Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage

Soroush Karami
University of Illinois at Chicago
skaram5@uic.edu

Panagiotis Ilia
University of Illinois at Chicago
pilia@uic.edu

Jason Polakis
University of Illinois at Chicago
polakis@uic.edu

Abstract*—Service workers are a powerful technology supported by all major modern browsers that can improve users' browsing experience by offering capabilities similar to those of native applications. While they are gaining significant traction in the developer community, they have not received much scrutiny from security researchers. In this paper, we explore the capabilities and inner workings of service workers and conduct the first comprehensive large-scale study of their API use in the wild. Subsequently, we show how attackers can exploit the strategic placement of service workers for history-sniffing in most major browsers, including Chrome and Firefox. We demonstrate two novel history-sniffing attacks that exploit the lack of appropriate isolation in these browsers, including a non-destructive cache-based version. Next, we present a series of use cases that illustrate how our techniques enable privacy-invasive attacks that can infer sensitive application-level information, such as a user's social graph. We have disclosed our techniques to all vulnerable vendors, prompting the Chromium team to explore a redesign of their site isolation mechanisms for defending against our attacks. We also propose a countermeasure that can be incorporated by websites to protect their users, and develop a tool that streamlines its deployment, thus facilitating adoption at a large scale. Overall, our work presents a cautionary tale on the severe risks of browsers deploying new features without an in-depth evaluation of their security and privacy implications.

I. Introduction

As the Web continues to evolve, browsers have become complex application platforms that mediate a significant part of our online activities. With web apps continuously introducing novel functionality to increase user engagement, browsers deploy new APIs and technologies to support such initiatives. As a result, modern web browsers often integrate new technologies and mechanisms that introduce novel attack vectors with significant security and privacy implications [35], [36], [47], [27]. As such, it is crucial that the security community conducts in-depth investigations of the risks introduced by emerging browser features.

Service workers (SWs) are such an emerging technology, gaining significant traction within the browser ecosystem [48] as they provide functionality that bridges the gap between web apps and applications that run natively on a user's device. Their capability to run in the background independently of the web application's page, coupled with browser APIs, enables a rich set of features that were previously out of the realm of capabilities of web apps (e.g., push notifications, background syncing, programmatically-driven caching). To better understand their prevalence and use in the wild we develop an automated testing framework for the dynamic analysis of SWs. Our system, which is built on top of an instrumented version of Chromium, automatically visits websites, extracts their SWs, and analyzes their use of APIs. We leverage our framework for studying how SW APIs are used in the top one million Alexa sites, and identify over 30K domains currently setting SWs and taking advantage of their capabilities.

Subsequently, we conduct an empirical exploration of the privacy threats that the presence of SWs poses to users and identify several novel privacy-invasive attacks. We demonstrate how one of the cornerstones of SW capabilities (that of prefetching and caching resources) can be misused for history sniffing attacks. We design and implement two attack techniques that use `iframes` on a third-party website to fetch cross-domain resources, resulting in the activation of other origins' SWs. Then, by using the information provided by the Performance API or by measuring those resources' loading times, our techniques can detect the presence of a SW in the user's browser, indicating that the user has previously visited a particular website. With the use of `iframes` for the activation of SWs, our techniques essentially circumvent browsers' site isolation mechanisms. Our attacks, which work on all major browsers that implement SWs except Safari, are more practical and robust than prior history sniffing attacks: (i) each SW's cache is programmatically managed by the SW and not subject to the browser's common cache eviction policy that affects prior attacks, (ii) our Performance-API-based attack is not subject to false positives or negatives and (iii) it is also non-destructive, as opposed to prior cache-based techniques. Furthermore, we have built a tool that automatically identifies resources appropriate for our attacks on a target domain, allowing us to conduct the largest experiment for evaluating domains' susceptibility to history sniffing to date.

