Tracking Private Browsing Sessions using CPU-based Covert Channels

Nikolay Matyunin
Nikolaos A. Anagnostopoulos
Spyros Boukoros
Markus Heinrich
André Schaller
lastname@seceng.informatik.
tu-darmstadt.de
TU Darmstadt, CYSEC, Germany

Maksim Kolinichenko
m.kolinichenko@gmail.com
Unaffiliated

Stefan Katzenbeisser
katzenbeisser@seceng.informatik.
tu-darmstadt.de
TU Darmstadt, CYSEC, Germany

ABSTRACT

In this paper we examine the use of covert channels based on CPU load in order to achieve persistent user identification through browser sessions. In particular, we demonstrate that an HTML5 video, a GIF image, or CSS animations on a webpage can be used to force the CPU to produce a sequence of distinct load levels, even without JavaScript or any client-side code.

These load levels can be then captured either by another browsing session, running on the same or a different browser in parallel to the browsing session we want to identify, or by a malicious app installed on the device. To get a good estimation of the CPU load caused by the target session, the receiver can observe system statistics about CPU activity (app), or constantly measure time it takes to execute a known code segment (app and browser). Furthermore, for mobile devices we propose a sensor-based approach to estimate the CPU load, based on exploiting disturbances of the magnetometer sensor data caused by the high CPU activity.

Captured loads can be decoded and translated into an identifying bit string, which is transmitted back to the attacker. Due to the way loads are produced, these methods are applicable even in highly restrictive browsers, such as the Tor Browser, and run unnoticeably to the end user. Therefore, unlike existing ways of web tracking, our methods circumvent most of the existing countermeasures, as they store the identifying information outside the browsing session being targeted.

Finally, we also thoroughly evaluate and assess each presented method of generating and receiving the signal, and provide an overview of potential countermeasures.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Browser security**; **Mobile and wireless security**; *Software security engineering*; *Web application security*;

KEYWORDS

web tracking, user identification, covert channels, CPU, JavaScript, HTML5, CSS, Tor, electromagnetic emanations

ACM Reference Format:
Nikolay Matyunin, Nikolaos A. Anagnostopoulos, Spyros Boukoros, Markus Heinrich, André Schaller, Maksim Kolinichenko, and Stefan Katzenbeisser. 2018. Tracking Private Browsing Sessions using CPU-based Covert Channels. In *WiSec '18: Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, June 18–20, 2018, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3212480.3212489

1 INTRODUCTION

In our modern interconnected world, users tend to share an ever increasing amount of information on the web. In order to provide personalized services, modern websites rely on identification mechanisms, which associate a particular web session with a specific user. The most common way of identifying users online are cookies, unique values stored in the browser and sent to the server within every web request to associate a session with a concrete user. Interfaces such as Local Storage or Indexed Database can also be used to store the identifier in the browser. Alternatively, browser fingerprinting, which is achieved by combining unique information about user's settings from multiple JavaScript APIs, can be used to identify the user in a probabilistic manner. Finally, the network layer can be used in order to identify users based on their IP address.

Such identifiers may also be used to collect additional data about users, in order to improve Quality of Service, or provide targeted advertisements. Moreover, modern web applications often include resources from third-party sources, e.g., embedded video content, sharing buttons for social networks, advertisement and analytics components, etc. By using the described identification mechanisms, third-party providers can track user activity on hundreds of websites, and link this information together. This introduces a serious privacy threat, since sensitive information about one's private life may be exposed in this way, including personal interests, location data, or even information about one's health, beliefs and sexuality.

Many different countermeasures have been proposed in order to mitigate such privacy threats. Web browsers have added a setting to block third-party cookies, while plugins such as Ghostery and DoNotTrackMe have been developed to block third-party trackers. To defend on the network level, a VPN or a Proxy are ways to mask one's IP address, while anonymity networks such as Tor and I2P

WiSec '18, June 18–20, 2018, Stockholm, Sweden

ACM ISBN 978-1-4503-5731-9/18/06…$15.00
https://doi.org/10.1145/3212480.3212489

can completely anonymize the traffic flow. Users may also manually delete their cookies after each session, or even use multiple browsers to prevent data sharing between sessions. Modern browsers have introduced an "incognito mode", a browsing mode which guarantees to remove all user traces stored in the browser after closing the window. Finally, the Tor Browser, a special bundle based on Firefox, has been developed with a special focus on user's privacy. It is configured to connect only through the Tor network and block sensitive APIs [25].

One way to circumvent above countermeasures, would be to exfiltrate an identifying token out of the browser. This cannot be easily performed, though, because web pages are isolated from external processes and cannot access the file system. Thus, the need for side channels arises, which allow to bypass such restrictions.

To this end, in this work we study CPU load covert channels in modern browsers. More precisely, we propose to transmit an identification token received from a server in a private browsing session, protected from traditional tracking methods, to another browsing session, which does have access to long-term identifying information about the user, or to a malicious application installed on the same device. This way, the received token can be linked to the victim. The proposed method allows to transmit the tracking token between private and non-private sessions opened in two tabs of the same browser, between two instances of the same browser (e.g., between an "incognito" and a normal browsing modes), or even between different browsers. The transmission is performed by encoding the token into distinct loads caused by the CPU. We demonstrate that basic web components, such as HTML5 videos, GIF images, or CSS animations, can be used to covertly produce controllable CPU loads, without need of JavaScript or plugins.

A malicious app, running in a background on victim's device, can collect the encoded token by constantly accessing system statistics about CPU activity (e.g., the `/proc/stat` file on Linux platforms), in order to get a precise estimate of the CPU loads caused by the victim browser session. However, as this information is a known source of side-channel leakage [18, 30], access to `/proc/stat` is forbidden for mobile applications in Android 8 (released in August 2017) [16], and also not available from web pages.

To overcome these restrictions, we propose two alternative methods to receive the token. First, the receiving (non-private) session or application can continuously time the execution of some JavaScript code, and thus can estimate the amount of CPU load caused by the target browser session. Second, magnetometer sensor measurements can be analyzed on mobile devices. We show that electromagnetic activity of the CPU causes noticeable disturbances in the sensor measurements on smartphones, leaking information regarding background CPU activity. The magnetometer data is available from within mobile applications (through native system APIs), as well as from web pages, using the recently introduced Generic Sensor API [29]. The proposed receiving methods do not rely on a very precise timer, unlike receivers for memory- and cache-based covert channels (e.g., [24, 28]) and therefore are not mitigated by the latest countermeasures against Spectre [14] attacks in browsers [20, 36].

Therefore, unlike the traditional tracking approaches, the proposed solution makes tracking applicable to very restrictive browser configurations (e.g., the Tor Browser), but requires an additional, less restricted browser session to be opened at the same time, or a malicious application installed on the same device. We show that both timing-based and sensor-based approaches provide sufficient information about the CPU load, and allow to achieve identification of the victim browser sessions using a CPU-based covert channel.

Contributions

Our contributions are threefold:

- We thoroughly examine and evaluate in a systematic way the usage of CPU-load covert channels to exfiltrate a tracking ID from a private browser session. Our method works even across different browsers, and is applicable to both desktops and mobile platforms.
- We present and compare four distinct ways to force the CPU load to follow a specific pattern. Apart from the traditional way of executing CPU-intense client-side code, e.g., using JavaScript, we show that HTML5 videos, GIF images, or CSS animations can be used for this purpose. The latter two pose a significant risk, as displaying GIF animations or the execution of CSS in a web browser, unlike the execution of Javascript, has up to now been considered as completely safe, and is enabled even in restrictive browser configurations, such as the Tor browser.
- We propose a novel approach to estimate CPU loads on mobile devices, thereby receiving tracking information by analyzing magnetometer sensor data. To the best of our knowledge, our work is the first to demonstrate a privacy implication of the new Generic Sensor API in browsers.

