-timing attacks in native code [36]. It consists of an online phase where timing traces are acquired on a victim machine and an offline phase for post-processing and evaluation.

Online phase. In the online phase of our attack, we run an interrupt-timing attack in sandboxed JavaScript. Interrupt-timing attacks have only minimal requirements, most importantly access to the x86 `rdtsc` instruction [36]. Consequently, keystroke interrupt-timing attacks have only been demonstrated in native code. We face several challenges to perform keystroke interrupt-timing attacks from remote websites, as JavaScript can neither execute this instruction nor run endless loops on websites.

There is no high-resolution timestamp available in JavaScript, as the resolution of `performance.now` is limited to 5 µs to mitigate side-channel attacks [45]. Therefore, we implement a counter to simulate a monotonic clock by constantly incrementing a value [13, 24, 37, 47]. The number of increments, *i.e.*, the instruction throughput, is proportional to the time the counter function is scheduled. Thus, any interrupt reduces the instruction throughput and, therefore, leads to a lower number of increments within a fixed time frame. Consequently, we can read the counter value at fixed time intervals and deduce from the number of increments since the last interval whether the counter function was interrupted.

As JavaScript is based on a single-threaded event loop, browsers usually do not allow websites to use endless loops and inform the user when detecting such a construct. The usual solution is to either use `setTimeout` or `setInterval` to constantly trigger execution of the loop body after a specified number of milliseconds have passed. However, these functions enforce a minimum pause of 4 ms before scheduling the same code again, yielding a resolution that is significantly lower than the resolution of `performance.now`.

To work around this limitation, we introduce a new variant of previously published timing primitives [13, 24, 37] called cooperative endless-loop slicing. The idea is to slice the endless loop into smaller finite loops where every loop slice has an execution time of approximately 4 ms. Before running this loop, we schedule the next loop slice using `setTimeout` with a timeout of 4 ms. Thus, in the optimal case, the next slice of the endless loop is executed immediately after the current slice, giving the impression of an actual endless loop. However, as higher priority events, such as user inputs, can still be processed between the loop slices, the browser is responsive and will not stop the endless loop. Algorithm 2 illustrates how we use this construct to continuously schedule our counter to obtain continuous timing traces.

The instruction throughput per loop slice, *i.e.*, the counter increments, varies depending on how often and how long the thread was interrupted during this loop slice. Within one loop slice, we achieve on average 72 764 increments of the counter, resulting in a resolution of approximately 69 ns ($\sigma = 3$ ns, $n = 4000$) on an Intel i5-6200U. This resolution is three orders of magnitude higher than the

Algorithm 2: Interrupt-timing attack implemented in JavaScript

```
Function measure_time(id):
    setTimeout(measure_time, 0, id + 1);
    counter ← 0;
    begin ← window.performance.now();
    while (window.performance.now() - begin) < 5 do
        counter ← counter + 1;
    publish(id, counter);
```

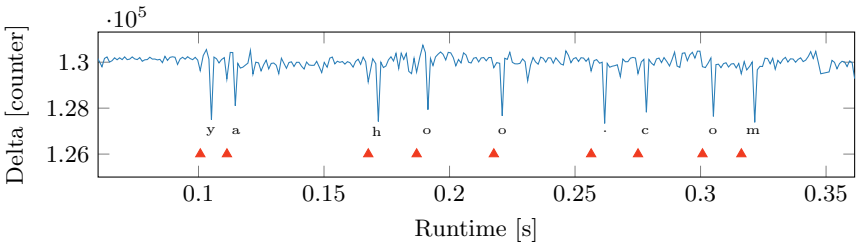

Fig. 2: Interrupt-timing attack in JavaScript: The lower peaks indicate that the measured script has been interrupted, allowing to infer single keystrokes.

result of Vila and Köpf [43] who achieved a resolution of only 25 µs to 100 µs. On ARM, we achieve on average 5038 increments on the Google Nexus 5 and 17 454 increments on the OnePlus 3T, yielding a resolution of 994 ns ($\sigma = 55$ ns, $n = 4000$) and 287 ns ($\sigma = 4$ ns, $n = 4000$) respectively.

