Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation

Panagiotis Papadopoulos,* Panagiotis Ilia,* Michalis Polychronakis,† Evangelos P. Markatos,*
Sotiris Ioannidis,* Giorgos Vasiliadis*
*FORTH, Greece, {panpap, pilia, markatos, sotiris, gvasil}@ics.forth.gr
†Stony Brook University, USA, mikepo@cs.stonybrook.edu

***Abstract*—The proliferation of web applications has essentially transformed modern browsers into small but powerful operating systems. Upon visiting a website, user devices run implicitly trusted script code, the execution of which is confined within the browser to prevent any interference with the user's system. Recent JavaScript APIs, however, provide advanced capabilities that not only enable feature-rich web applications, but also allow attackers to perform malicious operations despite the confined nature of JavaScript code execution.**

In this paper, we demonstrate the powerful capabilities that modern browser APIs provide to attackers by presenting MarioNet: a framework that allows a remote malicious entity to control a visitor's browser and abuse its resources for unwanted computation or harmful operations, such as cryptocurrency mining, password-cracking, and DDoS. MarioNet relies solely on already available HTML5 APIs, without requiring the installation of any additional software. In contrast to previous browser-based botnets, the persistence and stealthiness characteristics of MarioNet allow the malicious computations to continue in the background of the browser even after the user closes the window or tab of the initially visited malicious website. We present the design, implementation, and evaluation of our prototype system, which is compatible with all major browsers, and discuss potential defense strategies to counter the threat of such persistent in-browser attacks. Our main goal is to raise awareness about this new class of attacks, and inform the design of future browser APIs so that they provide a more secure client-side environment for web applications.

I. Introduction

Our increasing reliance on the web has resulted in sophisticated browsing software that essentially behaves as an integrated operating system for web applications. Indeed, contemporary browsers provide an abundance of APIs and sensors (e.g., gyroscope, location, battery status) that can be easily used by web applications through locally-running JavaScript code. The constantly expanding JavaScript interfaces available in modern browsers enable users to receive timely updates, render interactive maps and 3D graphics, or even directly connect to other browsers for peer-to-peer audio or video communication (e.g., through WebRTC).

In the era of edge computing, the capabilities offered by the available APIs have pushed a significant part of web application logic to the endpoints. Web publishers transfer parts of the critical computations on the user side, thus minimizing latency, providing satisfactory user experience and usability, while at the same time increasing the scalability of the service. Despite all these advancements, the web largely works in the very same way since its initial inception: whenever a user visits a website, the browser requests from the remote web server (and typically from other third-party servers) all the necessary components (e.g., HTML, CCS, JavaScript and image files), executes any script code received, and renders the website locally. That is, whenever a user visits a website, the browser blindly executes any received JavaScript code on the user's machine.

From a security perspective, a fundamental problem of web applications is that by default their publisher is considered as trusted, and thus allowed to run JavaScript code (even from third parties) on the user side without any restrictions (as long as it is allowed by the website's content security policy, if any). More importantly, users remain oblivious about the actual operations performed by this code. This problem has became evident lately with the widespread surreptitious deployment of cryptocurrency mining scripts in thousands of websites, exploiting the visitors' browsers without their consent [18], [66]. Although there are some blacklist-based extensions and tools that can protect users to some extent, such as Google's safe browsing, these do not offer complete protection.

On the other hand, disabling entirely the execution of JavaScript code often breaks intended legitimate functionality and affects the overall user experience. In general, the highly dynamic nature of JavaScript, the lack of mechanisms for informing users about the implemented functionality, and the instant execution of script code, which does not leave room for extensive security checks before invocation, are facilitators for malicious or unwanted in-browser code execution.

On the positive side, unwanted JavaScript execution so far has been constrained chronologically to the lifetime of the browser window or tab that rendered the compromised or malicious website. Consequently, cryptocurrency mining or other malicious JavaScript code can affect users only temporarily, typically for just a few minutes [53], depending on the time a user spends on a given website. Unfortunately, however, some recently introduced web technologies—already supported by the most popular browsers—can severely exacerbate the threat of unwanted JavaScript computation in terms of stealthiness, persistence, and scale, and the support of such capabilities has already started raising concerns of the community [29].

In this paper, we present MarioNet: a system that enables

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23070
www.ndss-symposium.org

a remote attacker to control users' browsers and hijack device resources. Upon visiting a website that employs MarioNet, the user's browser joins a centrally orchestrated swarm that exploits user machines for unwanted computation, and launching a wide variety of distributed network attacks. By leveraging the technologies offered by HTML5, MarioNet goes beyond existing approaches and demonstrates how malicious publishers can launch *persistent* and *stealthy* attacks. This is possible by allowing malicious actors to continue having control of the victim's browser *even after the user browses away* from a malicious or infected website, and by bypassing most of the existing in-browser detection mechanisms.

MarioNet consists of two main parts: (a) an in-browser component, and (b) a remote command and control system. Although MarioNet enables the attacker to perform attacks similar to those carried out by typical botnets [32], there are some fundamental differences. First and foremost, MarioNet does not exploit *any* implementation flaw on the victim's system and does not require the installation of *any* software. In contrast, MarioNet, leverages the provided capabilities of JavaScript and relies on some already available HTML5 APIs. Consequently, MarioNet is compatible with the vast majority of both desktop and mobile browsers. In contrast to previous approaches for browser hijacking (e.g., Puppetnets [4]), a key feature of MarioNet is that it remains operational even after the user browses away from the malicious webpage.

In particular, our system fulfills three important objectives: (i) isolation from the visited website, allowing fine-grained control of the utilized resources; (ii) persistence, by continuing its operation uninterruptedly on the background even after closing the parent tab; and (iii) evasiveness, avoiding detection by browser extensions that try to monitor the webpage's activity or outgoing communication. Besides malicious computation some of the attacks the infected browsers can perform include DDoS, darknet creation, and malicious file hosting and sharing.

Overall, in this paper, we make the following contributions:

1) Present *MarioNet*: a novel multi-attack framework to allow persistent and stealthy bot operation through web browsers. MarioNet is based on an in-browser execution environment that provides isolated execution, totally independent from any open browsing session (i.e., browser tab). Therefore, it is able to withstand any tab crashes and shutdowns, significantly increasing the attacker's firepower by more than an order of magnitude.
2) Demonstrate and assess the feasibility of our approach with a proof of concept implementation of MarioNet for the most common web browsers (i.e., Chrome, Firefox, Opera, and Safari). To measure its effectiveness, we thoroughly evaluate MarioNet for various attack scenarios.
3) Discuss in detail various defense mechanisms that can be applied as countermeasures against MarioNet-like attacks.

The main goal of this work is to raise awareness about the powerful capabilities that modern browser APIs provide to attackers, so that a more secure client-side environment can be provided for web applications in the future.

II. Background

In this section, we discuss several features that have been recently introduced as part of HTML5 and influence our design. We also discuss the capabilities of web browser extensions, especially with regards to these HTML5 features, and finally, for each feature, we analyze its security aspects, access policies, permissions, and threat vectors that may open.

A. HTML5 features

1) Web Workers: Browsers typically have one thread that is shared for both the execution of JavaScript and for page rendering processing. As a result, page updates are *blocked* while the JavaScript interpreter executes code, and vice versa. In such cases browsers typically ask the user whether to kill the unresponsive page or wait until the execution of such long-running scripts is over. HTML5 solves this limitation with the Web Workers API [46], which enables web applications to spawn background workers for executing processing-intensive code in separate threads from the browser window's UI thread.

Since web workers run as separate threads, isolated from the page's window, they do not have access to the Dynamic Object Model (DOM) of the webpage, global variables, and the parent object variables and functions. More specifically, neither the web worker can access its parent object, nor the parent object can access the web worker. Instead, web workers communicate with each other and with their parent object via message passing. Web workers continue to listen for messages until the parent object terminates them, or until the user navigates away from the main webpage. Furthermore, there are two types of web workers: dedicated and shared workers. Dedicated web workers are alive as long as the parent webpage is alive, while shared web workers can communicate with multiple webpages, and they cease to exist only when all the connections to these webpages are closed.

Typically, web workers are suitable for tasks that require computationally intensive processing in an asynchronous and parallel fashion, such as parsing large volumes of data and performing computations on arrays, processing images and video, data compression, encryption etc. Indeed, during the recent outbreak of web-based cryptocurrency mining, we have observed that typically these scripts utilize web workers for mining, and that they deploy multiple such workers to utilize all available CPU cores of the user's system.

2) Service Workers: Service workers are non-blocking (i.e., fully asynchronous) modules that reside in the user's browser, in between of the webpage and the publisher's web server. Unlike web workers, a service worker, once registered and activated, can live and run in the background, *without* requiring the user to continue browsing through the publisher's website—service workers run in a separate thread and their lifecycle is completely independent from the parent page's lifecycle. The characteristics of service workers enable the provision of functionality that cannot be implemented using web workers, such as push notifications and background syncing with the publisher. Furthermore, another core feature of service workers is their ability to intercept and handle network requests, including programmatically managing the caching of responses. This allows developers to use service workers as programmable network proxies, thus enriching the offline user experience by controlling how network requests from a webpage are handled.

A service worker can be registered only over HTTPS via the `serviceWorkerContainer.register()` function,

which takes as argument the URL of the remote JavaScript file that contains the worker's script. This URL is passed to the internal browser's engine and is fetched from there. For security purposes, this JavaScript file can be fetched only from the first-party domain (i.e., cannot be hosted in a CDN or other third-party servers). Also, no iframe or third-party script can register its own service worker. Importantly, no browser extension or any in-browser entity can have access either in the browser's C++ implementation that handles the retrieval and registration of the service worker or in the first-party domain.

When the user browses away from a website, the service worker of that website is typically paused by the browser; it is then restarted and reactivated once the parent domain is visited again. However, it is possible for the publisher of a website to keep its service worker alive by implementing periodic synchronization. It should be noted though that the registration of a service worker is entirely non transparent to the user, as the website *does not* require the user's permission to register and maintain a service worker. Furthermore, similarly to web workers, service workers cannot access the DOM directly. Instead, they communicate with their parent webpages by responding to messages sent via the `postMessage` interface.