2 RELATED WORK

2.1 Tracking web browser users

The most ubiquitous way of web tracking is storing tracking identifiers in-browser, typically as cookies, or in other storage facilities available in the browser, such as the HTML5 Storage, Indexed Database APIs, etc. Furthermore, researchers presented the concept of evercookies [4, 13], where a tracking identifier is saved in several places at once. If the user removes it from one of the storage sites (e.g., by clearing the cookies), a script automatically "respawns" the evercookie from the other storage places. Usage of this technique has been observed on popular websites [1, 4]. To prevent it, browser vendors introduced a private browsing mode, which guarantees to remove all user traces after closing the browser window. In this work, we propose to exfiltrate tracking identifiers out of the browser using covert channels, in order to circumvent such protection.

Another approach for web-tracking is browser fingerprinting, where users are identified by unique properties of their browser, system environment or hardware [35]. Eckersley [10] found that more than 80% of fingerprints are unique for a sample of around 450,000 desktop browsers, and Laperdrix et al. [15] demonstrated the same effectiveness of fingerprints for mobile devices. Additional features were proposed for mobile fingerprinting, such as imperfections in sensor calibrations [6, 9], or unique audio hardware characteristics [8]. Despite its high effectiveness, fingerprinting remains only a probabilistic solution, while the use of tracking identifiers allows to unambiguously identify each user. Moreover, most of the commonly used fingerprinting techniques rely on JavaScript or plugins

to collect the identifying features [35], and browser vendors develop defenses against common fingerprinting methods [25].

2.2 CPU-based covert channels

Covert channels based on CPU activity have typically been applied to virtualized environments, where virtual machines share the same physical machine. In particular, Okamura and Oyama [23] presented a load-based covert communication system between virtual machines on the Xen hypervisor. By measuring the execution time of small pieces of known program code, researchers achieved a bitrate of 0.49bps with 100% accuracy. Rushanan et al. [26] proposed to use the WebWorker API to create CPU loads in background web pages and observe them using native applications. Recently, White [37] described in his blog how this approach can be applied to the communication between two browsers, by measuring execution times of JavaScript fragments. In this work, we further develop this idea, showing that a covert channel can be established even without execution of JavaScript code on the transmitter side. We additionally propose two ways of receiving the signal, and evaluate the solution on different devices.

Oren et al. [24] demonstrated the feasibility of cache attacks using JavaScript, and were able to recover information about websites visited in private browsing mode. Schwarz et al. [28] investigated indirect ways to acquire precise timing in the browser, and implemented an in-browser receiver for a DRAM-based covert channel. Although such covert channels may also be potentially employed to exfiltrate data from private browsing sessions, it may prove difficult to successfully control the memory when JavaScript is turned off; furthermore, their receiver relies on very precise timing information, blocked in the latest web browsers [20, 36].

Several channels have been proposed to establish covert communication between sandboxed mobile applications, based either on accessing common APIs and system resources [7, 18], or of hardware components and during I/O operationssensors [2, 22, 27]. In particular, Marforio et al. [18] described several covert channels, including one based on exploiting CPU statistics over `/proc/stat`. In our work, we show that distinct CPU loads can be generated within a browser, even without JavaScript, and be used for tracking users, and apply both timing- and sensor-based approaches to receive loads.

Apart from covert channels within one device, covert communication between devices has also been examined. Specifically, Hasan et al. [11] investigated the ability of smartphone magnetometers to detect the signal emitted by a nearby located electromagnet, while Matyunin et al. [19] presented a complete covert channel between smartphones and laptops, demonstrating that laptops emanate electromagnetic (EM) signals during I/O operations, which are detectable by magnetometers at a distance. In this paper, we show that distinct EM signals can be generated and captured even on a smartphone itself, and apply the relevant covert channel scheme to the web-tracking use case.

3 ATTACK SCENARIO

In this section, we discuss the assumptions required to establish CPU-load covert channels in browsers, and present how to use these covert channels for the purpose of web user tracking.

3.1 Assumptions

For our attack, we consider a victim who uses a desktop machine or a smartphone, and interacts with the Internet through one of the most commonly-used web browsers, such as Google Chrome, Mozilla Firefox, or the Tor Browser. The victim visits an intrusive website, which aims to identify the user session and therefore track the victim. The intrusive website either fully belongs to the attacker, or contains components from an attacker-controlled server. The latter corresponds to a typical scenario when websites include third-party code from advertisement or analytics services, such as Google Analytics or Facebook Pixel. This way, third-party components from advertisement networks are present on thousands of websites, and any of these websites could carry the tracking code payload. As user tracking is also of commercial interest of advertisement companies, we consider this scenario as more scalable and practical.

Further, we assume that the intrusive website does not possess any information which uniquely identifies the user (e.g., personal or login information entered on the web page). Furthermore, we assume that the client does not allow the intrusive website to permanently store any tracking identifiers (e.g., by disallowing cookies, or by using private browsing mode), and prevents known fingerprinting techniques (e.g., by using the Tor Browser to hide the IP address, and even disabling JavaScript). In any case, we assume that traditional tracking mechanisms are not applicable for this web session, and refer to it as *target* or *private* session.

Additionally, we assume that the victim has another web page opened, containing attacker-controlled JavaScript code. Similarly to the private session, this web site can either belong to the attacker, or contain third-party inclusions under attacker's control. However, in this case we assume that the web page *does* have access to information which allows to uniquely identify the user, or store the tracking ID. Therefore, this session is assumed not to be as restrictive as the private session, and is referred as *receiving* or *non-private* session. More specifically, the private and non-private sessions are considered to be opened:

- in different tabs of the same browser instance (e.g., the user provides login information in only the non-private session, or specifically disallows tracking information for the private session);
- in different instances of the same browser (e.g., the user opens the private session in the private browsing mode, and the non-private session in a "normal" mode).
- in different browsers (e.g., the user opens the private session in the Tor Browser, but has the non-private session opened in another browser for convenience, since the Tor network is usually slower).

Alternatively, instead of having a non-private session, we could assume the attacker to have control over a background application running on the victim's machine. This application does not require any privileged access rights from the operating system, or additional permissions from the user. Therefore, the code can be hidden in any app that already requires Internet access and which the victim is likely to install on his device. This use case complies with existing examples of mobile applications which silently performed user tracking, and have been recently discovered in the wild [3]. Nevertheless, as this scenario requires an additional application

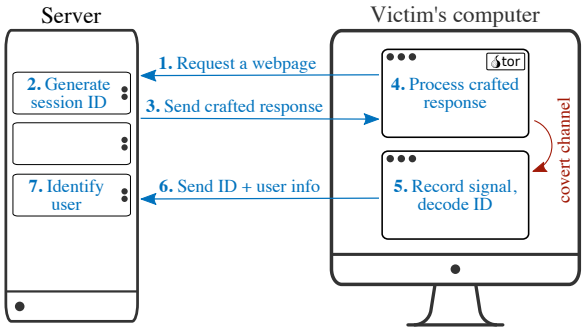

Figure 1: Overview of the general attack scenario utilizing the CPU-based covert channel.

installed, we consider it as less practical, but present evaluation for the sake of completeness. In case the receiver is a web browser, we will refer to this scenario as the *in-browser* scenario; in case the receiver is an additional application, we speak of an *in-app* scenario.

