Analysis of Methods for Background Execution in Modern Web Applications

Analyse von Verfahren für Hintergrundausführung in modernen Webanwendungen
Bachelor-Thesis von Yannick Reifschneider
Tag der Einreichung:

1. Gutachten: Nikolay Matyunin
2. Gutachten: Prof. Dr. Stefan Katzenbeisser

TECHNISCHE
UNIVERSITÄT
DARMSTADT

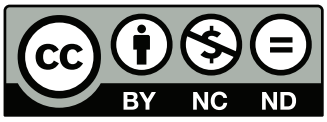
Fachbereich Informatik
Security Engineering

Analysis of Methods for Background Execution in Modern Web Applications
Analyse von Verfahren für Hintergrundausführung in modernen Webanwendungen

Vorgelegte Bachelor-Thesis von Yannick Reifschneider

1. Gutachten: Nikolay Matyunin
2. Gutachten: Prof. Dr. Stefan Katzenbeisser

Tag der Einreichung:



Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den September 8, 2019

_____

(Yannick Reifschneider)

Contents

1 Abstract

In this thesis we analyse modern browsers regarding their behaviour of JavaScript code exection in background tabs. We show that the energy conserving methods of desktop browsers can easily be circumvented to do arbitrary calculations for unlimited time while the web site is not user visible. With these findings we trace popular websites to see if they use similar methods to execute uninterrupted in the background.

2 Introduction

Web technology received rapid advances in the last years. Many websites not only deliver static information in form of text and images, but are complete highly dynamic applications which compete with traditional software products. For many of the standard business applications like an E-mail client, a word processor or even a bitmap graphic manipulation software there are alterantives[1][2][3], which are developed using web technologies and used via a modern web browser. The web browser has grown from a renderer of static information to the operating system of web applications. Some reasons for the rise of these web applications are, that browsers are getting more capable every year and the JavaScript performance is increasing drastically so that these applications are possible in the first place. Developing a web application instead of traditional software has also many benefits for the developer of said software. Web apps don't need installation, just a modern web browser, which comes preinstalled with any common operating system and mobile device. Visiting a web site to try out a piece of software is a much smaller hurdle to overcome then downloading, installing and running a piece of software. These reasons make the web a popular platform for developing new applications.

At the same time, the revenue model for many websites is advertising. Websites include third party JavaScript from ad networks, which display ads and track impressions. The included code can be controlled by the advertiser. For a malicous entity, it is therefore possible to include JavaScript on reputable websites just by paying for the ad impression. This method of distribution for malicous script is known as malvertising[18].

The web browser is a fundamental application on most personal computing devices and is trusted by it's users. Because many modern applications are implemented as web apps, web browser usage is ubiquitous in the usage of computing devices. As such, many users leave the web browser open during the whole time they use their device, often times with many website tabs open. The browser vendors are encouraged to conserve as much energy as possible to enable users on battery powered devices, such as mobile phones or notebooks which are not tethered to a power outlet, a longer usage time. On the other side, the browser should not break web applications which require regular CPU time to function correctly. Web apps, which need regular CPU time are sites, which notify the user about new content such as a news site or a social network, a web application which plays audio or video, or a web video conference application. All modern browsers limit the amount of CPU time a tab in the background can use and how often web sites can schedule new work. For maximum energy efficiency, all tabs in the background should eventually be halted completely and only woken up, when the page is moved to the foreground again. This is also the goal of browser vendors[1], but the execution of this goal is easy without breaking existing web applications or providing a workaround for some applications.

In this thesis we find out, if the throttling mechanisms employed by browser vendors can be circumvented, so that web sites which are not visible to the browser can gain code execution for arbitrary time. We compare the different throttling mechanisms between desktop and mobile browsers and we trace popular websites to see if the found cirumvention methods are actively used in the wild.

2.1 Motivation

To render a website, the browser has to execute all JavaScript code which was included in the web site. With browser default settings this happens without the users explicit consent. This means the web site authors can use the browser of it's visitors to execute arbitrary scripts. This very fundamental behaviour of the web is therefor also attracting malicous entities. There are numerous activities, which malicous entities could perform with access to a web site with high amount of daily visits.

- The attacker could inject a browser crypto-currency miner to convert electricity of the website visitors into crypto currency. The existance of crypto currencies such as Monero, which uses proof of work algorithm, which is inefficient to calculate on custom hardware, makes the browser a viable target for such mining. Existing implementations of Monero miners implemented in web assemlby are available for this attack.