While our main focus is inferring which websites a user has visited, our methodology also enables other forms of privacy-leakage attacks. We present a series of such use cases that illustrate how SW-specific behavior enables attacks that infer sensitive application-level information. First, we show that cached resources can reveal more fine-grained information about which specific pages have been visited in sensitive domains like an e-shop with sexual paraphernalia and a portal for searching people. Second, we demonstrate how post-login

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23104
www.ndss-symposium.org

resource caching can allow attackers to infer that users have an account or are currently logged into a given website (e.g., Tinder, Gab). Third, we outline how attackers can use WhatsApp to uncover if a target user is part of the victim's social circle. In certain cases, a more resourceful attacker can even infer if a visitor is part of a given WhatsApp group, which could enable a (partial) deanonymization attack.

Overall, our research demonstrates that the strategic placement of SWs for handling HTTP requests, combined with access to functionality-rich APIs and the lack of proper isolation, presents several opportunities for misuse that result in severe privacy loss for users. As SWs continue to gain significant traction in the web development community, the threat that they pose to users will only increase over time. As such, we have set remediation efforts in motion by disclosing our findings to all vulnerable browser vendors and web services. Accordingly, the first attack that uses the Performance API for determining if a resource is fetched through a SW, has been addressed by most notified browsers at the time of this writing.

Alarmingly, the underlying issues that enable our attacks lie in browsers' site isolation mechanisms that allow `iframes` in third-party websites to activate other parties' SWs and use them for retrieving resources. As such, the appropriate solution to address the root cause of our attacks is to redesign site isolation mechanisms and prevent the use of SWs in third-party websites. This, however, is not trivial to implement and requires significant effort. To that end, we propose an access control mechanism for restricting websites' SWs from being activated and used by other sites. To assist web developers in implementing the proposed countermeasure and facilitate its adoption at a large scale, we will open source our tool for automatically incorporating appropriate checks that implement the desired access control policy.

In summary, our research contributions are:

- We present an overview of the inner workings and capabilities of SWs and develop a framework for dynamically analyzing them. Subsequently we conduct a large-scale measurement study on the use of SWs and provide the first, to our knowledge, comprehensive analysis of their API usage in the wild. We will publicly share our data to facilitate further research.
- We introduce a series of novel privacy attacks that exploit the placement and functionality of SWs. We present two practical and robust history sniffing attacks (including a non-destructive version) that exploit the lack of appropriate isolation in browsers, and we conduct the largest study on the applicability of such attacks to date.
- We present a series of use cases that highlight how SWs can be misused for more privacy-invasive attacks that infer application-level information.
- The severity and impact of our attacks has driven browsers to modify their systems and has prompted Chromium's exploration of a redesign of their site-isolation mechanism, which poses significant challenges. To better protect users in the short term, we will publicly release a tool that streamlines the deployment of an access control mechanism that can prevent our attacks.

II. Background and Threat Model

As web browsers continue to mediate a significant portion of our online activities, there is an ongoing trend of pushing functionality that is typically associated with native apps to cloud and web applications (e.g., collaborative document writing in Overleaf). Browsers are in a constant state of evolution, with new functionality-rich APIs being deployed. One such recent feature are *Service Workers (SWs)*, which aim to fill the gap between native and web apps. Traditionally, web apps have lacked certain capabilities common to native apps, preventing them from reaching their full potential. Functionality such as sending push notifications, syncing in the background, working offline, and pre-caching for optimization, is now within the realm of capabilities of modern web apps with SWs.

Service workers run independently of the web application and do not have access to the DOM tree, i.e., cannot directly read a page's content. They are also event-driven scripts and, unlike other workers, can exist without a reference from the document. That means that they can become idle when not in use and restart when next needed (on an event). Specifically, there are six events that service workers can respond to:

Install event. Once the browser registers a service worker, the `install` event occurs. This is, conceptually, a similar process to installing a native application. With this event service workers go through a preparation process that establishes them for subsequent use, e.g., by populating an IndexedDB and caching necessary assets.