3.2 Attack scheme

Our attack scenario is illustrated in Figure 1. As a first step, a victim user visits a target private session **(1)**. For every request, the server generates a unique session ID **(2)** and a specific response for each session ID, and sends it to the client **(3)**. The response is crafted in such a way as to cause the client browser to produce distinct CPU loads, encoding the ID, when the response is processed **(4)**. We present four different ways that trigger such CPU loads and describe them in detail in the following sections.

Depending on the considered scenario, the information is decoded in different ways:

- In the in-browser scenario, the JavaScript code running in the non-private session estimates CPU loads and decodes the transmitted ID **(5)**. Then the decoded ID is linked to the available non-private session information. Depending on the browser configuration and the method of retrieving load information, this code can be run in background, or only in a foreground tab. We investigate the applicability of each method and browser configuration in Section 5.1.
- In the in-app scenario, the attacker-controlled application constantly records the system statistics regarding CPU activity in a background, captures the produced loads and decodes the ID **(5)**. Every ID decoded by the app is considered to belong to a specific user who may be identified by another permanently stored unique token, created for each instance of the recording application.

In both scenarios, the decoded ID is transmitted back to the server together with the aforementioned unique app or browser data **(6)**, which finally allows the server to identify the user by linking the session IDs to the same unique information **(7)**.

4 TECHNICAL DESIGN

In this section, we describe implementation details of our covert channel. First, we present our method of encoding data into CPU loads and several approaches in order to produce these loads. Subsequently, we describe and compare three different approaches of measuring these loads, and examine the decoding process.

4.1 Encoding and transmission

To encode a binary identifier into CPU loads, we apply on-off keying (OOK) modulation. We force a browser to produce intense CPU activity within a time frame of length t to encode a 1, and perform no activity to encode a 0. Additionally, the transmitted ID is prepended with a predefined binary sequence, in order to help the decoding algorithm to recognize the start of the transmission. The time periods for transmitting bits of this synchronization sequence and for actual ID bits are of different length, in order to avoid possible false positive detections when the synchronization sequence bits happen to appear in the ID itself. In our implementation, we used the 11-bit Barker sequence for synchronization, due to its low autocorrelation properties [5, 34], which facilitates its detection.

We implemented four different methods of producing CPU loads in the browser, using either (1) JavaScript loops, (2) HTML5 videos, (3) filtered GIF images, or (4) CSS animations. Figure 2 illustrates all proposed approaches to encode a binary string into CPU loads, which are explained in more detail in the following subsections.

Transmission: JavaScript. Our first approach to generate CPU loads is to execute CPU-intense JavaScript code in the browser. A loop that repeatedly checks if a time period has elapsed (*busy waiting*) results in potential high use of one logical CPU core during this time, corresponding to logical 1. In contrast, for time frames corresponding to the encoding of a logical 0, our implementation forces the target session to *passively* wait, using the `setTimeout` JavaScript function. The relevant JavaScript code, combining time frames of busy and passive waiting, is generated on the server's side for each ID, and then sent to the client for execution.

Furthermore, to increase the CPU load, we utilize the WebWorker API, which allows running scripts in separate background threads. To produce a high CPU load, busy waiting loops are concurrently run in a number of threads, equal to the amount of available logical processors, accessible through the `hardwareConcurrency` property. Therefore, the OS will potentially use all available logical cores to execute the JavaScript loop code. The actual CPU utilization depends on background activities and the OS scheduling mechanism.

Although this approach is an efficient way to utilize up to 100% of CPU time, the necessity of executing JavaScript in the target session makes it potentially less practical, as JavaScript may be disabled in restrictive browser configurations, e.g., in the Tor Browser. Therefore, we were motivated to find alternative ways to produce intensive CPU activity, which do not rely on the execution of scripts in the target session.

Transmission: HTML5 video. The second method we propose in order to create high CPU loads is rendering video with HTML5. Video decoders require more CPU power to process video frames with higher resolution. Moreover, the decoder requires slightly more CPU resources to decompress so-called I-frames, which contain a whole picture, than so-called B- and P-frames, which encode only differences between the preceding and the succeeding I-frames.

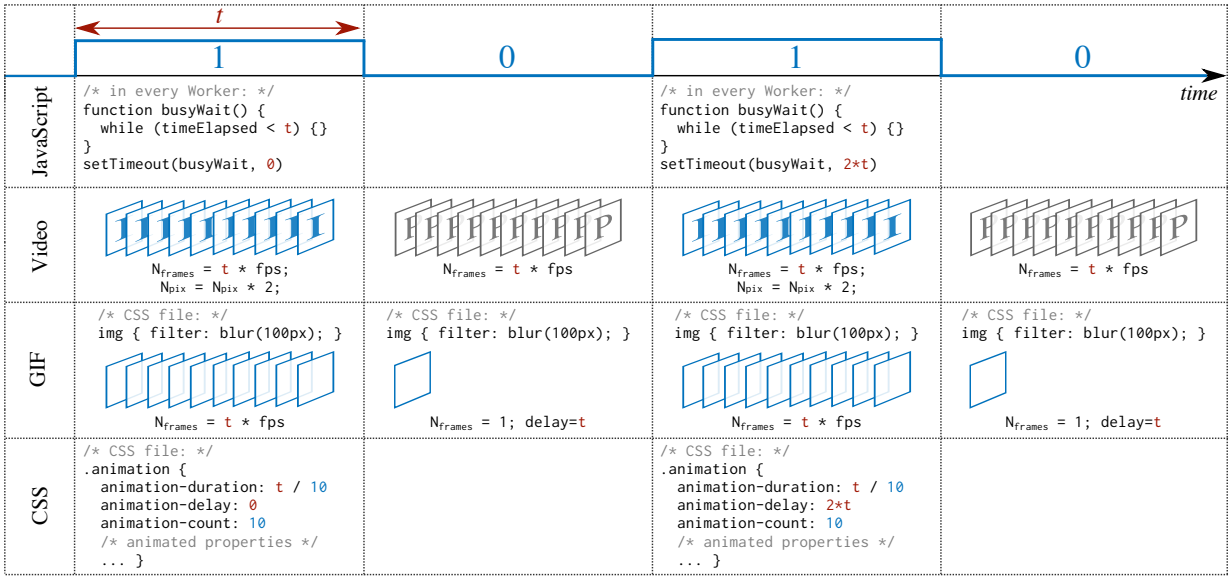

Figure 2: Illustration of four approaches to generate CPU loads: JavaScript, HTML5 video, GIF image, and CSS animations.

Therefore, to increase CPU activity within a time period, we can play a video where each frame has a twice higher resolution, and is encoded as an I-frame (*hard fragments*). In contrast, we can reduce CPU work required for playback within a time frame by playing a video consisting mostly of P-frames (*easy fragments*) with the original resolution. A video combining hard and easy fragments is created as a result, to produce distinct CPU loads.

As the target environment may not allow execution of JavaScript, we may not have the ability to programmatically start and stop video playback in the browser. Instead, the server creates a single video containing a sequence of hard and easy fragments, corresponding to the bits of the ID, to respectively increase or reduce CPU activity. The `autoplay` property of the `video` HTML5 tag containing our crafted video is then set, in order to start playback automatically after loading. Additionally, the `loop` property can be set to continuously repeat the playback in the target session.

The produced video element is set to have a fixed size corresponding to the original resolution. It may be directly shown to the user, as a seemingly benign component of the interface, since the aforementioned encoding process results only in barely noticeable quality differences between hard and easy fragments. Therefore, the encoding process is not evident and the user cannot easily notice that a transmission is taking place. Alternatively, the video object can be hidden from the user (e.g., by setting a negligible opacity, or setting the width and height of a video container to one pixel).