3) WebRTC: Popular web-based communication applications (such as Web Skype, Google Meet, Google Hangouts, Amazon Chime, Facebook Messenger) nowadays are based on Web Real-Time Communication (WebRTC) API [23], which enables the establishment of peer-to-peer connections between browsers. The WebRTC technology enables browsers to perform real-time audio and video communication and exchange data between peers, without the need of any intermediary.

As in every peer-to-peer protocol, a challenge of WebRTC is to locate and establish bidirectional network connections with remote peers residing behind a NAT. To address this, WebRTC uses STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT) servers for resolving the network address of the remote peer and reliably establishing a connection. There are several such servers publicly available [21], maintained either by organizations (e.g., Universities) or companies (e.g., Google).

4) Cross-Origin Resource Sharing: Before HTML5, sending AJAX requests to external domains was impossible due to the restrictions imposed by the same-origin policy, which restricts scripts running as part of a page in accessing only the DOM and resources of the same domain. This means that a web application using AJAX APIs (i.e., `XMLHttpRequest` and the Fetch API) can only request resources from the same domain it was loaded.

However, the Cross-Origin Resource Sharing (CORS) [43] capabilities introduced in HTML5, allow scripts to make cross-origin AJAX requests to other domains. To enable this feature, CORS uses extra HTTP headers to permit a user agent to access selected resources from a server on a different domain (origin) than the parent sits. Additionally, for HTTP request methods that can cause side-effects on server-side data (in particular, for HTTP methods other than `GET`, or for `POST` usage with certain MIME types), the specification mandates browsers to "preflight" the request, soliciting supported methods from the server with an HTTP `OPTIONS` request method, to determine whether the actual request is safe to send.

B. Web Extensions

The current design of modern browsers' extensions allows two types of JavaScript scripts within a browser extension: (a) content scripts and (b) background scripts. Content scripts run in the context of the websites visited by the user, thus they can read and modify the content of these websites using the standard DOM API, similarly to the websites' scripts (i.e., those JavaScript scripts that were included in the website by the publisher). Furthermore, content scripts can access directly a small subset of the WebExtension JavaScript APIs.

On the other hand, background scripts run as long as the browser is open (and the extension is enabled), and typically implement functionalities independent from the lifetime of any particular website or browser window, and maintain a long-term state. These background scripts cannot access directly the content of the websites visited by the user. However, background scripts can access all the WebExtension JavaScript APIs (or chrome.* APIs for Google Chrome), if the user's permission is granted during the installation of the extension.

Indicatively, the large set of WebExtension JavaScript APIs contains the bookmarks, cookies, history and storage APIs, which allow access on various types of user data, the tabs and windows APIs, browserSettings and the webRequest API among many others. However, even though content scripts cannot access all WebExtension APIs directly, and background scripts cannot access the content of the visited website, this can be achieved indirectly since the content and background scripts of an extension can communicate with each other.

In addition to the above mentioned APIs, Google Chrome also supports some HTML5 and other emerging APIs for its extensions (e.g., application cache, local storage, geolocation, canvas, notifications, etc.). However, it is important with regards to this work to emphasize that *none of the browsers allow extensions to use HTML5 APIs* such as the Service Workers API or the Push API. Consequently, browser extensions cannot interact with possible deployed service workers in any way (e.g., modify their code, monitor their outgoing traffic, etc.).

C. Security Analysis

Table I summarizes the characteristics of various APIs of interest. We categorize them along four axes related to the efficiency of a distributed botnet: (i) the execution model (i.e., whether it can run in parallel to the main webpage or in the background), (ii) if direct network access is possible, (iii) the ability to use persistent storage, and (iv) the ability to access the DOM of the webpage.

JavaScript code (running either as part of the webpage, or in a web worker or service worker) has access to persistent storage (e.g., using WebStorage), as well as the ability to communicate with other servers or peers (e.g., using XHR requests, WebSockets, or WebRTC). However, local JavaScript code embedded in the webpage also has direct access to the page's DOM and therefore, the ability to access or manipulate any element of the webpage, as well as any network request or response that is sent or received. Page-resident JavaScript code cannot be detached from the webpage, neither run without blocking the rendering process. This results to a major limitation (for the purposes of malicious scripts), as long-running operations would affect the user experience. Also,

TABLE I: Analysis of HTML5 JavaScript execution methods.

Feature	Concurrent Execution	Background Execution	Webpage Detached	Intercept HTTP Requests	Persistent Storage	DOM Access	Network Access
Local JavaScript code				✓	✓	✓	✓
Web Worker (Shared)	✓				✓		✓
Web Worker (Dedicated)	✓				✓		✓
Service Worker	✓	✓	✓	✓	✓		✓

a suspicious code snippet could be detected easily by browser extensions, since it needs to be embedded in the main website, and extensions' JavaScript code can access, inspect and in general, interfere with the content of the visited website.

Web workers, on the other hand, can perform resource-intensive operations without affecting the user's browsing experience, as they run in separate threads. This allows utilizing all different available CPU cores of the user's machine, by spawning a sufficient number of web workers. Service workers behave in a similar fashion, but have the important advantage of being completely detached from the main webpage, running in the background even after the user has navigated away. Moreover, service workers can intercept the HTTP requests sent by the webpage to the back-end web server. Importantly, since service workers are completely detached from the page's window, extensions cannot monitor or interfere with them.

Finally, using the CORS capabilities of HTML5, it is possible to send multiple GET or POST requests to third-party websites. However, the `Access-Control-Allow-Origin:*` header has to be set by the server, in order for the request to be able to fetch any content. Besides sending HTTP requests, WebRTC allows the peer-to-peer transfer of arbitrary data, audio, or video—or any combination thereof. This feature can open the window for malicious actions such as illegal hosting and delivery of files, and anonymous communication through a network of compromised browsers, as we showcase later on.

III. Threat Model and Objectives

The motivation behind this work is to design a system capable of turning users' browsers into a multi-purpose "marionette" controlled by a malicious remote entity. Our goal is to leverage *solely* existing HTML5 features in order to highlight the lack of adequate security controls in modern browsers that would have prevented the abuse of these advanced features.

A. Threat Model

We assume a website that delivers malicious content to execute unwanted or malicious *background* operations in visitors' browsers. Once the website is rendered, this malicious content is loaded in a service worker that is capable of continuing its operation even *after the victim browses away from the website*.

Websites can deliver such malicious or unwanted content *intentionally*, to gain profit directly (e.g., by attracting visitors and thus advertisers), or indirectly, by infecting as many user browsers as possible to carry out distributed (malicious) computations or mount large-scale attacks. The websites in this category can range from typically malicious ones, to websites of shady reputation, and even to trustworthy and reputable websites that aim to increase their revenue without actually intending to conduct any illegal activities or harm the user.

There are also several cases where a website can end up hosting such malicious content *unintentionally*. Those cases include: (i) the website registers a benign service worker that includes untrusted dynamic third-party scripts [35], which in turn possibly load malicious code; (ii) the website includes third-party libraries,[1] one of which can turn rogue or be compromised, and then divert the user to a new tab (e.g., using popunders [22] or clickjacking [64]) where it can register its own service worker bound to a *third-party* domain; (iii) the website is compromised and attackers plant their malicious JavaScript code directly into the page, thus registering their malicious service worker—a scenario that we see quite often in recent years [40], [36]; or (iv) the website includes iframes with dynamic content, which are typically auctioned at real-time [57] and loaded with content from third parties.

In the latter case, malicious actors can use a variety of methods (e.g., redirect scripts [28], [37] or social engineering) to break out of the iframe and open a new tab on the user's browser for registering their own service worker. The important advantage of this latter approach is that the user does not need to re-visit the website for the service worker to be activated. After registration, just an iframe loaded from the malicious third party is enough to trigger the malicious service worker, regardless of the visited first-party website. This relieves the attackers from the burden of maintaining websites with content attractive enough to lure a large number of visitors. Instead, attackers can activate their bots just by running malvertising campaigns, purchasing iframes in ad-auctions [67].

To summarize, our threat model considers that such an attack can be launched intentionally by a malicious or "shady" website that includes malicious content, unintentionally, by a hijacked/compromised website or a website that includes a compromised library, and also by third-party dynamic content loaded in iframes (typically used for real-time ad auctions).

B. Challenges

The greatest challenge for systems like MarioNet is to keep the user's device under control for as long as possible. This is a challenging task given that a connection with the server may be possible only for the duration of a website visit; recent studies have estimated the average duration of a typical website visit to be less than one minute [53]. In addition, there is a plethora

[1] Modern websites often include numerous third-party scripts [55], [14] for analytics or user tracking purposes, aiming to gain insight, improve performance, or collect user data for targeted advertising.