A further limitation of JavaScript is that once the user switches the tab or minimizes the browser, the default minimum timeout value of 4 ms is reduced to 1000 ms. Increasing the loop slices to 1000 ms is not practical since it would make the browser unresponsive again. In order to circumvent this issue, we utilize the Web Worker API which explicitly allows JavaScript code to be executed in the background [44]. We discovered that the minimum timeout is not reduced for web workers and we can still measure interrupt timings with a high frequency. This allows us to monitor keystrokes when the victim is visiting a different page or even a different application.

Figure 2 shows a measured trace while a user typed the URL `yahoo.com` into the browser bar. If no interrupt occurs, the counter variable has been incremented for the full time window of 4 ms, defining the baseline. If an interrupt disrupts the measuring JavaScript, the counter variable is not incremented as often in the same time window, yielding to downward-facing peaks. Thus, the typed letters leave clear marks in the measured trace, which allows inferring single keystrokes.

Offline phase. In the offline phase of the attack, the measurements gathered from the online phase are processed and analyzed. Over time, an adversary can gather thousands of traces in order to learn about the individual typing behavior of the victim or to derive an entered passphrase or PIN code. Depending on the goal of

the adversary, different methods to evaluate the gathered data can be applied. In order to detect single keystrokes in a measured trace, we filter the measured trace in order to reduce noise and to deduce threshold values for keystrokes by manually inspecting one recorded trace of the target device. Using this threshold, we can further reduce the number of points in recorded traces to a minimum and, thus, increase the performance of further computations. We build a classifier by calculating the correlation between our training set and the queried trace. In order to classify entered words, we need to take into account that the points in time where a character has been entered can vary in time in our trace. Therefore, we use k-nearest neighbors (k-NN) classification [4] and calculate the correlation of the trace with every other trace in the training set using different alignments. We chose the alignment that yields the highest correlation and decide on the class giving the best match. While more computational expensive methods working with time series [5, 35] to build classifiers exist [9, 23, 48], we show that the features of the recorded measurements are strong enough such that also simpler techniques allow to build an efficient and accurate classifier.

4 Practical Attacks and Evaluation

In this section, we demonstrate the significant attack potential of our JavaScript interrupt-timing attack. Our attack does not depend on any specific browser or operating system and can therefore be performed on personal computers, laptops and smartphones. We show that it is possible to infer which website a user has entered into the browser's address bar and to profile different users sharing the same computer. Furthermore, we show that the attack can be utilized to obtain the exact timings of every digit of the PIN that is used to unlock the phone while the attack code is executed in the web browser running in the background.

4.1 URL Classification

In our first experiment, we demonstrate that using our JavaScript keystroke timing attack on a personal computer in combination with machine learning techniques, we can infer URLs that a user has entered into the address bar of the browser. We train a classifier to successfully label measurement traces of user input sequences for the URLs of the top 10 most visited websites [2]. For this experiment we used an Intel i7-6700K CPU and Firefox 52.0 running on Linux.

Every single trace consists of timestamps with a corresponding counter value (cf. Section 3) and the corresponding URL. As there are small timing variations when the user starts typing the URL and whenever the user pressed a key, the length of the trace as well as the position of the features, *i.e.*, the characteristics in the measured values describing a key stroke, within the trace varies. Thus, we need to build our classifier in a way that overcomes those difficulties. In a preparation step, we determine the maximum trace length as well as the timestamp resolution. The resolution can be obtained from the greatest common divisor of