Transmission: GIF images. The third approach to produce CPU loads is based on applying CPU-intense CSS styles to GIF animations. The *filter* CSS property allows to apply Gaussian blur to web elements. For GIF animations, the filter has to be reapplied for every frame of the image. If the radius of the filter is high, the resulting rendering becomes computationally expensive, even for a very small original image (30x30px in our experiments). Furthermore, the GIF file format allows to set custom delays after individual frames of the animation. Therefore, a high CPU load can be generated by showing a blurred GIF animation with a maximum available frame-per-second (FPS) rate. In contrast, showing only a single frame with a delay of a time frame duration results in a small CPU load.

The resulting GIF animation can again be hidden by setting a negligible opacity, or be used as a seemingly static benign image, if all the frames of the GIF are identical. The *loop* property can be set to continiously repeat the load generation. GIF animations do not require JavaScript to be shown, and currently cannot be turned off in user-level settings, even in the Tor Browser, which makes it hard to prevent transmission using this approach.

Transmission: CSS animations. Our forth approach to produce CPU loads is based on using CSS animations, which make it possible to animate different CSS properties of custom web page elements (e.g., their size, opacity, color, etc.). Declaring CSS animations also do not require JavaScript in order to be declared and executed, and are recommended for usage by modern browser vendors due to their optimized performance [21].

Some of the animated properties, such as scale or rotation, are highly optimized in modern browsers [17]. Other properties, however, require recalculation of the page layout or redrawing of elements, and therefore cause intense CPU activity. We animated concurrently 10 properties, including element size, position, color and font, to produce distinct CPU loads. To cause even more CPU loads, we declared the animations for several identical objects at once, and repeat them several times within a time frame. We demonstrate an example of such animation declaration in Appendix A.

To transmit each bit set to the logical value 1 in the ID, the server declares a new animation, specifies its `animation-duration` property to the length of a single time frame, and sets its `animation-delay` property to start its playback at the corresponding time frame. Therefore, time frames related to the encoding of bits with logical value 0 result in passive waiting between animations. Finally, all

Table 1: Comparison of methods of measuring the CPU load.

	/proc/stat	Timing		Sensor	
Scenario	app only	app	browser	app	browser
Setup: desktops	+	+	+	−	−
Setup: mobiles	∘ (Android≤7)	+	+	+	+
Background execution	+	+	∘ (desktop only)	+	−
Sampling rate	50Hz	40Hz		50Hz	10Hz
CPU usage	≤ 5%	15–30%		≤ 5%	

"+": fully supported, "∘": partially supported, "-": not supported

the animations can be made invisible, by setting negligible opacity to all the animated elements.

4.2 Receiving the signal

The produced CPU loads are to be recorded by either a non-private session or a background application. In both cases, the receiving code constantly evaluates the CPU load at a given frequency. The obtained values are then analyzed by the decoding algorithm. We propose three approaches to record CPU loads: the straightforward method of reading `/proc/stat` data, as well as timing- and sensor-based methods, which indirectly estimate the CPU load. Table 1 provides an overview of implemented approaches.

Receiver: `/proc/stat`. If the attacker controls a background application running on a victim's device (the in-app scenario), the system statistics regarding CPU utilization can be directly accessed in order to record CPU loads produced by the target session. On Linux-based platforms, including the Android OS, (up to Android 7), this can be achieved by accessing the `/proc/stat` file, which provides information regarding the amount of ticks (clock cycles) that the CPU has spent performing different kinds of work. By regularly reading the `/proc/stat` file, an estimate of the CPU load can be computed as the ratio of ticks spent on non-idle activities, to all clock ticks within the interval. Therefore, the resulting recorded signal contains discrete-time values ranging from 0 to 1, where 1 represents 100% utilization of CPU.

We have implemented this method for Linux, MacOS and Android platforms. The recording application can run completely in the background, does not require privileged access or specific permissions from the user, and consumes a small amount of system resources to perform constant recording. Therefore, the recording is hard to detect, and can be a part of a seemingly benign application. The recording sampling rate is limited by the update interval for `/proc/stat` information. In our implementation, we were able to record samples at a frequency of 50Hz, and constant recording required less than additional 5% of available CPU time on all tested devices.

It must be noted that this method is not applicable to the in-browser scenario, as there is no JavaScript API to directly access system statistics of CPU utilizations, or to access the filesystem. Moreover, starting from Android 8 the `/proc/stat` is no longer accessible from mobile applications. Nevertheless, we present evaluation results for this method, to be able to compare indirect ways to estimate CPU activity with precise system information provided in `/proc/stat`.

Receiver: timing-based. In order to estimate the CPU activity, the receiving browsing session or app constantly executes a small code segment and measures its execution time. The operating system has to allocate CPU resources to the target session when its execution requires intense CPU activity, and therefore the execution of the receiving session code takes more time. Therefore, we can use the execution time of a code segment run in the receiving session as estimate of the overall system CPU utilization within the recorded interval. For this reason, the resulting recorded signal consists of discrete-time values of the execution times of the code fragment.

This approach can be run from a background application, as well as from a background non-private tab having the receiving session opened. On mobile devices, browsers are prevented from executing JavaScript in background tabs, in order to reduce power consumption. Therefore, on mobile devices the in-browser scenario is limited to the use case when both private and non-private sessions are opened side by side in the foreground.

The recording code in the receiving session has to consume a significant amount of CPU time, in order to create a competition for CPU resources between the two sessions. Moreover, in order to capture the CPU activity performed by the target session, the recording code needs to be run on several available physical threads. In our implementation, we executed a recording with a sampling rate of up to 40Hz, running counting loops in multiple threads, consuming 15–30% of the overall CPU time on tested devices.

The optimal amount of threads to be used, as well as the number of iterations to be executed in these threads, depends on a particular hardware configuration. Therefore, to choose best parameters, the receiving session can perform a calibration phase right after being opened. More specifically, the receiving session can initiate a transmission of a predefined sequence using the JavaScript-based method described above 4.1, and at the same time execute the receiving code fragment with different parameters, measuring the resulting execution time difference between intervals corresponding to bits 1 and 0. The number of threads and iterations which results in a higher difference, is then used for continuous decoding. In our implementation, we probe a set of 16 different parameters, and the resulting calibration phase can be finished within 8 seconds.

Interestingly, for some configurations we observed the opposite effect: the execution time of the receiving session becomes noticeably *smaller* when CPU-intensive code is executed by the target session. This effect occurs due to CPU throttling: when applications require only a small amount of CPU resources, the system throttles the performance to save power; when the target session triggers an intense CPU load, the system increases its performance, and the code in the receiving session executes faster. Our decoding implementation allows us to address this case by additionally checking the cross-correlation with the inverse synchronization sequence.

Receiver: sensor-based. Finally, we propose a sensor-based approach to estimate the CPU load on mobile devices. Modern smartphones are equipped with magnetometer sensors, which measure the amplitude of the ambient magnetic field along the three axes. This data is normally used to implement a digital compass. However, it has been shown [19] that the sensor data of a magnetometer can be disturbed by the electromagnetic signals emitted by a nearby laptop's CPU or hard drive. We discovered that the magnetometer

Table 2: Applicability of the four approaches proposed to generate CPU loads.