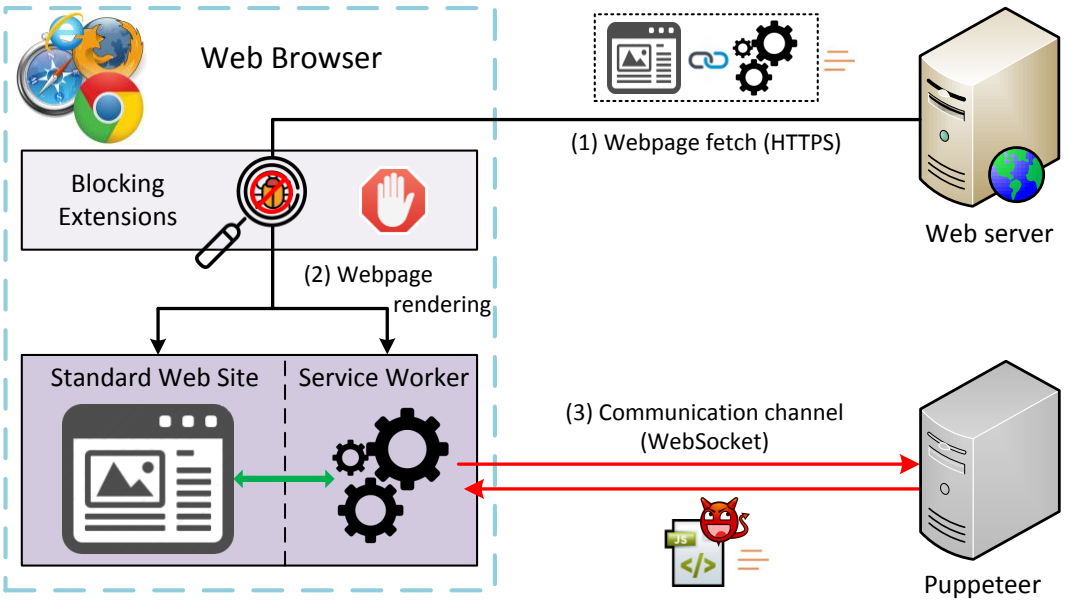

Fig. 1: High level overview of MarioNet. The in-browser component (Servant), embedded in a Service Worker, gets delivered together with the actual content of a website. After its registration on the user's browser, it establishes a communication channel with its remote command and control server (Pupeteer) to receive tasks.

of sophisticated browser extensions [20], [39] that monitor the incoming and outgoing traffic of in-browser components. Consequently, another challenge for MarioNet is to evade any such deployed countermeasure installed in the browser. Finally, it is apparent that the malicious or unwanted computation of MarioNet must not impede the normal execution of the browser, and avoid degrading the user experience. Otherwise, the risk of being detected by a vigilant user or at least raising suspicion due to reduced performance is high.

To summarize, in order to overcome the above challenges, a MarioNet-like system should have the following properties:

1) *Isolation:* the system's operation must be independent from a browsing session's thread or process. This isolation will allow a malicious actor to perform more heavyweight computation without affecting the main functionality of the browser.
2) *Persistence:* the operation must be completely detached from any ephemeral browsing session, so that the browser can remain under the attacker's control for a period longer than a short website visit.
3) *Evasiveness:* operations must be performed in a stealthy way in order to remain undetected and keep the browser infected as long as possible.

IV. System Overview

In this section, we describe the design and implementation of *MarioNet*, a multi-purpose web browser abuse infrastructure, and present in detail how we address the challenges outlined earlier. Upon installation, MarioNet allows a malicious actor to abuse computational power from users' systems through their browsers, and perform a variety of unwanted or malicious activities. By maintaining an open connection with the infected browser, the malicious actor can change the abuse model at any time, instructing for instance an unsuspecting user's browser to switch from illicit file hosting to distributed web-based cryptocurrency mining.

Our system, which is OS agnostic, assumes *no* assistance from the user (e.g., there is no need to install any browser extension). On the contrary, it assumes a "hostile" environment with possibly more than one deployed anti-malware browser extensions and anti-mining countermeasures. We also assume that MarioNet targets off-the-shelf web browsers. Hence, the execution environment of MarioNet is the JavaScript engine of the user's web browser. Breaking out of the JIT engine [5] is beyond the scope of this paper.

A. System components

Figure 1 presents an overview of MarioNet, which consists of three main components:

1) **Distributor:** a website under the attacker's control (e.g., through the means discussed in Section III-A), which delivers to users the MarioNet's Servant component, along with the regular content of the webpage. It should be noted that the attacker does not need to worry about the time a user will spend on the website. It takes only one visit to invoke MarioNet and run on the background as long as the victim's browser is open.
2) **Servant:** the in-browser component of MarioNet, embedded in a service worker. It gets delivered and planted inside the user's web browser by the Distributor. Upon deployment, the Servant establishes a connection with its Puppeteer through which it sends heartbeats and receives the script of malicious tasks it has to perform. The Servant runs in a separate process and thereby it continues its operation uninterruptedly even after its parent tab closes.
3) **Puppeteer:** the remote command and control component. This component sends tasks to the Servant to be executed,

and orchestrates the performed malicious operations. The Puppeteer is responsible for controlling the intensity of resources utilization (CPU, memory, etc.) on the user side, by tuning the computation rate of the planted Servant.

As illustrated in Figure 1, MarioNet is deployed in three main steps: First, (step 1) the user visits the website (i.e., the Distributor) to get content that they are interested in. The Distributor delivers the JavaScript code of the Servant along with the rest of the webpage's resources. During the phase webpage rendering (step 2), the Servant is deployed in the user's browser. As part of its initialization, the Servant establishes a communication channel with its remote command and control server (Pupeteer) and requests the initial set of tasks (step 3). The Pupeteer, which is maintained by the attacker, responds with the malicious script (e.g., DDoS, password cracking, cryptocurrency mining) the Servant has to execute.

B. Detailed Design

MarioNet leverages existing features of HTML5 to achieve the objectives presented in Section III: isolation, persistence, and evasiveness. In-browser attacks that involve computationally heavy workloads require *isolation* in order to avoid interfering with a webpage's core functionality. Previous approaches [13], [54] rely on web workers to carry out heavy computation in the background (in a separate thread from the user's interface scripts). Although this isolation also prevents the code of the web worker from having access to the DOM of the parent page, it has the benefit of allowing multi-core utilization. As a result, attackers can utilize multiple cores for their malicious computations. However, web workers run in the same browser tab as the website, and consequently, their execution is tightly coupled with the parent tab: whenever the tab closes, the web worker terminates as well. In addition, security-related browser extensions can (i) monitor all traffic and (ii) tamper with the script running in the web worker.

To remedy these shortcomings, MarioNet leverages a different component of HTML5, namely *service workers*. As described in Section II-A2, service workers are typically used as an in-browser caching proxy, serving the user during offline periods. In contrast to web workers, service workers run in a separate process, completely detached from the parent tab. In addition to service workers, we use the SyncManager interface [45] to register background "sync registrations" for the service worker, to keep the Servant always alive. The tab independence and indefinite lifetime properties of the Servant provide MarioNet with *persistence*, allowing attackers to carry out their malicious computation for the entire period that a browser remains open—a major benefit over existing approaches based on web workers, which remain operational only for the duration of a browsing session (open tab).

Another advantage of leveraging service workers is that they conceptually operate between the browser and the remote server. As a consequence, any security monitoring performed by browser extensions cannot monitor the activity and network communication of the service worker, allowing the Servant to operate in a stealthy way. Consequently, the Servant can establish a communication channel with the remote Puppeteer that no browser extension can snoop. In addition, the established communication channel is TLS-encrypted (as required by the service worker API [19]), ensuring the integrity and confidentiality of the transmitted data. Consequently the C&C communication channel cannot be inspected by any eavesdropping third party sitting either (i) inside (e.g., browser extension) or (ii) outside (e.g., ISP) of the browser.

The only request that reveals the existence of the service worker is the initial GET request at the time of the user's first website visit, when the service worker gets initially registered. Although during that GET request a monitoring extension can observe the contents of the service worker, it will still not observe any suspicious code—the code that will carry out the malicious tasks is delivered to the Servant only after its first communication with the Puppeteer, and this communication is hidden from browser extensions (as discussed in Section II-B)

Along with the evasiveness of MarioNet against monitoring and blocking extensions, it is also important to maintain its stealthiness to avoid detection from users themselves. Existing web-based botnet approaches [13], [54] follow an opportunistic approach, utilizing greedily all available resources on the device during their limited period of activity. When browsers run such a malicious script, the louder noise of the fans, the sudden power drainage, or the sluggish responsiveness of their system, alerts users who are likely to close the associated browser tab, or even report the website to their blocker extension.

In contrast to existing in-browser attacks, MarioNet aims to prolong its presence on a user's device by allowing the attacker to monitor the device's state at real time, and adjust accordingly the resources utilization to minimize the possibility of getting detected. To that end, the Servant monitors the device's current status (e.g., CPU utilization, battery status) and by utilizing HTML5's high-resolution performance timers [72], throttles or even pauses the execution of the malicious workload. This allows it to minimize the risk of self-exposure in case there is a CPU capping mechanism in the browser [8].

Persistence across Browser Reboots: MarioNet runs in the background as long as the browser is open. After that, the victim has to re-visit the malicious domain or render the malicious iframe where the malicious domain resides, in order to re-activate the service worker and allow the Servant to continue its operation. To increase persistence even further, we have developed a technique that allows MarioNet to persist even after the browser has been restarted. This can be achieved by utilizing the Push API [44]. This feature allows a web server to deliver asynchronous notifications and updates to service workers, in an attempt to provide users with better engagement and timely new content. By abusing this mechanism, MarioNet can enable the Puppeteer to periodically probe its Servants and re-activate them after the browser restarts.

In contrast to the non-transparent to user process of service worker registration, security policies in modern browsers restrict the use of the Push notifications feature only after the user's permission. Of course, some users may get suspicious on that behavior, depending on the website they visit. However, an advanced attacker can convince reluctant users to give their consent for push notifications by advertising enticing offers (e.g., virtual points or participation to contests) or by performing more advanced types of social engineering using custom permission requesting popups. Recent studies have shown that 12% of users give such permissions when they are

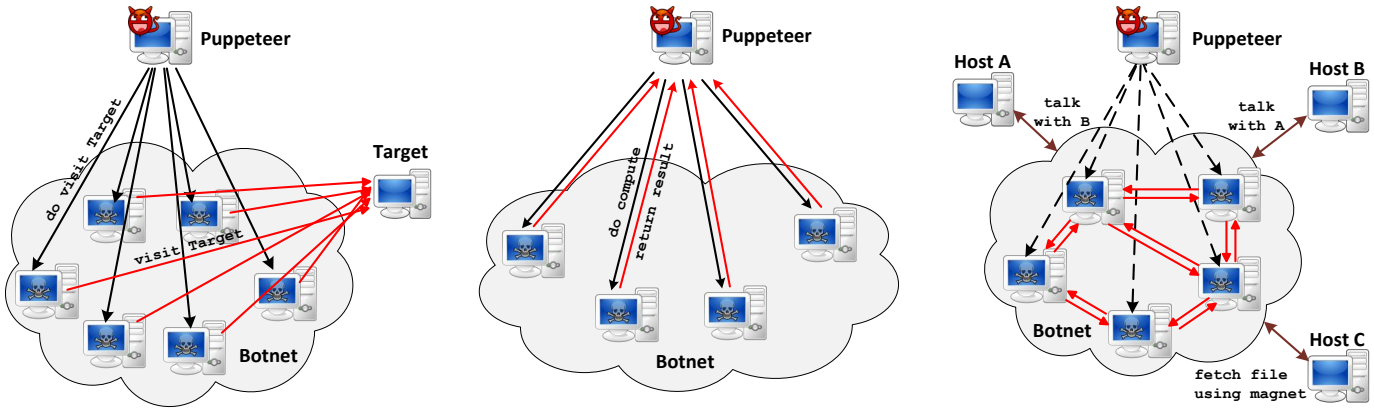

Fig. 2: Different use cases of MarioNet. After victims get compromised, the attacker can instrument them to perform (a) visits to a selected server or URL, for DDoS attack or fake ad-impressions, (b) requested computations, such as cryptocurrency mining or password cracking, and (c) illegal services, such as illicit file hosting or hidden/anonymized communications.

asked to [3], which constitutes a fairly large number of nodes, sufficient for deploying a persistent botnet that is capable to survive browser reboots.