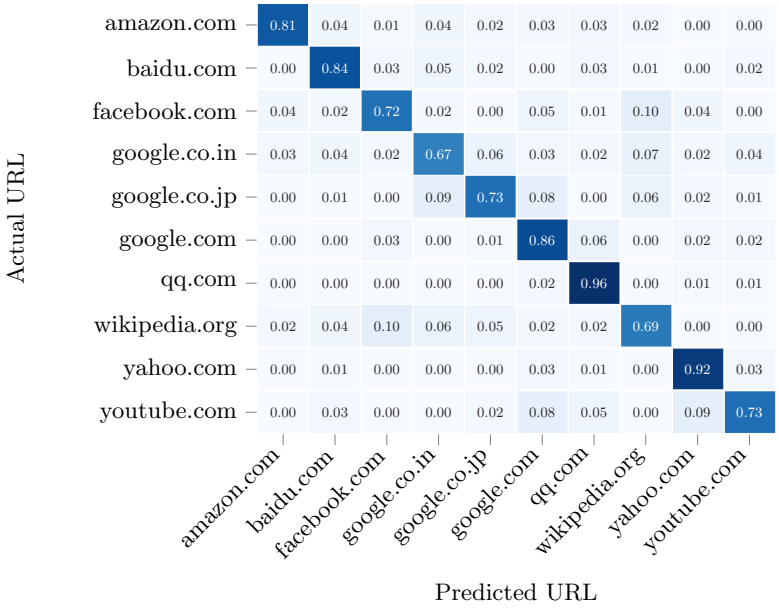

Fig. 3: Confusion matrix for URL input. The user input can be correctly predicted with a probability of 67 % in the worst case and 96 % in the best case. The probability of random guessing is 10 %.

all measured timestamps of all samples. Finally, we create a linear interpolation of every sample based on the actual resolution.

The classifier assigns a class label to an unlabeled trace where each class corresponds to one URL that we train our classifier with. In order to classify a new trace, we compute the correlation of the new trace with a fixed number of randomly chosen samples for every class. As the timestamps where the user started entering the URL vary, we need to compute the correlation of two traces for different alignments. Thus, we shift one trace within a fixed time window back and forth in order to find an alignment where the correlation reaches its maximum. The average of the five highest correlations for each class decides which class the trace belongs to, *i.e.*, we choose the highest average correlation.

We evaluate our classifier by using k-fold cross-validation. We first randomly draw 20 samples as *training set* from our collected 100 measurements from every class. We then test the classifier on a randomly drawn set of the remaining 800 samples (80 per class), the *test set*. We cross-validate our classifier by performing this evaluation multiple times with randomly selected training sets.

Figure 3 shows the confusion matrix. Every cell shows the probability that the classifier labels a sample of a class specified by the row into a certain class specified by the column. We can clearly see that for every domain the classifier proposes the correct class with a higher probability than an incorrect one, and a significantly higher probability than random guessing (10 %). The identification

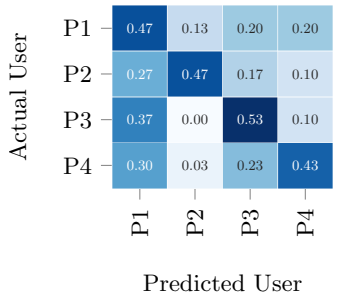

Fig. 4: Confusion matrix for input by different users. The user can be correctly predicted with a probability of 43 % in the worst case and 53 % in the best case. The probability of random guessing is 25 %.

Table 1: Mobile test devices.

Device	SoC	Keystrokes	Screen lock
Google Nexus 5	Qualcomm MSM8974 Snapdragon 800	✓	-
Xiaomi Redmi Note 3	Mediatek MT6795 Helio X10	✓	✓
Homtom HT3	MediaTek MTK6580	✓	✓
Samsung Galaxy S6	Samsung Exynos 7420	-	✓
OnePlus One	Qualcomm MSM8974AC Snapdragon 801	✓	✓
OnePlus 3T	Qualcomm MSM8996 Snapdragon 821	-	-

rate of *qq.com* in comparison with other domains is also very high as the domain contains only a small number of characters to be typed. The overall identification rate of our classifier is 81.75 %.

4.2 User Classification

As a second experiment, we evaluate whether it is possible to distinguish different users in order to determine who is actually sitting in front of the personal computer. In order to do so, we have collected only 5 traces of the input of the top 10 most visited websites [2] of 4 different persons to train the classifier. The results with 2 training set and 3 test set traces for each user are illustrated as a confusion matrix in Figure 4. While it is much harder to determine the user responsible for the given trace, our classifier is with an overall identification rate of 47.5 % still better than random guessing.

4.3 Touchscreen Interactions

In our third experiment we show that interrupt-timing attacks also work on modern smartphones and on different web browsers. Although battery saving techniques should make attacks harder, the attack can still be applied if the measuring program is executed in a different tab or if the browser app is running in background. Furthermore, we show that the attack can be used to detect when the screen is locked and unlocked.