	JavaScript	HTML5 video	GIF	CSS animations
Browser applicability	all tested browsers			
Background tab execution	fully supported [a]	fully supported	not supported	not supported
Tor Browser security level that permits execution	low (default) level	low (default) level [b]	all levels	all levels
Prevention ways in browsers	disable JavaScript	disable automatic playback	impossible without plugins	impossible without plugins
Ways of detection	CPU monitoring, JavaScript code analysis	CPU & network monitoring, HTML5 video file analysis	CPU monitoring, CSS and GIF file analysis	CPU monitoring, CSS source analysis
Traffic overhead	<1kB	≈3Mb for 30s video	10–100kB	<2kB
Transmission speed (with BER ≈ 10%)	*/proc/stat*: 20–30bit/s timing: 10–20bit/s sensor: 8–12bit/s	*/proc/stat*: 3–4bit/s timing: 1–3bit/s sensor: 1–3bit/s	*/proc/stat*: 7–12bit/s timing: 5–8bit/s sensor: 5–8bit/s	*/proc/stat*: 13–15bit/s timing: 8–12bit/s sensor: 5–8bit/s

[a] JavaScript can be throttled or even blocked in background tabs in mobile browsers

[b] On medium and high security levels, video playback can be triggered by the user.

on a smartphone can be disturbed even by the EM signal emitted by its own CPU. Moreover, the higher the CPU activity is, the stronger the relevant EM emissions are, and, therefore, also the higher the resulting disturbance is. Thus, by constantly observing the magnetometer data, one can indirectly estimate the produced CPU loads.

Sensor data can be accessed in Android applications by using the Sensor API. Moreover, a new Generic Sensor API [29], introduced in Google Chrome 63[1], allows to access magnetometer data from web pages. Therefore, this method is applicable to both in-app and in-browser scenarios. In order to convert the three-dimentional magnetometer data into discrete-time values, we follow the solution proposed in [19], by applying Principal Component Analysis [12] to the data and choosing the first component as the result, as it represents the direction with the biggest data disturbance.

Although the Generic Sensor API allows to access sensor data only from foreground tabs, and limits the sampling rate to 10Hz, this method provides a good estimate of the CPU load without consuming many resources, unlike the timing-based approach. In our in-browser and in-app implementations, we were able to record magnetometer data using less than 5% of available CPU time.

4.3 Decoding

The decoding process is identical for three recording scenarios. First, we calculate the cross-correlation between the recorded signal and the predetermined synchronization sequence, both resampled to have the same sampling rate. A time point corresponding to a high peak in the cross-correlation is considered as the start of a transmission.

We finally start decoding the recorded signal from the point where it matches the predetermined synchronization sequence. Then, using the synchronization sequence, we calculate the average value of all measurements within time frames corresponding to a logical 0, and, similarly, the average value of all measurements corresponding to a logical 1. Subsequently, we define the mean of these two averages as a threshold. Afterwards, the transmitted ID is decoded bit by bit, by comparing this threshold with the average of measurements within a time frame corresponding to a bit of the ID. If this average is above the threshold, the bit is decoded as 1; otherwise, the bit is decoded as 0.

5 EVALUATION

In this section, we evaluate the presented covert channel. First, we compare the applicability of the four proposed methods of CPU load generation to different browser setups. Then, we evaluate each solution on different hardware and software configurations, by measuring the Signal to Noise Ratio (SNR). Afterwards, we determine the achieved transmission speed for all proposed methods of transmission and reception of the signal. Finally, we evaluate the robustness of the signal in the presence of background CPU noise.

5.1 Applicability

In this section, we investigate browser configurations applicable for each of the four approaches proposed to generate CPU loads, specifically focusing on their applicability to the Tor Browser. Additionally, we analyzed available ways in browsers to prevent the success of these methods, described how the transmission can be detected, and measured the traffic overhead introduced by each approach. Table 2 summarizes the results. For convenience, we also included in the table the average achieved bitrates for all three recording approaches, discussed in more detail in Section 5.3.

We confirmed that all four approaches work in popular desktop and mobile browsers. The default configurations of all tested browsers[2] allow execution of JavaScript, as well as Web Workers API support, playback of HTML5 videos, showing filtered GIF animations, and execution of CSS animations. To prevent successful generation and transmission of the signal, some settings need to be changed. In particular, JavaScript and automatic video playback can be manually disabled in browsers. However, as mentioned before, the generated video can be presented to the user as a seemingly benign component of the interface. In the Tor browser, access to sensitive content is regulated through the so-called "Security slider" with three security levels, which represent a trade-off between usability and security, by disabling several components at each level [25]. All four approaches to create CPU loads work at a default (Low) security level. At the Medium and High levels, untrusted

[1] Currently an experimental feature, available to developers as an origin trial upon request. To be released in Google Chrome 67 in May 2018.

[2] Tested browsers: Chrome 63.0 (desktop and mobile versions), Firefox 57.0 (desktop and mobile versions), Tor Browser 7.0.

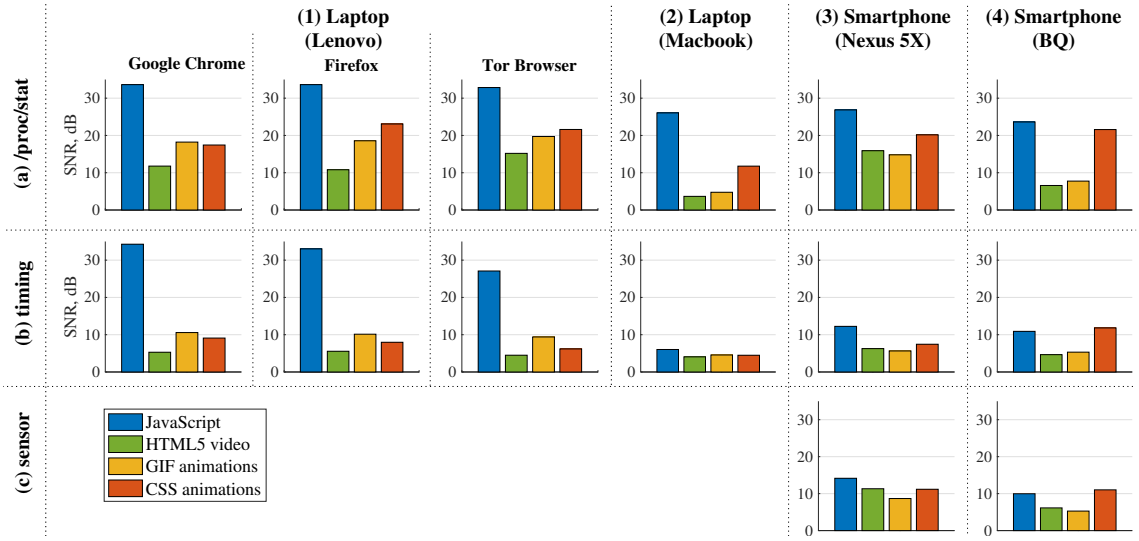

Figure 3: SNR levels recorded for two laptops (1,2) and two smartphones (3,4), received using `/proc/stat` (a), timing (b), and sensor (c) approaches. For the first laptop, recording are made in three different browsers (1.1, 1.2, 1.3).

JavaScript and video autoplay are disabled. However, the GIF and CSS animations approaches work even at the highest security level.

Furthermore, GIF and CSS animations cannot be disabled at all using the settings of all tested browsers, including the Tor Browser. To prevent the transmission, users have to install an extension which allows to block GIF animations or to override rules declaring CSS animations. The potential shortcomings of the GIF and CSS animations approaches are that a target web page has to be opened in a foreground tab and the browser window must not be minimized. In contrast, JavaScript-based and video-based generation methods work even in minimized browsers and background tabs.

A straightforward way to detect covert transmission in all four cases is to monitor the CPU activity. Additionally, a user can manually observe the sources of the web page and analyze the executed JavaScript code, or the declaration of CSS animations, or discover the hidden video or GIF element. The video approach also introduces a traffic overhead which may be noticed at the network level. However, even if the components of the attack are discovered, their purpose may remain unclear to the user. In particular, additional file content analysis is required to recognize hard and easy fragments in the video file, or a significantly different amount of frames in the GIF animation. Moreover, both JavaScript code and CSS animation declarations can be obfuscated to hamper the analysis.