V. Attack Vectors

Our design, described in Section IV, opens the space for a diverse set of attacks in users' web browsers, which can be categorized in three models, as shown in Figure 2.

A. DDoS Attacks

A simple yet powerful attack that can be launched with the devices the attacker controls is a Distributed Denial-of-Service attack. In MarioNet we implemented a DDoS attack module enabling the Puppeteer to instruct the Servants to connect to a specific Internet host. As a result, the targeted host will get overwhelmed by the large amount of connections and thus become non-responsive to benign users.

A limitation of using a high-level language, such as JavaScript, to initiate a DoS attack is that it does not provide low level networking access. Directly manipulating the network packets to be sent is thus not an option (e.g., force TCP-SYN packets only, or spoof source network address). In addition, it results to much higher latency, due to the extra memory copies and context switches that are caused from the resulting system calls. Instead, JavaScript offers more high-level approaches, such as `XMLHttpRequest` objects [48] or methods provided by cross-platform libraries (e.g., the `get()`, `post()`, and `ajax()` methods provided by jQuery). These methods can be used to perform HTTP GET and POST requests, either synchronously (i.e., in a blocking fashion, waiting for the connection to be established) or asynchronously. In addition, some methods may return cached responses (e.g., the `get()` method provided by jQuery).

In order to increase the DDoS fire power of MarioNet, we use the `XMLHttpRequest` API, which can be used to perform AJAX (asynchronous HTTP) requests, and does not cache any responses. Moreover, it allows to control the request method, and set an arbitrary HTTP body, as well as some HTTP request headers (e.g., the request content type). One concern though, that we already mentioned in Section II-A4, is that if the target web server does not enable the `Access-Control-Allow-Origin:*` header, the request will not fetch any content. Even in that case though, the attack can still succeed, as it does not necessarily rely on forcing the web server to send a response. As long as the requests are sent, the incoming network link is filling up and also the server needs to spend resources to handle the incoming requests.

Apart from HTTP fetching mechanisms, HTML5's WebSockets API [47] provide additional opportunities. WebSockets can be used to send messages to a WebSocket-enabled server over TCP and receive event-driven responses. Obviously, to mount a DoS attack using WebSockets, the targeted server needs to implement this protocol; this is indeed the case for many popular web sites, as well as for smaller ones, which increasingly adopt the WebSockets protocol. Besides that, as already has been shown in [58], malicious JavaScript code may still misuse the handshake by requesting resources even by targeting a non-WebSocket web server. Although the targeted web server may ignore the characteristic WebSocket HTTP headers (as it is not supported), it can still accept WebSocket handshake HTTP requests as normal HTTP requests [58]. As a result, the web browser will start the WebSocket handshake with the target, while the non-WebSocket web server will process the HTTP request as a valid request. In MarioNet, we use the `WebSocket()` method to initiate connections with web servers, and then the `send()` method to send a flood of data to the targeted server.

Using `XMLHttpRequest.send()`, jQuery's `ajax()` and WebSocket's `send()` methods, we can continuously send a flood of messages to a targeted host. Each approach allows MarioNet to connect to any host, by specifying the hostname or IP address and the corresponding port number. By doing so, JavaScript code can misuse the TCP handshake by requesting connections even to non-HTTP or non-WebSocket servers. In those cases, the targeted servers will either receive only the TCP SYN packets (e.g., when the destination port is in a closed, reject, or drop state), or the full HTTP request. Furthermore,

the WebSocket API allows to open many different connections, which enables attackers to orchestrate different styles of attacks (e.g., stealthy, low-volume, etc.). For instance, it allows to perform Slowloris-like attacks, by keeping many connections to the target server open as long as possible [9].

Of course, MarioNet cannot send messages to any port at the targeted host. To avoid Cross-protocol Scripting [69], which allowed the transmission of arbitrary data to any TCP port, modern browsers block by default outgoing messages to a list of reserved ports [42]. Finally, we note that the resulting network performance of JavaScript is not that high, compared to DoS attack tools that can leverage direct access to OS internals (i.e., memory map techniques between the network interface and the application) and low-level APIs (i.e., raw sockets). However, this is not a serious limitation, as it has been shown that short, low-volume DDoS attacks pose a great security and availability threat to businesses [71].

B. Cryptocurrency Mining

The rise of lightweight cryptocurrencies, such as JSEcoin and Monero, together with the features of Web Workers API that have been described in Section II, have recently enabled the widespread adoption of cryptocurrency mining on the Web. As a result, attackers have started migrating mining algorithms to JavaScript and embed them to regular websites, in the form of web worker tasks. By doing so, the website visitors become mining bots unwittingly every time they access these websites.

However, the short website visiting times make the profitability of the web workers approach questionable [65]. Instead, MarioNet increases the potential profits of web cryptocurrency mining, due to the background execution it offers, completely detached from the website. As a matter of fact, we have implemented a service worker module that computes hashes of the popular CryptoNight algorithm [11]. CryptoNight is a proof-of-work (PoW) algorithm used in several cryptocurrencies, such as Electroneum (ETN) and Monero (XMR). The service worker that we have implemented within MarioNet, connects with Coinhive [10], which is a web service that provides an API for users to embed a JavaScript miner on their websites. Alternatively, the cryptocurrency miner can connect to any mining pools, through the HTTP stratum proxy, using a registered account, as shown in previous works [54]. By doing so, attackers will be credited the payout directly to their wallets. Finally, we notice that other hash algorithms that are used for cryptocurrency mining, such as Scrypt-based miners [7], can be implemented in a straightforward way by porting their implementations to JavaScript.

C. Distributed Password Cracking

The idea of distributed password hash cracking on the web is not new [4]. Orthogonal to other approaches that try to boost the sustained performance by either increasing the parallelism using different web workers [13], or exploiting the computational capabilities of modern GPUs using the WebGL/WebCL API [52], MarioNet can help towards increasing the uptime of hash cracking techniques, and as a result the overall performance.

The basic concept in MarioNet is to have the Puppeteer distribute the computation between the infected browsers. The server contains a list of the hashes to be cracked and gives each node a range of character combinations along with the hash to be cracked. Each node then hashes these combinations and checks if it matches the original hash; if it matches, the node reports the recovered password back to the Puppeteer. A major advantage of MarioNet is that it can be agnostic to the hashing function used, since the function code is transferred from the Puppeteer and executed from the MarioNet nodes through `eval()`. As a matter of fact, in Figure 7 we show the performance achieved by MarioNet for executing two popular hashing algorithms, namely SHA-256 and MD5.

D. Malicious or Illegal Data Hosting

Having a large network of MarioNet nodes can also enable the delivery of illegal or otherwise unwelcome content. The advantages of MarioNet is not only that the content can be served by unsuspecting users, making it hard to track down the real culprits behind it, but also allows efficient data distribution between the MarioNet nodes.

Indeed, the release of WebRTC (Web Real-Time Communications) protocol in the browser a few years ago, enables peer-to-peer networking communications. In particular, WebRTC allows web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary. Even though this technology opens new opportunities for distributed networking to the web, it also brings some significant security concerns when used maliciously. In the case of MarioNet, for instance, it could be easily used as an illegal content provider, leveraging the distributed nature and persistence that offers. As a proof-of-concept, similar to [13] we used the WebTorrent API [16] to implement a simple, yet flexible, data hosting mechanism over WebRTC which allows the sharing of torrent files through the infected MarioNet nodes. WebTorrent allows users to seed and leech files with other peers entirely through their web browsers. A new torrent file can easily be created using the `seed()` function which creates a new torrent and starts seeding it. The file can then be downloaded and further seeded from other nodes, using the returned `magnetURI`.

E. Other Attacks

1) Relay Proxies: Fully anonymous and transparent relay proxies that can route data between two peers, are an important asset for criminal use, making it difficult for the authorities to track down the perpetrators. Large groups of such proxies can form a hidden network (i.e., Darknet), where people buy and sell illicit products like weapons and drugs [12].

The MarioNet infrastructure can provide a platform for establishing such networks. Specifically, an infected browser can be used as an intermediate proxy to fetch illegal content from services in the Darknet on behalf of an anonymous user. Indeed, building upon the previous illegal data hosting scenario, MarioNet could form anonymous circuits (similar to mixnets), through which users could route their web traffic. Such chain could be created by bots connected with encrypted peer-to-peer channels with each other by using WebRTC.[2]

[2]WebRTC traffic is always encrypted. Transmitted data is protected by Datagram Transport Layer Security (DTLS) [49] and Secure Real-time Transport Protocol (SRTP) [41].

There are already such browser-based proxies implemented over WebRTC, like Stanford's Flash Proxies [17] and Tor Project's Snowflake [26]. Apparently, a solid implementation of such a service within a service worker, capable of providing strong anonymity guarantees (e.g., similar or close to Tor), is not a trivial task and requires deeper analysis. Hence, such an exploration is beyond the scope of this paper.