Activate event. This event is sent after the `install` event completes and the SW is activated. During this process, typical actions include cleaning up old caches and anything else associated with a previous version of the SW (i.e., after a website pushes an updated version of its SW).

Push event. In this event, SWs use `push API` and `notifications API` to provide push notification for web apps. The `push API` allows the SW to receive messages pushed from the server. The `notifications API` provides a method for integrating push notifications from web apps into the underlying operating system's native notification system. Servers can send push messages at any time, even when the web application is not running, and remotely activate the SW. Push functionality, however, requires explicit user approval.

Message Event. Host web applications cannot access their SWs directly. To communicate with their SWs they need to use the `postMessage()` method to send data. For receiving the data SWs must implement a `message` event listener.

Sync event. The `sync API` of SWs allows use of web application functionality even when the device is in offline mode, and defer the syncing of user actions until the device has stable Internet connectivity (e.g., offline email composition in Gmail). This API also allows servers to periodically push updates to the SWs; the next time users open the web application, they can use updated, cached data.

Fetch event. This functionality allows SWs to pose as a client-side programmable proxy between the web application and the outside world, and gives websites fine-grained control over network requests. For example, a developer can control the caching behavior of requests for the site's HTML code and treat them differently than image resources fetched by the website. A FetchEvent is fired every time a web application's resources are requested.

Using these events allows developers to create web applications that are reliable, fast, and engaging. However, apart from all the usability benefits that the Fetch API presents, it also poses a significant threat to users as we detail in Section IV.

A. Caching Files with Service Workers.

Caching resources can significantly improve performance as it will result in the app's content being loaded faster under a variety of network conditions. Unlike the typical browser cache (HTTP Cache), the Cache Storage API gives the SW full programmatic control of the cache. It allows SWs to store assets delivered by responses and keyed by their requests.

A common caching strategy implemented with SWs is to pre-cache assets during installation. During the first visit and the SW's install event, assets like HTML, CSS, JavaScript, images, etc., are downloaded and inserted into the cache. Listing 1 shows an example of such a pre-caching strategy.

```
this.addEventListener('install',function(event){
    event.waitUntil(
        caches.open('v1').then(function(cache){
            return cache.addAll([
                'index.html',
                'offline.html',
                'static/style.css',
                'static/app.js',
                'images/logo.jpg',
                'images/icon.png'
            ]); }) ); });
```

Listing 1. Pre-caching implemented in the install event.

To use cached assets the SW needs to have a FetchEvent listener. A FetchEvent fires every time any resource controlled by a service worker is fetched. The cache contains a list of requests and matching responses, and the SW uses caches.match(event.request) to match the requested resources to the corresponding ones that are available in the cache. The respondWith(response) method is used to send a response back to the web application. Listing 2 shows how a SW can uses the cache to provide the requested resources. It first asks the cache to look up the request and return the response. If the file is not in the cache, it will then try to fetch it from the network.

```
self.addEventListener('fetch',function(event){
  event.respondWith(
    caches.match(event.request)
    .then(function(response){
       return response || fetch(event.request);
    }) ); });
```

Listing 2. Service worker uses cache in the FetchEvent.

Another common caching strategy implemented with SWs is to provide offline access. In cases where the requested resources are not available in the cache and the network is unreachable, SWs can intercept requests and provide alternative resources. Listing 3 shows the implementation of this strategy. If the user is offline, the SW can detect it and respond to all requests with an HTML page that has already been stored in the cache.

```
self.addEventListener('fetch', (event) => {
    if (!self.navigator.onLine) {
        event.respondWith(
            caches.match("offline.html")
        ); } });
```

Listing 3. Responding to requests with offline pages.