- The browser could be used as a compute node in a botnet, which can be used for DDoS attacks or for cracking hashed passwords. Grossman et al.[5] show that these attacks can be used with standard web technologies and not exploiting any weaknesses.

- The computer of the visitor itself could be compromised by using exploits of the browser or the underlying system itself. Attacks which exploit hardware weaknesses such as Spectre[8], which could allow an attacker to read memory space of other browser tabs, could expose sensitive informations such as passwords or banking information of the uses. Rowhammer[7] attacks could even persistently infect the users system. These exploits usually need quite some time to be successful and could benefit from constant background execution in a background browser tab.

[1] https://www.google.com/gmail/about/

[2] https://products.office.com/en-us/free-office-online-for-the-web

[3] https://www.photopea.com/

2.2 Related works

- Papadopoulos et al.[13] analyses the if Cryptocurrency miners are a viable alternative to traditional ad serving as a revenue possibility for web sites. They come to the conclusion, that cryptocurrency mining is more profitable when a user stays longer then 5.3 minutes on the website. When permanent execution of a crypto miners are possible in background tabs, then this time could easily be achieved.

- Papadopoulus et al.[14] show that browsers can be infected with malicous service workers to act as a puppet in a botnet. Their method for persistence requires an implementation of a draft proposel HTML5 API, which is not implemented in mainstream browsers yet. They focus in their research on service workers, which are independent of browser tabs, but poses other limitations. Our research instead focuses on web sites which are not user visible, i.e. in an inactive tab.

- Measurements of existing popular websites is done for privacy related as well as security releated research. Engelhardt et al.[4] for example developed the OpenWPM framework for doing privacy measurements on million of websites. This framework is also used for ...

  Security related analysis of existing webpages are done for example: to measure the use of third party script inclusions[10], to assess the security claims provided by third party security seal providers[17] or to study malware distributed via ad networks[20].

- Pan et al. analyse the feasability to use the browser of website visitors for offloading large computing tasks[12]. They termed this usage of distributed data processing gray computing, because it can be done without the users explicit consent, for example while the users are watching video streams. Their research focus is the performance of web workers and the cost effectiveness in comparison to cloud computing offerings. Our research complements the findings of Pan, because they could allow grey computing even when the user is consuming other web sites.

3 Background

JavaScript is the only programming language supported by all modern web browsers to enhance web sites or web applocications. If you compare JavaScript to other more traditional programming languages like C, C++ or Java you see some major differences between them.

- JavaScript has no functions for I/O like writing to a file or opening a socket. It depends on a runtime environment like a browser to provide I/O functionality.
- JavaScript has no concept of parallelism like operating system threads or processes.
- JavaScript is an event driven language and has a built in event loop.

All of these differences stem from the fact, that JavaScript was designed as an addition to HTML for enhancing web applications and the interface to the browser internal workings, such as the DOM. Many of these design decisions are choosen, because JavaScript is run in the browser and therefore can be run on any website you visit. This poses many security implications, because you may not want to give a website you visit the power to execute arbitrary programms on your computer.

Also explain why webassembly does nothing special in regards to background execution. That is, because it allows seamleas interop between webasm and Javascript functions. Therefore it has to run with the same runtime guarantees as JavaScript. WebASM Threads are based on Webworkers with a shared memory pool. WebASM therefore behaves exactly like javascript functions and have to yield to the event loop to not block the browser.

3.1 The JavaScript execution model

The JavaScript runtime uses an event loop to schedule tasks for execution[3]. The browser or other JavaScript code can put tasks on the task queue. During a single run of the event loop, the runtime removes the first item of the task queue and executes it. When the event queue is empty, the run loop waits until an event is placed into the queue.

When a JavaScript task is running, it is guaranteed to not be interrupted until it is completed. This insures, that a variable cannot change from outside. This is in contrast to many other programming languages like Java or C, where another thread can mutate variables at any time. To guarantee this behaviour, the JavaScript runtime in the web browser has to block and wait until the task finishes, because the JavaScript task has access to the browser internal state and DOM. Once the task is finished, the runtime can perform other work, such as layout calculations, which mutates its internal state. Due to these properties JavaScript tasks should be as small as possible to ensure responsiveness of the browser. If a JavaScript task takes a long time to complete (i.e. it is executing an infinite loop) most browser show a warning to the user that a script is slowing down the website. The user then has the option to kill the task or wait for the task to finish.