2) Click Fraud: Having a large botnet can become profitable in many ways. One such way is to abuse the digital advertising ecosystem, by having bots rather than humans view or click on online advertisements. It is estimated that online advertising fraud will cost advertisers $19 billion in 2018, which represents 9% of total digital advertising spend [31].

MarioNet can be easily used to generate clicks, as well as surf targeted websites for a period of time, stream online videos to increase views, manipulate online polls, and possibly sign up for newsletters. To achieve that, the service worker can obtain periodically a list of online links that is requested to visit, possibly combined with metadata such as visit duration, number of clicks, etc. In addition, due to the rich programming features that JavaScript offers, MarioNet can be easily programmed to follow a human-centric online behavioral model (e.g., similar to the one proposed by Baldi et al. [6]) to evade countermeasures that seek to block users with unusual activity (e.g., clicking too many links in a short period of time).

VI. Evaluation

A. Prototype Setup

To assess the feasibility and effectiveness of our approach, and also to check the existence of possible code protection and restriction mechanisms, we build a real world deployment of our MarioNet prototype. Our prototype consists of two servers; the first server is an Apache web server that hosts a simple webpage, and the second one is a command and control server (i.e., Puppeteer), delivering tasks to the Servants. Upon the first website visit, the webpage registers a service worker in the Servant and a sync manager that is responsible to keep the service worker alive in the background. After its registration, the Servant opens a full-duplex connection—using the WebSocket API [47]—with the Puppeteer and retrieves a JavaScript code snippet that executes through `eval()`. In order to be able to use `eval()` from within the service worker, the collaborating web server gives the needed permission through the HTTP Content Security Policy (CSP).

Browser Compatibility: As discussed in Section IV, our approach is based on existing components of HTML5 such as Service Workers and its interface SyncManager. Table II summarizes the browser compatibility of these components, and thus the compatibility of our framework. As we can see, some vendors like Google, started supporting service workers quite early (2016), while others caught up only until recently, i.e., Safari (2018). Still, MarioNet is compatible with the most popular browsers in both desktop and mobiles.

In our experiments we tested MarioNet with four popular desktop browsers, namely Chrome, Firefox, Opera and Safari. However, we chose to exclude Safari from the performance evaluation results, due to its bad performance sustained in all the experiments conducted. Even though the service worker functionality is provided by Safari, we experienced several performance glitches. We believe that this behavior is due to the recently adaptation of service workers in Safari (2018). Even for simple workloads, i.e., a simple counting example, the performance achieved by the service worker is extremely slow (i.e., $20 - 50\times$ lower) compared to the performance achieved by the other three browsers.

TABLE II: MarioNet's browser compatibility

Device	Browser	SW compatibility
Desktop	Chrome	since v40
	Firefox	since v44
	Opera	since v26
	Edge	since v17
	Safari	since v11.1
	IE	NoSupport
Mobile	Samsung Internet	since v4
	Chrome Android	since v64
	UC Browser	since v11.8
	iOS Safari	since v11.3
	Firefox Android	since v57
	Android Browser	Partially since v62
	Opera Mobile	Partially since v37
	Opera Mini	NoSupport
	Blackberry	NoSupport

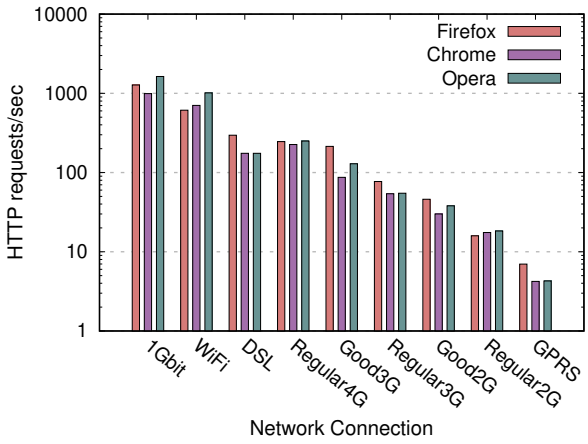

Fig. 3: Rate of asynchronous outgoing HTTP OPTION requests for different browsers and network connections in the DDoS scenario. An orchestrated DoS attack in MarioNet can achieve rates of up to 1632 reqs/sec per infected device.

B. Performance Evaluation

In order to demonstrate the effectiveness of MarioNet, we conduct several experiments with various popular browsers and hardware settings, which allow us to make useful and interesting comparisons. However, it is noted that in this paper we do not aim to provide an optimal implementation in terms of performance, but rather to demonstrate the feasibility of the aforementioned attacks. To that end, the performance of the system can be further improved by using WebAssembly. Furthermore, all the experiments presented in this section were

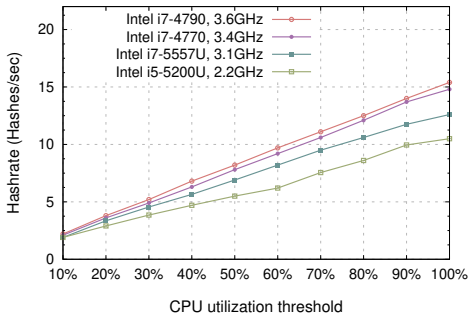

Fig. 4: Hashrate for different equipped CPUs and utilization levels in the cryptojacking scenario. As expected, victim's hardware affects significantly the computation power that MarioNet may obtain.

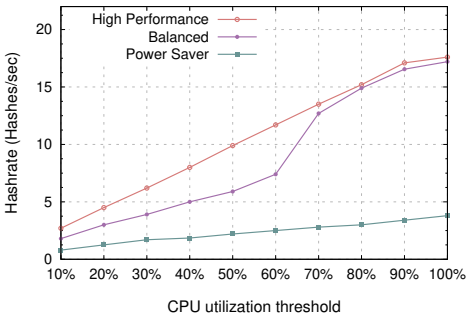

Fig. 5: Hashrate for different utilization levels and Power modes in the cryptojacking scenario. The OS may slow down clock speed of the victim's device, reducing up to 78.41% the computation power.

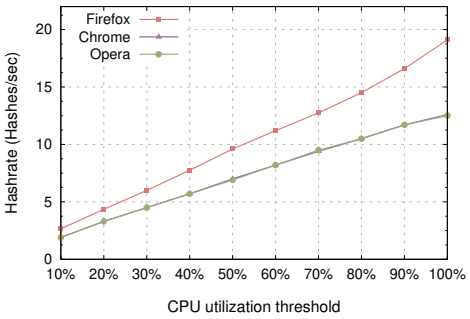

Fig. 6: Hashrate for different infected browsers and utilization levels in the cryptojacking scenario. Firefox browser can calculate up to 1.51× more hashes per second than Chrome and Opera.

conducted in a controlled environment, without reaching any host outside our local network (see Section VIII).

1) Abuse of network resources: In the first experiment, we measure the rate of HTTP requests that the MarioNet framework can achieve from a single browser. As described in Section V-A, the Puppeteer instructs the Servant to continuously send multiple HTTP requests to a remote server, via `XMLHttpRequest.send()`. Figure 3 shows the rate achieved for different browsers and different types of networks. To measure the rate, we ran `tcpdump` at the targeted server and captured all the incoming HTTP traffic. As can be seen in Figure 3, even devices over inferior network connections are capable of contributing a fair share in such a distributed attack (e.g., an average of 214 reqs/sec on Good3G networks). For high network bandwidth, i.e., 1 GbE, Opera tends to achieve higher rates (up to 1632 reqs/sec on average).

2) Abuse of computation power: The next experiment explores the computation capacity that the infected browsers can provide. Figure 4 presents the hashrate achieved when mining Monero coins in Chrome, for different CPU models and various utilization thresholds. As expected, the performance gain is highly affected by the equipped hardware. Specifically, we see that Intel i7-4790 can give 29% more hashes per second than Intel i5-5200U, when fully utilized.

After experimenting with different operating systems, we noticed that the different power mode characteristics they provide can drastically affect the sustained performance of CryptoNight execution. Figure 5 shows the performance achieved on a Windows 7 desktop computer that is equipped with an Intel i7-4790K, at 4.0GHz, under 3 different power modes (namely High Performance, Balanced and Power Saver). When fully utilized, the Power Saver mode forces the CPU to reduce the voltage and clock speed, which causes a decrease of up to 78.41% compared to the High Performance mode. In addition, in the Balanced mode, when CPU utilization exceeds 50% the operating system allows the CPU to run in full speed in order to cover the increased computation needs, thus verging the hashrate of High Performance mode.

Next, we explore how different infected browsers affect the computation gain of MarioNet. Figure 6 shows the hash-rate

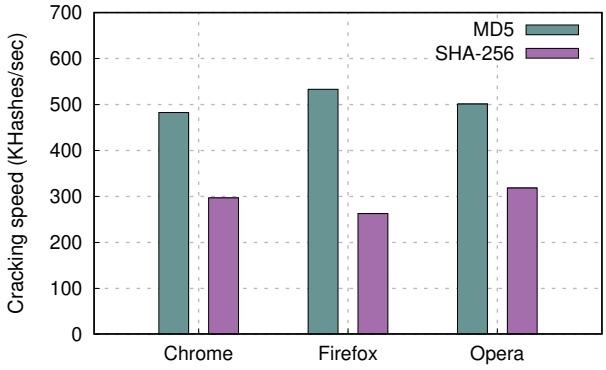

Fig. 7: Cracking speed of different browsers in the distributed password-cracking scenario. MarioNet can brute-force per victim around 500K MD5 hashes per second or around 300K SHA-256 hashes, irrespective of the infected browser.

achieved for different browsers, when using a Intel i7-5557U CPU. We observe that Firefox can calculate up to 34.55% more hashes per second than Chrome and Opera, which are both based on Chromium and the V8 JavaScript engine.

The earnings of an attacker that launches a distributed MarioNet-like Monero mining attack can be estimated with the following equation: $Earnings = ((total\_hashrate \times block\_reward)/current\_difficulty) \times time$. For this estimation we consider a scenario of an attacker that controls a website that attracts on average 10000 unique visitors per day, that the visitors of the malicious site have mid-range devices, and that the attacker utilizes only a single core of their devices at a utilization level of 60%-70% (i.e., user hashrate of 10 h/s). According to the current difficulty of Monero mining, the attacker will earn around 0.5 monero every 12 hours, which is easily achievable when considering the persistence characteristics of a MarioNet-like attack. It should be noticed though, that this is a very conservative estimation, since we only assume that the attacker infects a relatively small number of users, and that the victims' devices are only slightly utilized.