In summary, we believe that all four approaches are applicable to modern desktop and mobile browsers. The methods based on GIF and CSS animations introduce a significant privacy risk for users, since they work even at the highest security level of the Tor Browser, and cannot be disabled in browser settings.

5.2 Signal strength

In this section we measure and compare the strength of the signal for each of the transmitting and recoding methods described in Section 4. For this purpose, we measured the Signal to Noise Ratio (SNR) of the produced signals, i.e., the ratio of the difference between the two signal levels (corresponding to high and low CPU loads), to the standard deviation of measurements at the base CPU load level, when no loads are induced by the target session. For these measurements, we alternated high and low CPU loads lasting for 2 seconds, and repeated this pattern 50 times. We used a Lenovo X1 laptop with 2 CPU cores running Ubuntu 16.04, a MacBook Pro with 4 CPU cores running MacOS 10.12, as well as two smartphones: a Nexus 5X and a BQ Aquaris X5, both running Android 7.1.

Figure 3 demonstrates the SNR measured on a Lenovo laptop in one of the three tested browsers (1.1, 1.2, 1.3), on a MacBook Pro laptop (2), on a Nexus 5X (3), and on a BQ Aquaris (4), recorded by accessing `/proc/stat` **(a)**, measuring timing response **(b)**, or analyzing magnetometer disturbances **(c)** on the smartphone. A separate Google Chrome instance process was used for recording using timing and sensor-based approaches. We checked that timing recordings made in Mozilla Firefox produce similar results with slightly lower SNR (≈20%) due to a higher "noise" level. We also checked that recording magnetometer data using the native application results in the same SNR (within ≈20%).

One can notice that all four approaches generate a distinct signal in the tested browsers and on all tested devices. The JavaScript method directly occupies all available CPU cores, which results in the highest SNR in all recording scenarios. GIF animations, CSS animations and HTML5 videos request only a part of the available CPU resources, and therefore, the signal is lower in comparison to JavaScript-based generation. Although the SNR levels observed for the video-based method appear to be comparably low, we were able to distinguish the signal levels and correctly decode transmitted IDs, even for the minimum SNR value of ≈4 dB observed for this method.

One can also notice that the timing-based approach, which relies on competition between execution threads, results in a lower SNR for non-JavaScript approaches than `/proc/stat` approach, which

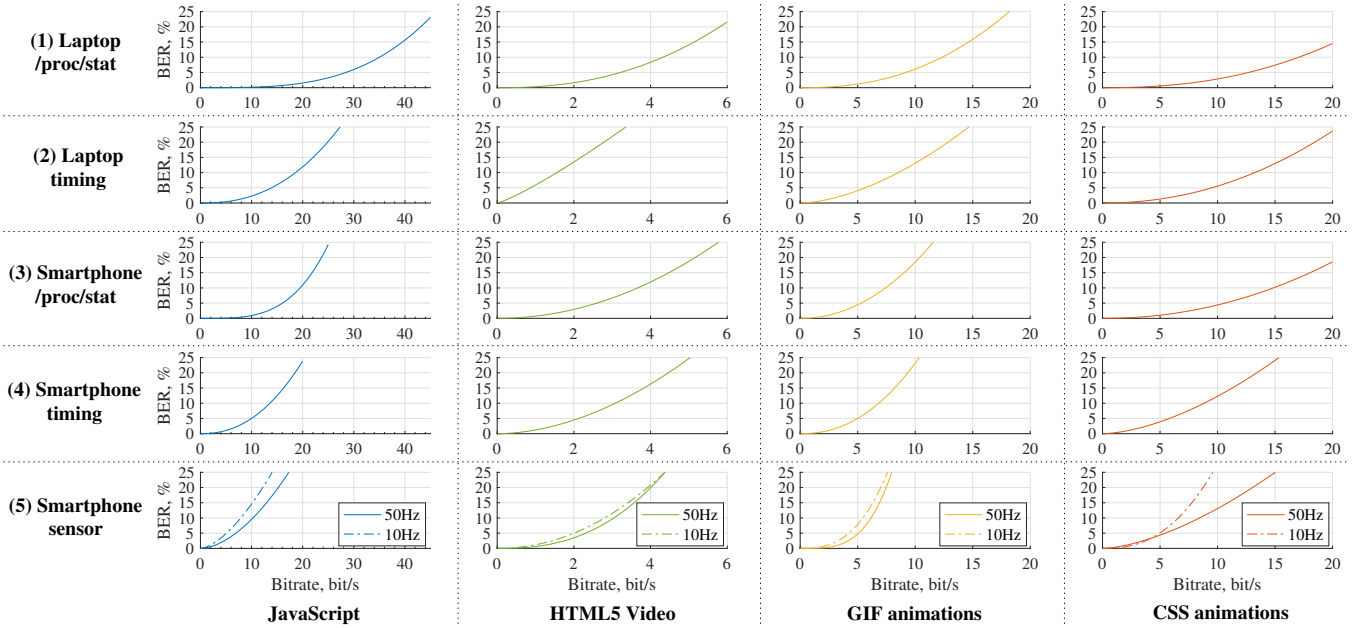

Figure 4: Bit Error Rate (BER) corresponding to different bitrate for each encoding method on a laptop using the `/proc/stat` receiver (1), a laptop using the timing-based receiver (2), a smartphone using the `/proc/stat` receiver (3), a smartphone using the timing-based receiver (4), and a smartphone using the in-app (solid) and in-browser (dashed) sensor-based receivers (5).

has access to cumulative CPU activity across all threads. Finally, one can notice that sensor-based approach generates a distinguishable signal on both smartphones, although requires much less resources than the timing approach.

As a result, we consider all of the approaches discussed as practical and potentially capable of successfully transmitting and receiving a signal under different hardware and software configurations.

5.3 Transmission bitrate

In this section, we evaluate the optimal speed of covert transmission using each generation method and three proposed receivers, when using the on-off modulation scheme described in Section 4.1.

For this purpose, we transmitted IDs through the covert channel with different bitrates, and calculated the decoding bit error rate (BER). We consider a BER of 10% as practically suitable, since in this case correct decoding can be achieved by using error-correcting codes with manageable overhead. For each bitrate we transmitted 50 IDs of a length of 30 bits, each prepended with a 11-bit synchronization sequence, which potentially allows to identify more than 1 billion unique sessions. We chose two representative hardware configurations: a laptop Lenovo X1 and a smartphone Nexus 5X, both using Google Chrome to open the target session.

Figure 4 demonstrates the dependency between BER and bitrate for all four methods of generating CPU loads, using the `/proc/stat`-based receiver (**1,3**) and the timing-based receiver (**2,4**) on both devices, as well as using the sensor-based receiver on the smartphone (**5**) in both in-app and in-browser implementations.

One can see that the JavaScript method achieves the best transmission speed, with up to 30 bit/s with <10% BER on a laptop when using the `/proc/stat`-based receiver. Since only the JavaScript approach allows to directly request all available CPU resources with precise timing, bitrates achieved for signals produced with CSS animations and GIF animations are comparably lower. Still, for CSS animations we achieved bitrates of 5–15bit/s, depending on the receiver. The signal produced by GIF animations is less stable on a smartphone, apparently due to a different implementation of blur filtering in mobile browsers. Finally, the lower SNR provided by the video-based method results in a noisy signal even at a comparably low transmission speed of 1–5 bit/s. Therefore, we consider this approach suitable only for short IDs, e.g., with length of 10 bits.