The FetchEvent is fired for navigation in the SW's scope. The default scope is the path to the SW file and extends to all directories below it. If the SW script is located in the root directory, the SW will control requests from all files in this domain. It is also possible to set an arbitrary scope during registration, but a SW cannot have a scope "above" its own path. For example, in Listing 4 the scope of the SW is set to /app/, which means that the SW will control requests from pages in the /app/ directory and below.

```
navigator.serviceWorker.register(
    '/service-worker.js',
    { scope: '/app/'}
);
```

Listing 4. Setting arbitrary scope for a service worker.

Despite the benefits of SWs for web apps' performance, they are not without cost. A SW can take time to start up if it is not already running; this can happen if a user has not visited the web app in a while. The time it takes a SW to boot up depends on the user's device; according to [54] it takes $20 - 100ms$ for desktop users and $> 100ms$ for mobile users.

B. SW Cache Storage vs. Browser Cache

There are several differences between the traditional browser cache and the SW cache, which result in our attacks being more robust and impactful than previous cache-based history-sniffing attacks. In general, resources are stored in the SW cache storage during the installation of SWs (upon first visiting a website). On the contrary, resources are stored in the browser cache during navigation of websites. While the browser cache relies on HTTP headers or the browser's built-in heuristics [40] to manage cached resources, a code-driven approach (CacheStorage API) is used for SW cache storage. Next, the resources in the SW cache storage are not ephemeral in nature; there are no automatic, built-in expiration algorithms or freshness checks, and once a SW stores an item in the cache storage, it will persist until its code explicitly removes it.

Furthermore, SWs ignore Cache-Control headers when caching data [10]. As such, the attacks that we present in this paper are more robust and practical than prior attacks, where target resources could be evicted by the browser, thus, removing the artifacts left by the user's browsing activity. Also, this storage space can grow to considerable size [12], with Chrome and Firefox allowing up to 6% and 10% of the device's free disk space per origin, respectively. As such, websites can aggressively cache resources without the need to remove cached resources due to space constraints. Moreover, the idiosyncrasies of SW-based caching enable a non-destructive attack (i.e., it can be performed multiple times), which is not the case with typical cache-based attacks.

Finally, we note that recent versions of major browsers separate the browser cache based on the origin, to prevent

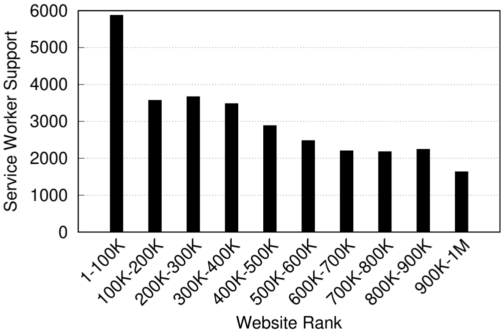

Fig. 1. Number of domains installing SWs, grouped based on their popularity.

documents from one origin from knowing whether a resource from another origin was cached [2]; this prevents previously proposed history-sniffing attacks that use the browser cache.

C. Threat Model

For the attacks presented in this paper, we follow a threat model typically used in prior work on history sniffing and other privacy-invasive attacks. We assume that the attacker is able to execute JavaScript code in the user's browser. For ease of presentation we assume that the user visits a website controlled by the attacker. In practice, however, our attacks could be deployed at an even larger scale (e.g., through ads).

III. Service Workers in the Wild

In this section we provide a large-scale measurement study on the prevalence of SWs in the top 1M Alexa websites and explore which API features these SWs currently implement.

Methodology. To automate the process of identifying websites that have SWs, we use Selenium to drive Chromium. Upon first opening a page that has a SW, it takes a few seconds for the browser to register the SW and for the `install` event handler to complete before it is ready to use. Then, for the SW to be able to control the page, either the user must refresh (or revisit) the page or the SW must call `self.clients.claim()`. In our experiments we open each page in a fresh browser instance, wait for 10 seconds to ensure that the SW is ready to use, and then refresh the page to bring it under the control of the SW. Finally, we inject JavaScript code that reads the `navigator.serviceWorker.controller` [14] object. If the website is using a SW this object contains the script URL and status of the SW; otherwise the object is `null`.