In our last experiment, we explore the performance sustained

for password cracking. Figure 7 plots the achieved rate for hashing 10-digit alphanumerical passwords on a brute-force manner, for both MD5 and SHA-256 algorithms. As we can see, all browsers achieve similar and comparable performance. This means that a single browser can brute-force around 500K MD5 hashes per second or 300K SHA-256 hashes, irrespective of the infected browser.

3) Persistent and Evasive abuse: In order to assess the persistence and evasiveness of our approach, we deliver MarioNet within a webpage destined to perform cryptojacking. Before fetching the webpage in a Chrome browser, (i) we open `tcpdump` and (ii) we deploy in our browser the following extensions/tools: Tamper Chrome HTTP capturing extension [20], Chrome's default DevTools, WebSniffer [2], and HTTP Spy [1] to explore in the real world, the stealthiness of MarioNet against state-of-the-art monitoring and blocking extensions. After fully rendering the webpage and planting the Servant, we close the associated browser tab. Then, from the Puppeteer, (iii) we push a cryptocurrency mining task to the Servant and let it run for 3 consecutive days. We see that although the Servant regularly communicated with the Puppeteer to obtain PoW tasks, as `tcpdump` correctly captured, *none* of the employed extensions was able to monitor *any* Servant-related traffic other than the very first GET request of the webpage, right before infection.

Comparison to state-of-the-art web botnets: In order to compare MarioNet with the state-of-the-art web-botnets, we load our password cracking algorithm in a set of web workers as described in related approaches [13], [54]. Given that these web-botnets run only for as long as the victim is surfing the webpage, they need to fully utilize the resources of the infected device in order to scrounge a meaningful gain from this short infection window. As a consequence, they usually occupy concurrently all system cores across the entire period of a website visit, which studies have shown that it is 1 minute on average [53].

In Figure 8, we plot the total number of SHA-256 hashes brute-forced by the 2 approaches in an infected browser for a period of 12 hours. For MarioNet, we measure two cases: (i) the best case, where the password cracker runs uninterruptedly in the victim's device, and (ii) the worst case, where along with the malicious computations there is heavy utilization from other processes too. In the second case, to simulate this heavy load, we concurrently run a multi-threaded pi digit calculator that fully utilizes all 8 system's cores. As we can see, although web-botnet utilizes greedily $8\times$ more resources, MarioNet due to its persistence, enjoys a higher efficiency after the 18th minute of an open browser, even under extreme heavy concurrent interference. Consequently, while until today, the business model of malicious websites were to deploy a web-botnet and find a way to keep the user on the website (by providing free movie streaming, online games or include pop-under windows [70]), with MarioNet it takes only a momentary visit to infect the user and take control of their browser.

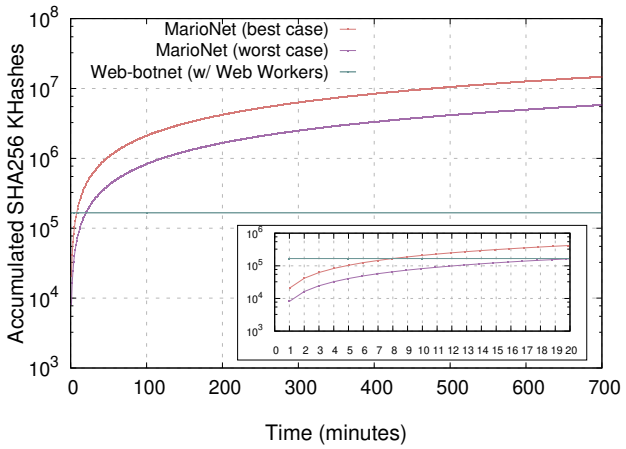

Fig. 8: Number of SHA256 hashes brute-forced by MarioNet and previous Web-botnet approaches that utilize Web Workers, in the lapse of time. The persistence of MarioNet makes a single infected browser compute hashes as long as the browser is open and thus be more efficient than opportunistic Web-Botnets.

VII. Defenses

In this section we examine potential defense mechanisms that could detect and mitigate MarioNet type of attacks. The goal is to determine whether it is feasible to detect the general methodology of the attack vectors that are opened through the misuse of the service worker mechanisms, rather than mitigating the specific use cases studied in this paper.

We present various defense strategies and discuss the corresponding tradeoffs they bring. We categorize the defenses in two classes: (i) those that can be deployed inside a vanilla browser (e.g., via an extension), and (ii) those that can be deployed in the host (through anti-virus tools, IDS/IPS, firewalls, etc.) or by modifying the browser.

A. In-browser Mechanisms

1) Restricting or Disabling Service Workers: Service workers have been introduced to enable rich offline user experience, such as periodic background synchronization and push notifications, embedded content fallback, caching, message relaying across pages, offline fallback, and user-side load balancing. Traditionally, these types of functionality required a native application. However, the window of opportunity for abuse by attackers, makes for a difficult tradeoff between rich user experience and security. In the case of MarioNet, disabling service workers could indeed prevent the persistence and stealthiness of in-browser malicious computation. Towards a similar direction, one could suggest to restrict the liveness of service workers, making it proportionate to the user presence in the website that hosts them (i.e., the service worker is suspended after the user leaves the website) or apply a time cap (i.e., a service worker gets terminated if it keeps running for an unreasonable amount of time). By doing so, the persistence characteristic of our attack will no longer be available. Service workers will still be able to intercept and modify navigation and resource requests, as well as cache resources, using the storage API for example, to allow applications to run even when the network is not available.

However, service workers were designed to provide important functionalities to long-running web applications (e.g., Google Docs, Gmail, Twitter, LinkedIn, Whatsapp web client) even after tab closing [50]. By forcing restrictions, service workers will not be able to provide the above background processing, thus significantly limiting the capabilities of contemporary web applications, resulting in a severe degradation

of user experience. To mitigate that, a better solution would be to selectively enable service workers only for some "trusted" websites, possibly via a browser extension that prevents the unconditional registration of service workers.

A step towards striking stealthiness is to disable the `eval()` family functions. By doing so, the attacker would need to ship the malicious functionalities together with the service worker, which would facilitate the signature-based content filtering browser extensions to detect them easier. Obviously, this would be an arm race between attackers and defenders, given the obfuscation and code scrambling techniques that are in use for similar cases. In addition, service workers can include minimal ISA emulators in order to execute malicious instructions received from the attacker. A more aggressive option would be to limit the functionality offered by service workers, by making only a subset of JavaScript available for use (e.g., allowing only the sending/receiving of data between the website and the server). Clearly, such a data-driven approach would require much more careful consideration and design.

2) Whitelists/Blacklists: Another possible defense strategy is to restrict the browser, with fine-grained policies, from fetching and deploying service workers. The simplest approach is the use of whitelists; i.e., service workers will be blocked, unless the domain of origin is whitelisted. These lists can initially include popular sites, which are typically considered more trusted, and enriched by web crawling and analysis platforms, such as [61], that perform web-wide analysis to detect malicious websites and JavaScript files.

3) Click to Activate: Another mitigation would be to require the user's permission for registration and activation of a service worker—similar to "Click to Play" mechanism [51] that disables by default plug-ins, such as Flash, Java, Silverlight and others. By doing so, the service worker functionality will be disabled by default, and users would need to explicitly give permission for the service worker to run. Currently, this user consent is needed only for the Push Notifications [44]. However, given the variation of attack vectors that can be achieved through a malicious service worker, we believe that explicit permission would raise user suspicion—in the same way it does for location, microphone, etc.—especially when browsing unreliable websites. One may say that these proposed permission-based defenses may be not practical to constitute the perfect mitigation for the presented attack. However, recent developments such as GDPR, mobile or browser permissions model etc., have demonstrated that user consent can be forced.

B. Host-based Approaches

1) Signature-based Detection: Traditional tools such as firewalls, anti-virus, and intrusion detection/prevention systems, are always a prominent methodology for the detection of malicious activities. The majority of these tools are typically using signatures to detect suspicious data or code that enters or leaves a user's computer. The creation of such signatures for the case of MarioNet may be trivial for some attack cases. For example, it could be easy to detect MarioNet messages that are exchanged between the service worker that lies in the browser and the back-end server, by monitoring the network traffic. A sophisticated attacker can obviously employ several techniques to raise the bar against signature-based detection mechanisms. For example, by installing end-to-end encryption with the back-end server can sufficiently hide the content of the messages. Given that a host-based approach can have full control of the client side though, the SSL connection can be intercepted to acquire the decrypted data. Besides, the messages still need to be transferred, which can be a good hint for detection mechanisms that are based on network flow statistics (e.g., number of packets exchanged, packet size). Even though covert channels and steganography may potentially help attackers, there are works that try to detect web-based botnets, by performing anomaly detection on features like communication patterns and payload size [30].

2) Behavioral Analysis and Anomaly Detection: A more drastic solution would be to develop techniques that try to detect suspicious behavior of JavaScript programs that are embedded in the web site or the service worker. Obviously, this would require more sophisticated analysis than simple fixed string searching and regular expression matching, due to the fact that the obfuscation of the malicious JavaScript code snippets can evade static analysis techniques. Instead, more advanced and complex analyzers should be used, such as the monitoring of the utilized resources or the behavioral analysis of the executed code. Even though this can be quite challenging, several works have been proposed in the past [25], [62], [27]. For instance, one of the first anomaly detection approaches is JaSPIn [62], which creates a profile of the application usage of JavaScript and enforces it later. IceShield [27] uses a linear decision function that differentiates malicious code from normal code based on heuristics for several attack types that apply code obfuscation. Finally, in [25] the authors audit the execution of JavaScript code, and compare it to high-level policies, in order to detect malicious code behavior. Although all these approaches are not trivial, they are a prominent step towards protecting against malicious JavaScript programs in general.