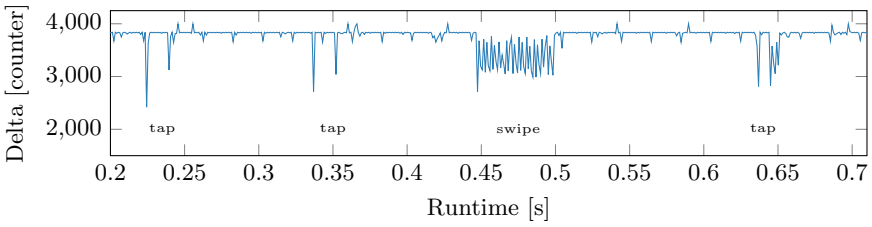

Fig. 5: Keystroke timing attack running in a native app on the Google Nexus 5.

Mobile phones usually use a soft-keyboard that is displayed on the screen. Every tap on the screen causes a redraw event that is clearly visible in the measured trace, making it easier to detect when a user touches the screen. While the redraw event is sufficient to monitor taps on the keyboard, we want to be able to identify any tap on the device, whether it causes a redraw event or not. Therefore, our test website implemented a custom touch area imitating a PIN pad.This touch area does neither register any events nor does it change its appearance. Thus, a touch onto this PIN pad should not issue any event at all, eliminating all events from the trace that are not caused by the touch interrupt itself. We provide the code for this experiment online.[1]

To cross-check whether we actually observe hardware events and not some browser-internal events, we implemented the same interrupt-detection algorithm in a native Android app. To achieve comparable results for the recorded traces, we reduced the timer resolution to 5 µs in the same way as Firefox and Chrome.

For our experiments, we used a Google Nexus 5 with a Qualcomm MSM8974 Snapdragon 800 SoC running Android 6.0.1 with Chrome 44.0.2403.133 and Firefox 54.0a1. Our second testing device is a Xiaomi Redmi Note 3 with a Mediatek MT6795 Helio X10 running Android 5.0.2 with Chrome 57.0.2987.132 and Firefox 52.0.2. In addition, we used all the device listed in Table 1 to record traces using the JavaScript implementation for visual inspection. Table 1 also shows whether we could detect keystrokes and screen locks without machine learning just by visual inspection.

Keystroke detection. Figure 5 shows the keystroke timing attack in a native Android app on a Google Nexus 5 where a user tapped the screen twice, before swiping once and tapping it again. The individual interrupts, caused by tapping on the phone, can easily be identified by the two following peaks representing the touch and release event. If the user swipes over the screen, many interrupts are triggered, one for every coordinate change. This results in many visible peaks and, thus, swipes and taps can be distinguished.

Our JavaScript implementation of the keystroke timing attack runs successfully in Chrome and allows distinguishing taps from swipes as illustrated in Figure 6. While in contrast to the native implementation, the measurements in

[1] `https://github.com/IAIK/interruptjs`

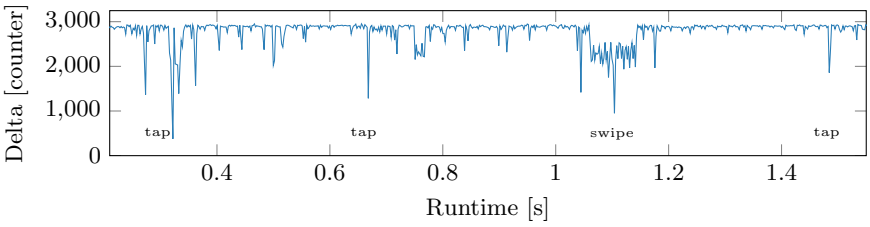

Fig. 6: Keystroke timing attack running in Chrome on the Google Nexus 5.

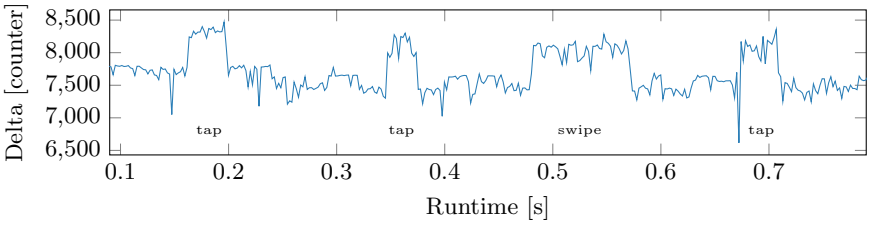

Fig. 7: Keystroke timing attack running in Chrome on the Xiaomi Redmi Note 3. The peaks face upwards instead of downwards as with other devices.