When using the timing-based approach, more errors appear at high bitrates for all signal generation methods, since the used decoding method relies on a less precise CPU load estimation mechanism. Similarly, the sensor-based receiver on the smartphone provides less precise and more noisy signal, but still achieves successful decoding at 5–8bit/s for transmission using CSS and GIF animations. The in-browser implementation of the sensor-based receiver (with a sampling rate limited to only 10Hz) remains stable for low speed, with additional errors appearing at high bitrates.

We can conclude that all proposed methods allow to covertly transmit identifiers within several seconds, and reliably receive them by reading the `/proc/stat` file, as well as indirectly estimating the CPU activity using the timing- and sensor-based approaches.

5.4 Robustness

In this section, we evaluate our covert channel in the presence of background CPU activity. For this purpose, we ran transmissions on a laptop simultaneously with synthetic stress tests using the `stress-ng` utility, involving 5, 10 and 20% of CPU time and utilizing all available threads, and measured the resulting BER. Furthermore,

Table 3: Robustness of the transmission: BER under different background activity.

	/proc/stat		Timing		Sensor	
	5bit/s	8bit/s	5bit/s	8bit/s	5bit/s	8bit/s
Laptop						
no noise	1%	2%	2%	4%	–	–
stress 5%	6%	9%	10%	12%	–	–
stress 10%	8%	11%	22%	26%	–	–
stress 20%	16%	23%	26%	31%	–	–
video 6%	1%	3%	8%	10%	–	–
video 22%	13%	14%	33%	36%	–	–
Smartphone						
no noise	1%	3%	4%	9%	3%	8%
video 6%	2%	6%	9%	15%	4%	10%
video 20%	8%	17%	22%	28%	8%	18%

as a real-life workload scenario, we also evaluated transmission in presence of videos played in background on the laptop and on the smartphone, consuming 6–22% of CPU time.

Based on the previous experiment, we chose CSS animations as the signal generation method, as it performed better among other methods, not utilizing JavaScript. Table 3 shows the resulting BERs for two fixed bitrates, using all three recording methods.

As one can see, the higher background activity, the more decoding errors appear, as any peak activity can be misinterpreted as transmission of bit 1 with the on-off modulation scheme applied. For this reason, video playback causes less errors than artificial stress activity (consuming similar total CPU time), as it results in "smoother" use of CPU. The */proc/stat* receiver, expectedly the most robust one, provides successful decoding with up to 10% stress noise. The timing-based approach is the most susceptible to background CPU activity, as additional running processes cause more system interruptions and context switches, which affect the used CPU estimation. The sensor-based approach, however, is comparably stable under low noise from the video playback, especially at low bitrates. Nevertheless, all the recording methods allow to successfully receive the signal under low CPU activity caused by playing videos in background, or stress activity under 5%.

As a result, we believe that all the approaches are applicable to typical usage scenarios (web browsing, video playback) with low background activity; under high noise, the decoding becomes unstable, and more robust encoding schemes may be necessary to improve quality of transmission.

6 COUNTERMEASURES

A number of potential countermeasures exist against the user tracking methods described in this paper. Nevertheless, the majority of these countermeasures would require modifications in the software or hardware of the target device, and may have significant performance drawbacks.

The most straightforward way to prevent the attack of the in-browser scenario has been proposed in [37]: limit the amount of available CPU resources for all browsers, e.g., by using the *cgroups* kernel feature on Linux platforms. For example, if each of the two browsers employed in the in-browser scenario is allowed to only use 25% of the CPU, the CPU loads induced by our code in the target session will not affect the execution time of the code in the receiving session. However, this countermeasure significantly reduces the performance of web browsers in general, and strongly contradicts the industry trend to increase the complexity of web applications. Moreover, correct decoding of the ID may still be possible if the code on the receiving session runs faster than usual when the target session's code is also running on the CPU, due to the throttling effects that we have described earlier in Section 4.2.

Another countermeasure in order to hinder usage of the CPU-based covert channel could be to randomize CPU activity, in order to balance and normalize the loads produced, and thus prevent the recognition of either the synchronization code preceding the identifier or even the correct decoding of the CPU loads to bits. However, in such a case, all applications would have to endure excessive overheads in their processing times and thus the performance of the overall system would also be significantly impeded.

Preventing the execution of JavaScript in background tabs on all platforms would limit the applicability of the covert channel, as both our JavaScript-based transmitter and the timing-based recorder would only work in foreground tabs. Currently, the background execution of JavaScript is limited in mobile browsers, and throttling of JavaScript events in background tabs is recently introduced for the desktop Chrome browser [31]. However, these limitations are not applied to code executed in separate threads using Web Worker API (as in our approach). The Chrome development team considers restricting background Web Worker execution at some time in 2018 [32], and we believe that similar measures must be taken for modern browsers. Nevertheless, the described HTML5 video approach would not be affected by this change, while GIF and CSS animations are already not executed in background tabs.

Additionally, obtaining precise timing information via the Performance API in JavaScript can be forbidden to limit the effectiveness of the in-browser decoding, at least for background tabs and in restrictive browser configurations, such as Tor Browser. However, we believe that decoding will still be possible with lower bitrates.

Unless CSS animations and GIF animations are disabled by default in the most restrictive mode of the Tor Browser, the related attack scenario is very difficult to prevent. The Tor browser development team already considered this idea to prevent timing information from being indirectly available through CSS Animations [33], and we strongly advocate this measure.

Finally, to prevent the direct way of accessing CPU statistics, access to the `/proc/stat` can also be disallowed for user-level applications on all platforms, as has been made for Android 8. Furthermore, even allowing access to the `/proc/stat` file only at a reduced frequency could potentially make it more difficult to measure CPU load correctly and thus hinder successful identification of the CPU-based transmission. Nevertheless, in this work we show that both timing- and sensor-based techniques can be used instead to indirectly estimate the CPU load.

Physical isolation (shielding) of the smartphone magnetometer from the CPU can mitigate the sensor-based recording. However, it would require hardware changes, and contradicts to the industry trend of making mobile devices thin and compact. Furthermore, access to magnetometers can be restricted in background applications, and/or require and explicit user permission.

We can therefore conclude that all methods described in this paper regarding the persistent identification of browser sessions through the usage of CPU-based covert channels remain completely feasible and effective at present.

7 CONCLUSION

In this work, we presented the use of CPU-based covert channels for the purpose of web user tracking. We showed that different web components, such as HTML5 videos, GIF animations, or CSS animations, can produce distinct CPU loads when executed in the browser. An identification token encoded into these loads can be effectively exfiltrated from a private browsing session to either another session, or to an app recording in the background. The resulting covert transmission can be initialized without any client-side code, which makes the solution applicable to very restrictive browser configurations.

To capture the produced loads, the receiver can observe system statistics about CPU activity. Moreover, we compared two approaches to indirectly estimate the CPU activity: running a known segment of code in several threads and observing execution time differences caused by resource competition with the transmitter, and observing magnetometer disturbances on smartphones which correlate with electromagnetic activity caused by the CPU. Although the resulting bit error rate becomes too high in presence of intense background noise, both approaches allow to reliably receive the covert signal under low background activity.

We therefore conclude that the covert CPU-based channel we discussed in this paper poses a significant threat against the privacy of online users, even for those who use more restrictive web browsers, such as the Tor Browser.