VIII. Discussion

Ethical considerations. In this paper, we implemented and deployed MarioNet in a strictly controlled environment. During our experimentation with attack scenarios, no user or web server outside this controlled environment were contacted or harmed in any way. As such, we constrained the evaluation of our system to a limited set of nodes, thus avoiding any attempt to measure our system on a larger scale, in the real world.

Attack difficulty. Based on our threat model (Section III-A) a MarioNet attack can be launched (a) intentionally, by a malicious or "shady" website; (b) unintentionally, by a hijacked or compromised website, or a website that includes a compromised library; and (c) through third-party dynamic content in iframes. Since our attack does not rely solely on a legitimate website or third-party library to be compromised, but it can also be performed by first-party websites that include such content, and also triggered by dynamic content in iframes, according to (a) and (c), it seems that such an attack can be practically launched quite easily (by actors with different incentives or intentions). Essentially, it requires only a contemporary web browser (all popular browsers are vulnerable—see Table II) and just accessing a webpage that provides such malicious content, either from first or third parties.

Attack impact. Aside from the cases of malicious or shady websites that can straightforwardly launch our attack, and web-

sites or third party libraries being compromised, the presented attack can be also launched by loading malicious third-party dynamic content in iframes residing in entirely legitimate websites. For instance, attackers can exploit the ad ecosystem's real-time auctioning mechanism to load their malicious content in iframes [57]. It has been already demonstrated in previous works [13] how programmatic ad delivery can be exploited to distribute malicious content. The ease of launching such an attack, and the ability of attackers to utilize legitimate, and more importantly, popular and trustworthy websites with possibly hundreds of thousands visitors, demonstrate the enormous impact our attack can potentially have. In addition, the stealthiness property of MarioNet, which can evade effectively monitoring extensions by design, and the fact that all major browsers are currently vulnerable to such attacks, as shown in the previous sections, make the number of potential victims even larger, and highlight the need for careful design and adequate protection mechanisms.

Registration of multiple service workers. Service workers are associated with specific scopes during registration and each service worker can only control pages that fall under its scope. If more than one service workers are registered (while the user is navigating throughout a website), then the browser enables only the service worker with the broader scope (typically the service worker registered at the root domain). However, during our experimentation we observed that a publisher can design its website on purpose so that multiple service workers can be registered in non-overlapping scopes (i.e., in file paths at the same level of the URI). As a consequence, this could allow MarioNet to have multiple Servants simultaneously active and utilize them for running its malicious tasks in multiple threads.

Cross-origin service workers. The cross-origin service worker (or foreign-fetch [60]) is an experimental feature of Chrome 54, to enable registration of third-party service workers. The motivation behind that, is to enable developers to implement advanced functionality, such as client-side caching of CDN-based third party content. However, this feature broadens the threat model of MarioNet-like approaches, enabling third-parties to misuse the service workers of the domains that include them. Even though this feature was discontinued one year after its announcement [68], mostly due to applicability issues, it still shows that such new functionality should be considered carefully in terms of security, before being applied.

Towards this direction, the aim of this work is to increase the awareness of developers and browser vendors about the provided powerful (but also potentially risky) capabilities of modern HTML, and hopefully lead to the deployment of restrictive policies that will adequately secure the user-side environments of future web applications.

IX. Related Work

Web browsers are a core part of our everyday life, being the door to the gigantic world of the web. As a result, they have become a valuable target for attackers, that try to exploit them in many different ways.

For instance, several approaches try to abuse the rich features of modern web applications, in order to form web-based botnets, the existence of which has seen a significant rise recently [33]. Provos et al. [61] present the threat of web-based malware infections, in which the infected browsers pull commands from a server controlled by the attacker. Contrary to traditional botnet-like attacks, web-based malware does not spread via remote exploitation but rather via web-based infection. In [4], the authors craft malicious webpages where users get infected upon visit. The attackers can then abuse users' browsers to perform attacks like DDoS, worm propagation, and node reconnaissance. Grossmann and Johansen [24] leverage ads to deliver malicious JavaScript to users, forcing browsers to establish connections with a victim server, thus performing a DoS attack. A major limitation of these approaches though, is that the corresponding malicious JavaScript snippets need to be embedded in the main webpage. As a result, long-running operations would block the rendering procedure and execution of the web application, making it practical only for short-lived attacks.

To overcome this limitation, many approaches started recently to use web workers—a feature that was introduced with HTML5. Web workers run as separate threads, and thus being isolated from the page's window. This allows the parallel execution of operations, without affecting the normal rendering of the web application, leading to the rise of more advanced web-based botnets. Kuppan [34] demonstrate this ability of using web workers to perform DDoS attacks. Rushanan et al. in [63], also use web workers to perform stealthy computations on the user side and launch not only attacks like DoS and resource depletion but also covert channel using CPU and memory throttling. Pellegrino et al. [58] also present different techniques to orchestrate web-based DoS attacks, by utilizing web workers among other HTML5 features, and provide an economic analysis and costs of browser-based botnets.

Similarly, Pan et al. [54] explore the possibility of using web workers for performing application-layer DDoS attacks, cryptocurrency mining and password cracking. Their results show that although DDoS attacks and password cracking are feasible and with comparable financial cost, cryptocurrency mining is not profitable for the attacker given the limited time a user spends in a website. Dorsey presented an in-browser botnet using web workers as well [13]. The user browser, after infection, participates in a swarm of bots performing various malicious operations like DDoS attacks, torrent sharing, cryptocurrency mining, and distributed hash cracking. To infect as many users as possible, Dorsey embedded his malware in a malicious advertisement and let the ad network to distribute it to the users browsers. Similar to MarioNet, all the above approaches do not require any software installation on the user side. However, the browser remains under the control of the attacker only for the duration that the user is browsing the malicious website, making it impractical for long-running botnet operations. Instead, MarioNet provides persistence that allows the attacker to perform malicious computations for a period longer than a website visit.

Besides the crypto-mining and crypto-jacking attacks, in which a website unintentionally hosts web-mining code snippets [40], [36], there are publishers that intentionally use mining to monetize their websites. Eskandari et al. analyze the existing in-browser mining approaches and their profitability [15]. Similar to web-based botnets, in-browser miners maintain a connection with a remote server to obtain PoW tasks and abuse

web workers to achieve the highest possible CPU utilization on the user side. However, the short website visiting times make the profitability of this approach questionable [65], [56]. MarioNet also uses crypto-jacking as a possible scenario, however instead of web workers we leverage service workers to enable an entity to gain much higher profits due to the provided persistence.

Finally, several attacks are based on malicious browser extensions that a user downloads and deploys in the browser [38], [59]. For instance, Liu et al. propose a botnet framework that exploits the browser extension update mechanism to issue batch commands [38]. By doing so, they are able to perform DDoS attacks, spam emails and passwords sniffing. Similarly, Perrotta et al. exploit the over-privileged capabilities of browser extensions to check the effectiveness of botnet attacks in contemporary desktop and mobile browsers [59]. Their results show that different attacks are feasible in different browsers. A major difference of these approaches with MarioNet, is that all the above approaches require the installation of software (i.e., browser extension) on the user side.

X. Conclusion

In this work, we presented MarioNet: a novel multi-attack framework to allow persistent and stealthy bot operation through web browsers. Contrary to traditional botnet-like approaches, our framework does not require any installation of malicious software on the user side. Instead, it leverages the existing technologies and capabilities provided by HTML5 APIs of contemporary browsers.

We demonstrate the effectiveness of this system by designing a large set of attack scenarios where the user's system resources are abused to perform malicious actions including DDoS attacks to remote targets, cryptojacking, malicious/illegal data hosting, and darknet deployment. Two important characteristics of MarioNet, that further highlight the severity of the aforementioned attacks, is that it provides persistence, thus allowing an attacker to continue their malicious computation even after the user navigates away from the malicious website. In addition, MarioNet provides evasiveness, performing all operations in a completely stealthy way, thus bypassing the existing in-browser detection mechanisms.

Essentially, our work demonstrates that the trust model of web, which considers web publishers as trusted and allows them to execute code on the client-side without any restrictions is flawed and needs reconsideration. Furthermore, this work aims to increase the awareness regarding the powerful capabilities that modern browser APIs provide to attackers, and to initiate a serious discussion about implementing restrictions while offering such capabilities that can be easily abused.

Acknowledgments

We thank our shepherd, Adam Doupé, and the anonymous reviewers for their valuable feedback. The research leading to these results has received funding from European Union's Marie Sklodowska-Curie grant agreement 690972 (PROTASIS); the Horizon 2020 Research & Innovation Programme under grant agreements 786669 (REACT), 740787 (SMESEC), 700378 (CIPSEC), and 786890 (THREAT-ARREST); and by the National Science Foundation (NSF) under grant CNS-1617902. The paper reflects only the authors' view and the Agency and the Commission are not responsible for any use that may be made of the information it contains.

References

[1] "HTTP spy," https://chrome.google.com/webstore/detail/http-spy/agnoocojkneiphkobpcfoaenhpjnmifb.

[2] 5ms.ru, "Web Sniffer," https://chrome.google.com/webstore/detail/web-sniffer/ndfgffclcpdbgghfgkmooklaendohaef.

[3] Accengage, "Push notification benchmark press release 2017," https://www.accengage.com/press-release-accengage-releases-the-push-notification-benchmark-2017-including-for-the-first-time-web-push-facebook-messenger-metrics-in-addition-to-stats-for-mobile-apps/, 2017.

[4] S. Antonatos, P. Akritidis, V. T. Lam, and K. G. Anagnostakis, "Puppetnets: Misusing web browsers as a distributed attack infrastructure," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 12:1–12:38, Dec. 2008.

[5] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines." in *Proceedings of Annual Network and Distributed System Security Symposium*, ser. NDSS'15, 2015.

[6] P. Baldi, P. Frasconi, and P. Smyth, *Modeling the Internet and the Web: Probabilistic Methods and Algorithms*. Wiley Online Library, 2003, ch. 7. Modeling and Understanding Human Behavior on the Web.