3.2 Web workers

Web workers are a mechanism to perform long running uninterruptable calculations. Due to the runtime guarantees which were described in the last section, Web workers have a completely seperate execution context. They don't share variables or resources with the invoking context.

The web worker and main thread context can communicate by passing messages to each other, which are handled by their respective event loops. In contrast to the main thread, which can manipulate the DOM of the browser, the Web worker has limited functionality. This limitation allows the web worker to run uninterrupted for longer times, because it does not does not block browser events.

4 Analysis of different background execution methods

As we have shown in the motivation section of this thesis, there numerous incentives to execute code in the background. Browsers vendors in contrast are motivated to limit the energy impact of JavaScript code in background tabs as much as possible to prolong the battery life of the users device. To assess the behaviour of the browser throttling mechanisms, we developed a framework to compare different methods for achieving background code execution. The framework handles the benchmarking and visualisation of each method. If new HTML5 APIs are released, that allow for a new method of scheduling tasks, this method can be plugged into the framework to compare it against existing methods. The framework also allows easy reproducability of our analysis, to test whether the throttling mechanisms of browsers changed in new versions. In our analysis we evaluate the throttling mechanisms employed by the following desktop browsers:

- Google Chrome 76
- Mozilla Firefox 69
- Apple Safari 12.1.2

For mobile browsers we evaluate these browsers:

- Google Chrome for Android 76
- Firefox for Android
- iOS 12.4.1 Mobile Safari

On iOS we only analyse Mobile Safari, because Apple does not allow other browser engines in the Apple AppStore. Every other browser app has to use the system-provided webview to be in accordance with § 2.5.6 from Apple Review Guidelines[4].

On Android, we can differentiate between different browser engines.

4.1 Methodology

In the following sections we look at different methods for scheduling JavaScript code, when the tab is in the background. We describe how each browser behaves when this method is used to execute JavaScript code in the background and we use our measureing framework to compare the browser behaviour for each method.

The measuring framework takes one parameter, to tune the measuring. With this parameter we can define the workload in milliseconds. This time defines, how long a simulated task should perform uninterrupted work. When this work is performed on the JavaScript main thread then the browser is unresponsive for this amount of time, until the simulated workload finished and yields back to the main thread. Google Chrome advises to keep all JavaScript tasks to under 50ms, to be percieved as immediate for the user[9]. For our analysis, we measure each method with two workload times for each browser, once for the recommended 50 ms workload and once for a long 1000 ms workload. The measurement is started automatically when the website page is moved to the background and stopped when it is visible again. The framework uses the Page Visibility API [11] for determining the current page visibility.

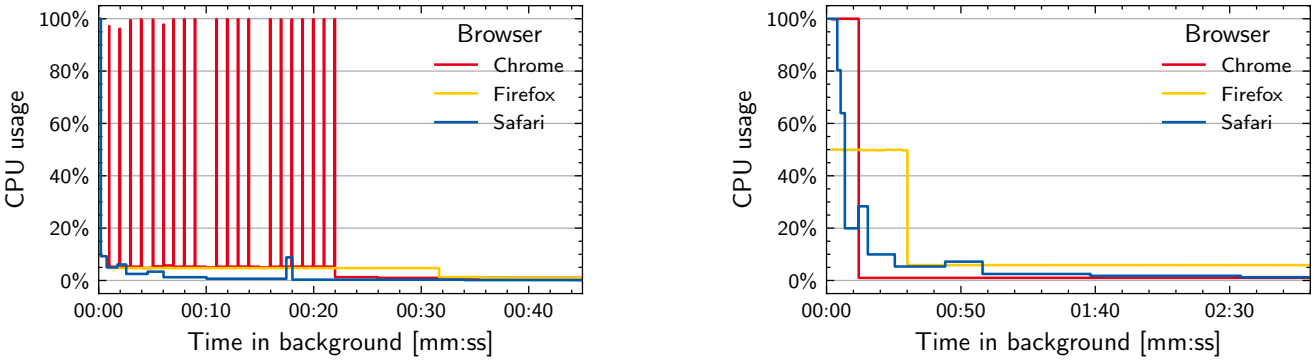
To test the browsers behaviour to background tasks, we developed a browser automation script using Selenium WebDriver [16]. WebDriver lets us automate different browsers using an API. In our automation script, we open our analysis page which embeds our measuring framework and starts the benchmark by opening a new empty browser tab. After a fixed amount of time, we close the empty browser tab again to make the analysis page visible again and therefor stopping the measuring. This procedure is repeated for every browser, with every test method and the two defined workload times.