JavaScript contain much more noise, the exact tap timings can easily be extracted and allow further, more sophisticated attacks.

Figure 7 shows the same trace of two taps, one swipe and one additional tap on the Xiaomi Redmi Note 3. Surprisingly, the peaks caused by the interrupts face upwards instead of downwards as one might expect. We observed that the Xiaomi Redmi Note 3 increases the CPU frequency whenever the screen is touched. Consequently, although the interrupt will consume some CPU time, the counter as described in Section 3 can be incremented more often due to the significantly higher CPU frequency. We have verified this behavior by running a benchmark suite on the Xiaomi Redmi Note 3. The benchmark suite has been up to 30 % faster, when swiping over the screen while the benchmark is executed. While this feature may be useful to handle touch interrupts more efficiently and to appear more responsive, it also opens a new side channel and allows detecting tap and screen events easily. We also verify the same behavior in our native Java implementation with higher peaks which allows detecting tap and swipe events even more reliably. On the OnePlus 3T we were not able to detect keystrokes at all. We suspect that this is due to the big.LITTLE architecture, which moves the CPU-intensive browser task to a high-performance ARM core, while the interrupts are handled by smaller cores. Thus, the browser is not interrupted if a hardware interrupt occurs.

Spying on other applications and PIN unlock. While the attack of Vila and Köpf [43] is limited to spy on tabs or pop-ups opened by the adversary, our attack is not restricted and can be used to monitor any other application running

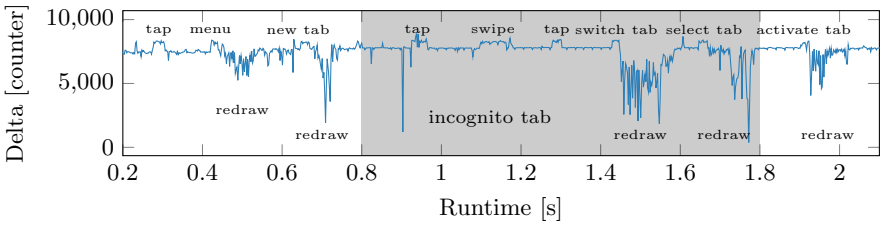

Fig. 8: Keystroke timing attack running while switching to a different tab in the Chrome browser on the Xiaomi Redmi Note 3.

on the system. Indeed, the attack of Vila and Köpf relies on the timing difference caused by the event loop of the render process, thus only tabs or windows sharing the same rendering process can be attacked. In contrast, our interrupt-timing attack is not restricted to the browser and its child processes as it allows monitoring every other event triggering interrupts on the target device. Moreover, our attack also provides a much higher resolution, which allows detecting interrupts triggered by user input more reliably.

Figure 8 shows a trace of a victim opening a website running the measurement code in Chrome on the Xiaomi Redmi Note 3. In addition, the victim opens a tab in incognito mode and taps the screen multiple times. We can even detect these user interactions in different tabs as the attack takes advantage of web workers which are not throttled when running in the background. Thus, the incognito mode offers no protection against our attack.

In the next scenario, we show that our attack is not restricted to processes of the browser application but can be used to spy on every other application as well. The victim visits the website running the measuring application in the Firefox app on the Xiaomi Redmi Note 3 and continues using the phone, switching to other tabs or applications, and later locks the screen. After some time the victim turns on the screen again, where the lock screen prompts the victim for the PIN code. Finally, the victim enters the PIN code, unlocking the phone. Figure 9 shows a trace of this scenario. We can clearly observe when the screen is turned off as the CPU frequency is lowered to save battery, as well as when the screen is turned on again. Furthermore, we can extract the exact timestamps where the victim entered the 4-digit PIN and the subsequent redraw event.