[7] D. Bradbury, "Scrypt-based miners and the new cryptocurrency arms race," https://www.coindesk.com/scrypt-miners-cryptocurrency-arms-race/, 2013.

[8] C. Cimpanu, "Firefox working on protection against in-browser cryptojacking scripts," https://www.bleepingcomputer.com/news/software/firefox-working-on-protection-against-in-browser-cryptojacking-scripts/, 2018.

[9] Cloudflare, "Slowloris DDoS Attack," https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/.

[10] Coinhive, "A Crypto Miner for your Website," https://coinhive.com.

[11] CryptoNote Technology, "Egalitarian proof of work," https://cryptonote.org/inside.php#equal-proof-of-work, 2015.

[12] M. Dittus, J. Wright, and M. Graham, "Platform criminalism: The 'last-mile' geography of the darknet market supply chain," in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW, 2018.

[13] B. Dorsey, "Browser as botnet, or the coming war on your web browser," https://medium.com/@brannondorsey/browser-as-botnet-or-the-coming-war-on-your-web-browser-be920c4f718, 2018.

[14] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016.

[15] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A first look at browser-based cryptojacking," *CoRR*, vol. abs/1803.02887, 2018.

[16] F. Aboukhadijeh and WebTorrent, LLC., "Torrents on the web," https://webtorrent.io/, 2017.

[17] D. Fifield, N. Hardison, J. Ellithorpe, E. Stark, D. Boneh, R. Dingledine, and P. Porras, "Evading censorship with browser-based proxies," in *International Symposium on Privacy Enhancing Technologies Symposium*, ser. PETS, 2012.

[18] D. Goodin, "Cryptojacking craze that drains your cpu now done by 2,500 sites," https://arstechnica.com/information-technology/2017/11/drive-by-cryptomining-that-drains-cpus-picks-up-steam-with-aid-of-2500-sites/, ArsTechnica.

[19] Google Developers, "Introduction to service worker," https://developers.google.com/web/ilt/pwa/introduction-to-service-worker, 2018.

[20] Google Open Source, "Tamper Chrome browser application," https://chrome.google.com/webstore/detail/tamper-chrome-extension/hifhgpdkfodlpnlmlnmhchnkepplebkb, 2017.

[21] P. Gregoire, "Public STUN server list," https://gist.github.com/mondain/b0ec1cf5f60ae726202e, 2016.

[22] G. Grigoreva, "What is a popunder ad & how to use it (explained)," https://www.mobidea.com/academy/popunder-ad/, 2018.

[23] I. Grigorik, "Browser APIs and protocols: WebRTC," https://hpbn.co/webrtc/, 2013.

[24] J. Grossman and M. Johansen, "Million Browser Botnet," Presentation at Black Hat USA, 2013.

[25] O. Hallaraker and G. Vigna, "Detecting Malicious JavaScript Code in Mozilla," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS, 2005.

[26] S. Han, "Snowflake Technical Overview," https://keroserene.net/snowflake/technical/, The Tor Project, 2017.

[27] M. Heiderich, T. Frosch, and T. Holz, "IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID, 2011.

[28] B. Hillmer, "URL redirect: Breaking out of an iframe," https://help.surveygizmo.com/help/break-out-of-iframe, 2017.

[29] E. Homakov, "Building botnet on serviceworkers," https://sakurity.com/blog/2016/12/10/serviceworker_botnet.html, 2016.

[30] F.-H. Hsu, C.-W. Ou, Y.-L. Hwang, Y.-C. Chang, and P.-C. Lin, "Detecting Web-Based Botnets Using Bot Communication Traffic Features," *Security and Communication Networks*, vol. 2017, 2017.

[31] Juniper Research, "Ad fraud to cost advertisers $19 billion in 2018, representing 9% of total digital advertising spend," https://www.juniperresearch.com/press/press-releases/ad-fraud-to-cost-advertisers-$19-billion-in-2018, 2018.

[32] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam, "A taxonomy of botnet behavior, detection, and defense," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 898–924, 2014.

[33] KrebsOnSecurity, "The Rise of Point-and-Click Botnets," https://krebsonsecurity.com/tag/web-based-botnets/.

[34] L. Kuppan, "Attacking with html5," Presentation at Black Hat, 2010.

[35] S. Lekies, B. Stock, M. Wentzel, and M. Johns, "The Unexpected Dangers of Dynamic JavaScript," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. USENIX Security, 2015.

[36] J. Leyden, "Real mad-quid: Murky cryptojacking menace that smacked ronaldo site grows," http://www.theregister.co.uk/2017/10/10/cryptojacking/, The Register, 2017.

[37] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq, "Hunting the Red Fox Online: Understanding and Detection of Mass Redirect-Script Injections," in *2014 IEEE Symposium on Security and Privacy*, ser. IEEE S&P, May 2014.

[38] L. Liu, X. Zhang, and S. Chen, "Botnet with browser extensions," in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, ser. PASSAT/SocialCom, 2011.

[39] L. McAfee, "Mcafee secure safe browsing," https://www.mcafeesecure.com/safe-browsing, 2018.

[40] K. McCarthy, "CBS's Showtime caught mining crypto-coins in viewers' web browsers," http://www.theregister.co.uk/2017/09/25/showtime_hit_with_coinmining_script/, The Register, 2017.

[41] D. A. McGrew and K. Norrman, "The secure real-time transport protocol (srtp)," 2004.

[42] MDN web docs, "Mozilla port blocking," https://developer.mozilla.org/en-US/docs/Mozilla/Mozilla_Port_Blocking, 2014.

[43] ——, "Cross-Origin Resource Sharing (CORS)," https://developer.mozilla.org/en-US/docs/Web/http/CORS, 2018.

[44] ——, "Push API," https://developer.mozilla.org/en-US/docs/Web/API/Push_API, 2018.

[45] ——, "ServiceWorkerRegistration.periodicSync," https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/periodicSync, 2018.

[46] ——, "Using Web Workers," https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers, 2018.

[47] ——, "WebSockets," https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API, 2018.

[48] ——, "XMLHttpRequest," https://developer.mozilla.org/en-US/docs/Web/API/XMLhttpRequest, 2018.

[49] N. Modadugu and E. Rescorla, "Datagram transport layer security," 2006.

[50] Mozilla Corporation, "Serviceworker cookbook," https://serviceworke.rs/, 2018.

[51] Mozilla Support, "Why do I have to click to activate plug-ins?" https://support.mozilla.org/en-US/kb/why-do-i-have-click-activate-plugins, 2018.

[52] MWR InfoSecurity, "Distributed hash cracking on the web," https://labs.mwrinfosecurity.com/blog/distributed-hash-cracking-on-the-web/, 2012.

[53] J. Nielsen, "How long do users stay on web pages?" https://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/, Nielsen Norman Group, 2011.

[54] Y. Pan, J. White, and Y. Sun, "Assessing the threat of web worker distributed attacks," in *Communications and Network Security (CNS), 2016 IEEE Conference on*, ser. CNS, 2016.

[55] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, "The Long-Standing Privacy Debate: Mobile Websites vs Mobile Apps," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW, 2017.

[56] P. Papadopoulos, P. Ilia, and E. P. Markatos, "Truth in web mining: Measuring the profitability and cost of cryptominers as a web monetization model," *CoRR*, vol. abs/1806.01994, 2018.

[57] P. Papadopoulos, N. Kourtellis, P. R. Rodriguez, and N. Laoutaris, "If You Are Not Paying for It, You Are the Product: How Much Do Advertisers Pay to Reach You?" in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC, 2017.

[58] G. Pellegrino, C. Rossow, F. J. Ryba, T. C. Schmidt, and M. Wählisch, "Cashing Out the Great Cannon? On Browser-Based DDoS Attacks and Economics," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, ser. WOOT, 2015.

[59] R. Perrotta and F. Hao, "Botnet in the browser: Understanding threats caused by malicious browser extensions," *CoRR*, vol. abs/1709.09577, 2017. [Online]. Available: http://arxiv.org/abs/1709.09577

[60] J. Posnick, "Cross-origin Service Workers: Experimenting with Foreign Fetch," https://developers.google.com/web/updates/2016/09/foreign-fetch, 2016.

[61] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The Ghost in the Browser Analysis of Web-based Malware," in *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, ser. HotBots, 2007.

[62] P. Raman, "JaSPIn: JavaScript based Anomaly Detection of Cross-site scripting attacks," Ph.D. dissertation, Carleton University, 2008.

[63] M. Rushanan, D. Russell, and A. D. Rubin, "Malloryworker: stealthy computation and covert channels using web workers," in *International Workshop on Security and Trust Management*. Springer, 2016, pp. 196–211.

[64] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, "Busting frame busting: a study of clickjacking vulnerabilities at popular sites," *IEEE Oakland Web*, vol. 2, no. 6, 2010.

[65] K. Sedgwick, "Mining Crypto In a Browser Is a Complete Waste of Time," https://news.bitcoin.com/mining-crypto-in-a-browser-is-a-complete-waste-of-time/.

[66] T. Soulo, "How many websites are mining cryptocurrency? we analyzed 175m+ domains to find out." https://ahrefs.com/blog/cryptomining-study/.

[67] The European Union Agency for Network and Information Security (ENISA), "Malvertising," https://www.enisa.europa.eu/publications/info-notes/malvertising, 2016.

[68] The World Wide Web Consortium (W3C), "Remove foreign fetch," https://github.com/w3c/ServiceWorker/issues/1188, 2017.

[69] J. Topf, "Vulnerability note vu#476267," https://www.kb.cert.org/vuls/id/476267, 2001.

[70] L. Tung, "Windows: This sneaky cryptominer hides behind taskbar even after you exit browser," https://www.zdnet.com/article/windows-this-sneaky-cryptominer-hides-behind-taskbar-even-after-you-exit-browser/, 2017.

[71] S. Weagle, "Short, low-volume DDoS attacks pose greatest security and availability threat to businesses," https://www.itproportal.com/features/short-low-volume-ddos-attacks-pose-greatest-security-and-availability-threat-to-businesses/.

[72] World Wide Web Consortium (W3C), "High resolution time level 2," https://www.w3.org/TR/hr-time-2/, 2018.