REFERENCES

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 674–689. https://doi.org/10.1145/2660267.2660347

[2] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. 2014. A New Sensors-Based Covert Channel on Android. *The Scientific World Journal* 2014 (2014), 1–14. https://doi.org/10.1155/2014/969628

[3] D. Arp, E. Quiring, C. Wressnegger, and K. Rieck. 2017. Privacy Threats through Ultrasonic Side Channels on Mobile Devices. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 35–47. https://doi.org/10.1109/EuroSP.2017.33

[4] Mika Ayenson, Dietrich James Wambach, Ashkan Soltani, Nathan Good, and Chris Jay Hoofnagle. 2011. Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning. *SSRN Electronic Journal* (2011). https://doi.org/10.2139/ssrn.1898390

[5] RH Barker. 1953. Group synchronizing of binary digital systems. *Communication theory* (1953), 273–287.

[6] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. 2014. Mobile Device Identification via Sensor Fingerprinting. *ArXiv e-prints* (Aug. 2014). arXiv:cs.CR/1408.1416

[7] Swarup Chandra, Zhiqiang Lin, Ashish Kundu, and Latifur Khan. 2015. Towards a Systematic Study of the Covert Channel Attacks in Smartphones. In *International Conference on Security and Privacy in Communication Networks*, Jing Tian, Jiwu Jing, and Mudhakar Srivatsa (Eds.). Springer International Publishing, Cham, 427–435. https://doi.org/10.1007/978-3-319-23829-6_29

[8] Anupam Das, Nikita Borisov, and Matthew Caesar. 2014. Do You Hear What I Hear?: Fingerprinting Smart Devices Through Embedded Acoustic Components. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/2660267.2660325

[9] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *Proceedings of the 2014 Network and Distributed System Security Symposium*. Internet Society. https://doi.org/10.14722/ndss.2014.23059

[10] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *Privacy Enhancing Technologies*, Mikhail J. Atallah and Nicholas J. Hopper (Eds.). Springer-Verlag, Berlin, Heidelberg, 1–18. https://doi.org/10.1007/978-3-642-14527-8_1

[11] Ragib Hasan, Nitesh Saxena, Tzipora Haleviz, Shams Zawoad, and Dustin Rinehart. 2013. Sensing-enabled Channels for Hard-to-detect Command and Control of Mobile Devices. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/2484313.2484373

[12] H. Hotelling. 1933. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology* 24, 6 (1933), 417–441. https://doi.org/10.1037/h0071325

[13] Samy Kamkar. 2010. Evercookie: virtually irrevocable persistent cookies. Retrieved 01.02.2018 from https://samy.pl/evercookie/

[14] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:cs.CR/1801.01203

[15] P. Laperdrix, W. Rudametkin, and B. Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. 878–894. https://doi.org/10.1109/SP.2016.57

[16] Paul Lawrence. 2017. Seccomp filter in Android O. https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html [Accessed: 01.02.2018].

[17] Paul Lewis and Sam Thorogood. 2017. Animations and Performance. Retrieved 01.02.2018 from https://developers.google.com/web/fundamentals/design-and-ui/animations/animations-and-performance

[18] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. 2012. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2420950.2420958

[19] N. Matyunin, J. Szefer, S. Biedermann, and S. Katzenbeisser. 2016. Covert channels using mobile device's magnetic field sensors. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 525–532. https://doi.org/10.1109/ASPDAC.2016.7428065

[20] Microsoft Edge Team. 2018. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. Retrieved 01.02.2018 from https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/#pfB3kzxgGMzILirt.97

[21] Mozilla Firefox development team. 2016. CSS and JavaScript animation performance. Retrieved 01.02.2018 from https://developer.mozilla.org/en-US/Apps/Fundamentals/Performance/CSS_JavaScript_animation_performance

[22] Ed Novak, Yutao Tang, Zijiang Hao, Qun Li, and Yifan Zhang. 2015. Physical Media Covert Channels on Smart Mobile Devices. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 367–378. https://doi.org/10.1145/2750858.2804253

[23] Keisuke Okamura and Yoshihiro Oyama. 2010. Load-based Covert Channels Between Xen Virtual Machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. ACM, New York, NY, USA, 173–180. https://doi.org/10.1145/1774088.1774125

[24] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1406–1418. https://doi.org/10.1145/2810103.2813708

[25] Mike Perry, Erinn Clark, and Steven Murdoch. 2015. The design and implementation of the Tor browser [draft]. Retrieved 01.02.2018 from https://www.torproject.org/projects/torbrowser/design/

[26] Michael Rushanan, David Russell, and Aviel D. Rubin. 2016. MalloryWorker: Stealthy Computation and Covert Channels Using Web Workers. In *Security and Trust Management*, Gilles Barthe, Evangelos Markatos, and Pierangela Samarati (Eds.). Springer International Publishing, Cham, 196–211. https://doi.org/10.1007/978-3-319-46598-2_14

[27] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.. In *Proceedings of the 2011 Network and Distributed System Security Symposium*. Internet Society, 17–33.

[28] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*, Aggelos Kiayias (Ed.). Springer International Publishing, Cham, 247–267. https://doi.org/10.1007/978-3-319-70972-7_13

[29] Shalamov,Alexander and Pozdnyakov, Mikhail. 2017. Sensors For The Web! Retrieved 01.02.2018 from https://developers.google.com/web/updates/2017/09/sensors-for-the-web

[30] Laurent Simon, Wenduan Xu, and Ross Anderson. 2016. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards.

Proceedings on Privacy Enhancing Technologies 2016, 3 (jan 2016). https://doi.org/10.1515/popets-2016-0020

[31] Alexander Timin. 2017. Chrome Platform Status: Intervention: Throttle expensive background timers. Retrieved 01.02.2018 from https://www.chromestatus.com/feature/6172836527865856

[32] Alexander Timin. 2017. Reducing power consumption for background tabs. Retrieved 01.02.2018 from https://blog.chromium.org/2017/03/reducing-power-consumption-for.html

[33] Tor browser bundle team. 2016. Tor bug tracker, 18273: CSS animations provide high resolution timer. Retrieved 01.02.2018 from https://trac.torproject.org/projects/tor/ticket/16110

[34] R. Turyn and J. Storer. 1961. On binary sequences. *Proc. Amer. Math. Soc.* 12, 3 (mar 1961), 394–394. https://doi.org/10.1090/s0002-9939-1961-0125026-2

[35] R. Upathilake, Y. Li, and A. Matrawy. 2015. A classification of web browser fingerprinting techniques. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2015.7266460

[36] Luke Wagner. 2018. Mitigations landing for new class of timing attack. Retrieved 01.02.2018 from https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/

[37] Ethan White. 2016. CPU Correlation Attacks. Retrieved 01.02.2018 from https://ethanwhite.xyz/cpu-correlation

A CSS ANIMATION DECLARATION EXAMPLE

The code on the right declares a CSS animation, which can be used to cause CPU loads in a browser (described in detail in Section 4).

```
/* declare an animation, can be applied to
    several elements at once */
div.animated {
  animation-name: cpu-intense-animation;
  animation-duration: 100ms;
  animation-delay: 2s;
  animation-count: 10;
}


/* declare animated properties, setting their
    initial (0%) and end (100%) values */
@keyframes cpu-intense-animation {
  0% {
    /* transforming */
    width: 20px;
    height: 20px;

    /* moving */
    left: 0px;
    top: 0px;

    /* repainting */
    background-color: red;
    border-color: white;

    /* transforming and repainting text */
    font-size: 1px;
    font-family: Arial;

    /* making the animated element invisible */
    opacity: 0;
  }

  100% {
    /* transforming */
    width: 100px;
    height: 1000px;

    /* moving */
    left: 100px;
    top: 100px;

    /* repainting */
    background-color: green;
    border-color: black;

    /* transforming and repainting text */
    font-size: 50px;
    font-family: Arial;

    /* making the animated element invisible */
    opacity: 0.0001;
  }}
```