After the benchmarking for one scenario is complete, the measuring framework produces a CSV file for downloading the recorded invocations in the background. Each row in the CSV file corresponds to one invocation of the simulated work load. The recorded data includes the time, since the web page moved to the background, the time since the last invocation and a computed average CPU usage. This computed CPU usage is derived from the last invocation time and the simulated work load duration with the following formula:

$$CPU_{avg} = \frac{t_{workduration}}{\Delta t_{lastinvocation}}$$

With this data we can then plot the computed CPU usage over time and compare different browsers for each scenario.

[4] https://developer.apple.com/app-store/review/guidelines/#software-requirements



(a) 50 ms task duration

(b) 1000 ms task duration

Figure 1: CPU usage over time for continuously scheduled timer tasks

4.2 Timer tasks

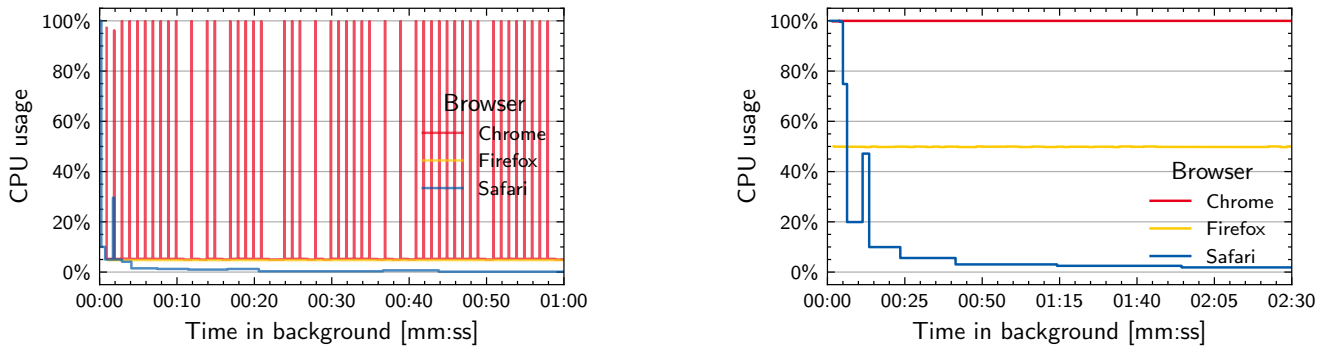
The default method to schedule new tasks for execution is to use the function `setTimeout()` or `setInterval()` which are available on window or worker global scope. These methods take at least two arguments. The first is the function which should be scheduled as a task. The second argument is the time in milliseconds to wait before the task should be executed. As this is the most straighforward way to schedule tasks, this method might be the one which gets throttled the most. The WHATWG specification for timers[6] even explicitly defines an optional waiting time which is user-agent defined to allow for optimization of power usage. Also, when `setTimeout()` calls are nested or `setInterval()` repeats the 5th time, the minimum waiting time is incresed to be 4ms.

Google Chrome throttles timers for a page, when it is in the background or not user visible [2]. Since Version 11 timers are batched at most once a second. This batching helps in reducing the battery impact of the background page. Since Version 57 Google Chrome also uses a budget based timer throttling. Budget based timer throttling works by introducing a timer budget for every page in the background. When the web site is in the background for longer then 10 seconds, the budget is considered. Every scheduled timer task is only executed when the budget for this page is greater then zero. The runtime of the task is subtracted from the budget. The budget regenerates for 0.01 seconds per second. This budget based throttling has multiple implications for background pages. First, it allows for sudden bursts in computing time, when these burst are very infrequent, because the budget regenereates continously, and can be depleted in a very short amount. Background web pages which want to use continous computation time, are limited to an average of 1% of CPU usage over time, because the budget regenerates at a rate of 1/100th of a second per second. The real CPU usage usually is a little higher then 1% on average though and only approaches 1% the longer the site is in the background, because the budget can be overdrawn with a long lasting task at the end of the measurement.