4.4 Covert Channel

In our fourth experiment, we implement a covert communication channel based on our attack. This allows us to estimate the maximum number of interrupts we can detect. We establish a unidirectional communication with one sender and one receiver. The receiver simply mounts the interrupt-timing attack to sense any interrupts. The sender has to issue interrupts to send a '1'-bit or idle to send a '0'-bit. There is no JavaScript API which allows to explicitly issue interrupts, thus we require an API that implicitly issues an interrupt.

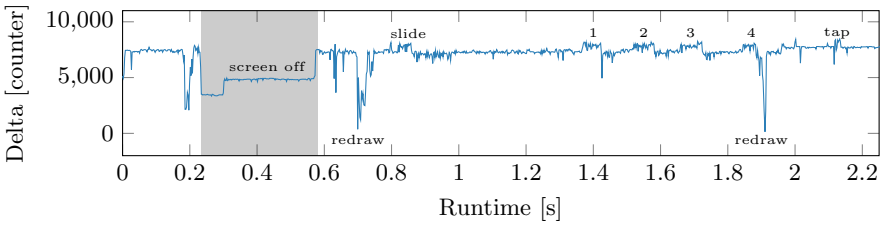

Fig. 9: Keystroke timing attack running in the Firefox browser on the Xiaomi Redmi Note 3. While the user locked the screen, the application still detects keystrokes as long as it is executed on the last used tab. The application extracts the exact inter-keystroke timings for the PIN input used to unlock the device.

We use `XMLHttpRequest`s to fetch a network resource from an invalid URL. Every `XMLHttpRequest` which cannot be served from the cache will create a network connection and therefore issue I/O interrupts. Even if the URL cannot be resolved, either because there is no Internet connection, or the URL is invalid, we are able to see the I/O interrupts. Such a covert channel based on hardware interrupts circumvents several protection mechanisms found in modern browsers.

Cross-tab channel. Using the covert channel across tabs breaks two security mechanisms. First, the same origin policy—which prevents any communication between scripts from different domains—does not apply anymore. Thus, scripts can communication across domain borders. Second, due to the security model of browsers, there is no way a HTTPS page is able to load HTTP content. For the covert channel, this security model does not hold anymore.

Cross-browser channel. As the interrupt-timing is not limited to a process, the covert channel circumvents policies such as proccess-per-site or process-per-tab which prevent sites or tabs from sharing process resources. The covert channel can even be used as a cross-browser communication channel. We tested a transmission from Firefox to Chrome and achieved the same transmission rate as in the cross-tab scenario. The communication channel can also be established with a browser instance running in incognito mode.

In all scenarios, the receiver uses a constant sampling interval of 40 ms per bit, resulting in a raw transmission rate of 25 bps. Thus, we are also able to spy on 25 interrupts per second in all those scenarios which is sufficient to monitor keystrokes of even the fastest typists [33]. To reliably transmit data over the covert channel, we can apply the techniques proposed by Maurice et al. [26].

5 Countermeasures

5.1 A Fine-grained Permission Model for JavaScript

In order to impede and mitigate our interrupt-timing attack and other similar side-channel attacks in JavaScript, we propose a more fine-grained permission

model for JavaScript running in web browsers. For instance, the existing permission system of Firefox only allows managing the access control to a limited number of APIs. However, as many websites do not require functionality such as web workers. The user should be capable to allow on a per-page level such features. If an online advertisement running potential malicious code requests for permissions to uncommon APIs, the fine-grained permission system prevents its further execution.

5.2 Generic Countermeasures

Myers [29] evaluated how various user-mode keylogging techniques in malware on Windows are implemented and suggested to generate random keyboard activity by injecting phantom keystrokes that will be intercepted by the malware. Furthermore, Ortolani [31] analyzed the statistical properties of noise necessary to impede the detection of real keystrokes in a noisy channel. While both do not protect against the interrupt-timing attack, Schwarz et al. [36] published a proof-of-concept countermeasure that aims to protect against this type of attacks. The countermeasure injects a large number of fake keystrokes that propagate through the kernel driver up to the user space application. We have verified that the countermeasure successfully injects fake keystrokes that cannot be distinguished from real interrupts by our implementation. Figure 10 shows a trace measured on the Google Nexus 5 with the countermeasure enabled. While this countermeasure appears to prevent this attack on personal computers as well, it remains unclear whether it closes the side channel on the Xiaomi Redmi Note 3 where the CPU gets overclocked for every touchscreen input. As the implementation of the countermeasure only supports the touchscreen of the Google Nexus 5 and the OnePlus 3T, we could not evaluate it against our attack on the Xiaomi Redmi Note 3. Therefore, we were unable to verify whether the fake keystrokes injected by the countermeasure also trigger the CPU overclocking and, thus, if the countermeasure protects against this attack on devices with such a behavior.