Firefox employs similiar throttling mechanisms to the ones from Google Chrome [11], though the details of the Firefox implementation differ slightly. When a page is in the background, Firefox invokes timers at least one second after the last timer finished. This is in contrast to Googles throttling to invocations once per second. The difference is most visible, when the task duration is 1 sec, because with Googles throttling implementation, the next task fires immediatly, whereas Firefox waits an additional second, before the next task is scheduled. Another differences is, that in Firefox, the budget based timer throttling is used after a page is in the background for 30 seconds, instead of 10 seconds for Google Chrome. Also Firefox caps the page budget at a minimum of -150 ms and a maximum of 50 ms. That means, that Firefox does not allow large burst of tasks, because the budget never increases above 50 ms, but longer lasting tasks, do not overdraw the budget more then -150 ms. This behaviour favors tasks, which are longer then 150ms. For example a repeated task with a duration of 1000 ms only needs to wait for 15 seconds, before it is executed again in Firefox, whereas the same task would have to wait for 100 seconds in Google Chrome, before the budget is posivite again. Firefox also employs additional throttling to tracking scripts [19], which limit repeated invocations to known tracker scripts to at most once every 10 seconds.

Safaris throttling behaviour is different then both Google Chrome's and Mozilla Firefox's behaviour. Scheduled tasks in background pages get invoked by Safari in increasing intervals. This ensures, that the CPU usage for background pages decreases much more rapidly then the CPU usage of the same page in Chrome or Firefox. Safari also coalesces timer calls, to prevent periodic wake ups of the CPU for the invocation of timers. Instead multiple timers are invoked one after the other to then let the CPU move to a energy saving state.

TODO: add iOS and Android



(a) 50 ms task duration

(b) 1000 ms task duration

Figure 2: CPU usage over time for continuously scheduled timer tasks during an open WebSocket connection

4.3 Timer tasks with WebSocket / AudioContext

In some browsers window timers in background tabs are not throttled, when either audible music is playing or a websocket connection is open. Playing a sound file or opening a websocket connection work equaliy in preventing the throttling, but they differ hugely in how visible both are the user of the web page. While playing audio is audible and also marks your tab with a speaker icon, so the user can quickly find out, which tab is producing the sounds, opening a websocket connection is invisible to the user. Also many browser prevent automatic playback of audio and videos, because it is considered bad behaviour of the website to autoplay audible sounds.

4.4 `postMessage()` tasks

4.5 Web workers

Using the worker-timers[5] library to run a scheduler on a web worker, which calls a callback on the main loop. This circumenvents the setInterval throttling, when a browser tab is in the background.

4.6 Service workers

Service workers have advantages. They run independent of the browser tab. They stay can stay aliver after the browser tab, which installed the service worker is closed.

- Multiple methods for background execution:
- Simple set interval after activation
- In response to network request (corresponding website has to be open to trigger a network call)
- Website push notifications (has to be allowed by user)
- Web Background Synchronization API[6]

Safari reloads background tabs which use too much energy after around 8 minutes with the message “This webpage was reloaded because it was using significant energy.”.

4.7 Summary

[5] `https://github.com/chrisguttandin/worker-timers`

[6] `https://wicg.github.io/BackgroundSync/spec/`

Method	Browser		
	Google Chrome	Mozilla Firefox	Apple Safari
Timers	Coalesced invocations at most once per second. Budget-based throttling after 10 seconds in background.	Next invocation at least one second after last invocation. Capped budget-based throttling after 30 seconds in background.	Coalesced invocations. Time between invocations increases.
Timers+WS	Coalesced invocations at most once per second. No budget-based throttling.	Next invocation at least one second after last invocation. No budget-based throttling.	Coalesced invocations. Time between invocations increases.
`postMessage`	No throttling	No throttling	No throttling, but page gets reloaded after around 8 minutes in the background
Web Worker	No throttling	No throttling	No throttling

Table 1: Summary of desktop browser throttling behaviour

5 Tracing of background execution on popular websites

In the first part of this thesis, we identified different methods to circumvent the default browser throttling mechanisms. With these findings in mind we can now trace popular websites[7] to analyse if these circumenvtion methods are used in the wild. Using the Alexa Top 1 million website list, we measure the average CPU usage of the websites. When we aggregate these tracing result, we can determine, if the browser throttling mechanisms are suiteable to limit CPU usage or if websites use these methods to actively or inadvertently bypass the background throttling mechanisms.

5.1 Automated measuring of background execution

To automate the tracing of popular websites we used the Puppeteer[15] library to control and automate Google Chrome. We choose Puppeteer because it allows us to access the Chrome internals like Web Profiler and also inject custom scripts into the loaded page to detect if circumvention methods which are described in part one of this thesis are being used. Puppeteer only allows to control Google Chrome, but with Google Chrome having the greatest market share among common Web browsers this is not a limiting factor.