Kohlbrenner and Shacham [24] implemented the fuzzy time concept [18,42] in order to eliminate high-resolution timers. While this would prevent our attack in its current implementation, we could use the experimental `SharedArrayBuffers` as suggested by Schwarz et al. [37] and Gras et al. [13] in order to obtain a resolution of up to 2 ns and, thus, to re-enable our attack.

6 Conclusion

In this paper, we presented the first JavaScript-based keystroke timing attack which is independent of the browser and the operating system. Our attack is based on capturing interrupt timings and can be mounted on desktop machines, laptops as well as on smartphones. Because of its low code size of less than 256 bytes, it can be easily hidden within modern JavaScript frameworks or within an online advertisement, remaining undetected by the victim. We demonstrated the potential of this attack by inferring accurate timestamps of keystrokes as

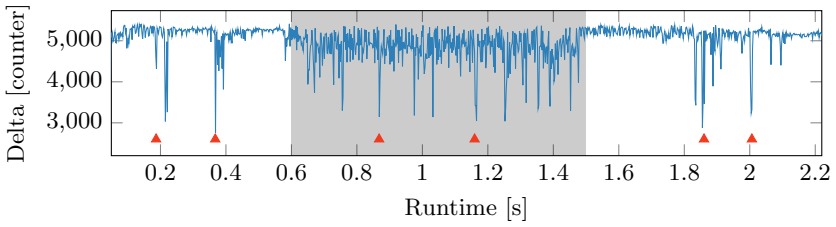

Fig. 10: Measurement of the keystroke timing attack running in the Chrome Browser on the Google Nexus 5. The red rectangles show when the user tapped the screen. In the gray area, we enabled the countermeasure [36], making it infeasible to distinguish real keystrokes from fake keystrokes.

well as taps and swipes on mobile devices. Based on these keystroke traces, we built classifiers to detect which websites a user has visited and to identify different users time-sharing a computer. Our attack is highly practical, as it works while the browser is running in the background, allowing to spy on other tabs and applications. As the attack is also executed when the phone is locked, we demonstrated that we can monitor the PIN entry that is used to unlock the phone. Finally, as a solution against our attack and other similar side-channel attacks in JavaScript, we proposed a fine-grained permission model for browsers.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback. This project has been supported by the COMET K-Project DeSSnet (grant No 862235) conducted by the Austrian Research Promotion Agency (FFG) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

1. Alex Christensen: Reduce resolution of performance.now. (2015), `https://bugs.webkit.org/show_bug.cgi?id=146531`
2. Alexa Internet, Inc.: The top 500 sites on the web (Dec 2016), `http://www.alexa.com/topsites`
3. Ali, K., Liu, A.X., Wang, W., Shahzad, M.: Keystroke recognition using wifi signals. In: Proceedings of the 21st Annual International Conference on Mobile Computing and Networking. MobiCom'15 (2015)
4. Altman, N.S.: An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician 46(3), 175–185 (1992)
5. Berndt, D.J., Clifford, J.: Using Dynamic Time Warping to Find Patterns in Time Series. In: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining (1994)