The general idea for the tracing is, that we open a new Chrome instance and start the web profiling. Then we open the website to trace and wait for the initial load to complete. After that we move this website to the background by opening a new tab, so that the browser throttling mechanisms are activated. We profile the page for a 15 minutes time period. After the 15 minutes are completed, we close the browser instance and save the website trace for analysis.

The website trace allows us to understand if the website is running JavaScript code while it is in the background and which methods it used to initiate the execution. On a invidual site level the website trace profile allows us to see, which JavaScript functions were run and how long they took to complete.

The website trace profile allows us to dig deep into what a single website is executing while it is in the background. But to get a better picture of how popular websites in general behave in the background we have find a way to aggregate the analysis of the traces.

We propose to use the average CPU usage during the profiling to use as a score for a single website. The average CPU score can be calculated from the trace file by calculating summed duration, in which the website executed JavaScript code in the main thread and in spawned worker threads and dividing the sum by the time the trace was running. Wall clock time is a suitable replacement for CPU time in a scenario where the machine on which the measurements are taken is not under any other load from other processes. A website which uses no JavaScript at all should therefore receive a score of 0, whereas a website which does run uninterupted calculations on the main thread should receive a score of 1. If a website uses multiple worker threads and the machine on which the measurements are taken has more then one physical CPU core, then the website could receive a score which is greater then 1, because all workers could be active at the same time.

The background throttling mechanism put in place by Google Chrome does limit the timers of websites based on a CPU time budget. This budget is currently set so that websites on average only use 1% of CPU usage. That would equal a score of 0.01 in our proposed metric. This limit gives a good estimation, if websites try to overcome the background throttling mechanism of Chrome. Websites which score greater then 0.01 presumably use one or more methods to prevent the throttling.

[7] We used the first one thousand web sites of the Alexa Top 1m Website list

6 Evaluation of tracing results

7 Conclusion

8 References

[1] *Background tabs & offscreen frames: further plans*. URL: https://docs.google.com/document/d/18_sX-KGRaHcV3xe5Xk_l6NNwXoxm-23IOepgMx4OlE4/pub.

[2] *Background Tabs in Chrome 57*. URL: https://developers.google.com/web/updates/2017/03/background_tabs.

[3] *Concurrency model and Event Loop*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[4] Steven Englehardt and Arvind Narayanan. "Online Tracking: A 1-million-site Measurement and Analysis". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 1388–1401. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978313. URL: http://doi.acm.org/10.1145/2976749.2978313.

[5] Jeremiah Grossmann and Matt Johansen. "Million Browser Botnet, Conference talk at Black Hat USA 2013". In: USA, 2013.

[6] *HTML Living Standard — Last Updated 3 September 2019, Timers*. URL: https://html.spec.whatwg.org/multipage/timers-and-user-prompts.html#timers.

[7] Yoongu Kim et al. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 361–372. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665726.

[8] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[9] *Measure Performance with the RAIL Model*. URL: https://developers.google.com/web/fundamentals/performance/rail.

[10] Nick Nikiforakis et al. "You are what you include: large-scale evaluation of remote javascript inclusions". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012.

[11] *Page Visibility API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Page_Visibility_API.

[12] Yao Pan et al. "Gray computing: an analysis of computing with background JavaScript tasks". In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 167–177.

[13] Panagiotis Papadopoulos, Panagiotis Ilia, and Evangelos P. Markatos. *Truth in Web Mining: Measuring the Profitability and Cost of Cryptominers as a Web Monetization Model*. 2018. arXiv: 1806.01994 [cs.CR].

[14] Panagiotis Papadopoulos et al. *Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation*. 2018. arXiv: 1810.00464 [cs.CR].

[15] *Puppeteer*. URL: https://pptr.dev/.

[16] *Selenium WebDriver*. URL: https://www.seleniumhq.org/projects/webdriver/.

[17] Tom Van Goethem et al. "Clubbing seals: Exploring the ecosystem of third-party security seals". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 918–929.

[18] Wikipedia contributors. *Malvertising — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Malvertising&oldid=911222206. [Online; accessed 31-August-2019]. 2019.

[19] *WindowOrWorkerGlobalScope.setTimeout() — Reasons for delays longer than specified*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout#Reasons_for_delays_longer_than_specified.

[20] Apostolis Zarras et al. "The dark alleys of madison avenue: Understanding malicious advertisements". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM. 2014, pp. 373–380.