6. Booth, J.M.: Not So Incognito: Exploiting Resource-Based Side Channels in JavaScript Engines. Bachelor Thesis, Harvard School of Engineering and Applied Sciences (2015)
7. Boris Zbarsky: Reduce resolution of performance.now. (2015), https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab
8. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: WWW'07 (2007)
9. Chen, W., Chang, W.: Applying hidden Markov models to keystroke pattern analysis for password verification. In: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (2004)
10. Chromium: window.performance.now does not support sub-millisecond precision on Windows (2015), https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110
11. Diao, W., Liu, X., Li, Z., Zhang, K.: No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: S&P'16 (2016)
12. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: CCS'00 (2000)
13. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS'17 (2017)
14. Gruss, D., Bidner, D., Mangard, S.: Practical memory deduplication attacks in sandboxed javascript. In: ESORICS'15 (2015)
15. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
16. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks: stealing the pie without touching the sill. In: CCS'12 (2012)
17. Hogye, M.A., Hughes, C.T., Sarfaty, J.M., Wolf, J.D.: Analysis of the Feasibility of Keystroke Timing Attacks over SSH Connections. Tech. rep., School of Engineering and Applied Science University of Virginia (2001)
18. Hu, W.M.: Reducing timing channels with fuzzy time. Journal of Computer Security (1992)
19. Idrus, S., Cherrier, E., Rosenberger, C., Bours, P.: Soft Biometrics for Keystroke Dynamics: Profiling Individuals While Typing Passwords. Computers & Security 45, 147–155 (2014)
20. Jana, S., Shmatikov, V.: Memento: Learning Secrets from Process Footprints. In: S&P'12 (2012)
21. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in javascript web applications. In: CCS'10 (2010)
22. Jia, Y., Dong, X., Liang, Z., Saxena, P.: I know where you've been: Geo-inference attacks via the browser cache. IEEE Internet Computing 19(1), 44–53 (2015)
23. Kobojek, P., Saeed, K.: Application of Recurrent Neural Networks for User Verification based on Keystroke Dynamics. Journal of Telecommunications and Information Technology (3), 80 (2016)
24. Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium (2016)
25. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
26. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17 (2017)
27. Mehrnezhad, M., Toreini, E., Shahandashti, S.F., Hao, F.: Touchsignatures: identification of user touch actions and pins based on mobile sensor data via javascript. Journal of Information Security and Applications (2016)

28. Mike Perry: Bug 1517: Reduce precision of time for Javascript. (2015), https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517
29. Myers, M.: Anti-Keylogging with Random Noise. In: PoC|GTFO. vol. 0x14 (2017)
30. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS'15 (2015)
31. Ortolani, S.: Noisykey: Tolerating keyloggers via keystrokes hiding. In: USENIX Workshop on Hot Topics in Security – HotSec (2012)
32. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
33. Pinet, S., Ziegler, J.C., Alario, F.X.: Typing is writing: Linguistic properties modulate typing execution. Psychon Bull Rev 23(6), 1898–1906 (Apr 2016)
34. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS'09 (2009)
35. Rumelhart, D.E., McClelland, J.L., PDP Research Group, C. (eds.): Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. MIT Press (1986)
36. Schwarz, M., Lipp, M., Gruss, G., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: Keydrown: Eliminating keystroke timing side-channel attacks (2017)
37. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In: FC'17 (2017)
38. Simon, L., Xu, W., Anderson, R.: Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. Proceedings on Privacy Enhancing Technologies (2016)
39. Song, D.X., Wagner, D., Tian, X.: Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security Symposium (2001)
40. Stone, P.: Pixel perfect timing attacks with html5. Context Information Security (White Paper) (2013)
41. Van Goethem, T., Joosen, W., Nikiforakis, N.: The clock is still ticking: Timing attacks in the modern web. In: CCS'15 (2015)
42. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating fine grained timers in Xen. In: CCSW'11 (2011)
43. Vila, P., Köpf, B.: Loophole: Timing attacks on shared event loops in chrome. In: USENIX Security Symposium (2017)
44. W3C: Web Workers - W3C Working Draft 24 September 2015 (2015), https://www.w3.org/TR/workers/
45. W3C: High Resolution Time Level 2 (2016), https://www.w3.org/TR/hr-time/
46. Weinberg, Z., Chen, E.Y., Jayaraman, P.R., Jackson, C.: I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In: S&P'11 (2011)
47. Wray, J.C.: An analysis of covert timing channels. Journal of Computer Security 1(3-4), 219–232 (1992)
48. Xi, X., Keogh, E., Shelton, C., Wei, L., Ann Ratanamahatana, C.: Fast Time Series Classification Using Numerosity Reduction. In: Proceedings of the 23rd International Conference on Machine Learning (2006)
49. Zhang, K., Wang, X.: Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security Symposium (2009)