While detecting the presence of a SW is straightforward, identifying which features and functionalities each SW implements presents a considerable challenge. A simplistic approach would be to statically inspect the SWs' code. However, the prevalence of obfuscation and minification [44] would significantly affect the correctness of such an approach. As such, we instrument the Chromium browser and build a dynamic analysis tool that logs the SW-specific API calls. To do so we modify the following Blink modules: *service_worker*, *cache_storage*, *background_sync*, *notifications*, and *push_messaging*. Each of these modules has a set of functions that implement the APIs that a SW can call. We add a logger to each one of these functions to record which one is being called and the arguments that are passed. The arguments that are collected from the API calls also include the URLs that are intercepted by the `FetchEvent`. In Section IV-D we describe how we utilize our instrumented browser to collect these intercepted URLs and evaluate their suitability for our attacks. Furthermore, since the web push API requires the user's permission for sending notifications, we modify its code to grant this permission to all websites by default.

TABLE I. Service Worker functionality in the Alexa top 1M.

Functionality	Websites with SWs Landing page	 w/ Additional pages
Caching	8,559	9,446
Fetch	8,895	9,900
Web Push	23,227	25,457
Sync	90	94
SW to Client Message	8,339	8,844
Client to SW Message	10,593	11,796
importScripts	22,380	23,706

Dataset. Using our instrumented browser we visit the top 1M Alexa websites (12/2019-02/2020) and identify SWs on 30,229 sites; we break down the relative popularity of the domains based on their Alexa rank in Figure 1. As one might expect, the most popular websites are more likely to install a SW, as they improve the user's browsing experience.

SW functionality. By inspecting which APIs are called by each SW we can infer the features and functionalities that they implement. Based on the assumption that a SW is typically installed on the landing page, we first visit the landing page of each website and if a SW is found we then randomly visit 10 additional pages under the same domain. During this process our instrumented browser records information about all the API calls. Our findings are presented in Table I. Since a SW may exhibit different types of functionality, the same domain may be counted in multiple categories. We provide a complete list of all the API calls and their mapping to each type of functionality in the Appendix.

Overall, we found 9,446 websites that implement caching functionality in their SWs. These have at least one method of the `Cache` or `CacheStorage` interfaces, such as `put`, `addAll`, `match`, `open`, etc. 8,559 of those websites have a SW implementing caching on the landing page. Fetch functionality is provided by the `FetchEvent` interface and is found on 9,900 websites, while 8,895 of those implement Fetch in the SW controlling their landing pages. API calls of this interface are `request`, `respondWith` etc. For having Web Push, websites use the `PushSubscription`, `PushManager`, and `Notification` interfaces. We observed API calls related to Web Push on 25,457 websites. Only 94 websites use Sync; this is most likely due to the fact that the `SyncEvent` and `SyncManager` interfaces have not been standardized yet.

Websites can communicate with their SWs with the post messaging APIs. We identified 11,796 websites that use `ServiceWorker.postMessage` API for sending a message, and 8,844 websites that use `client.postMessage` in their SWs to send a message to clients (i.e., pages or iframes which are currently open and within the SW's scope).

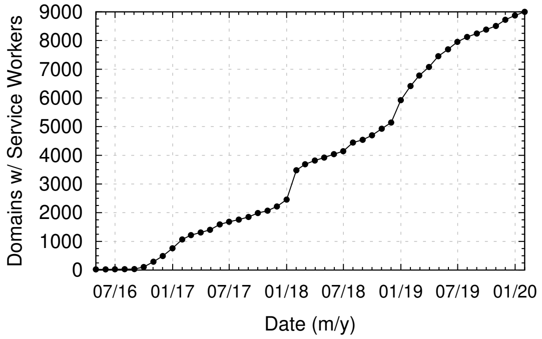

Fig. 2. Longitudinal view of the deployment of service workers.

TABLE II. MOST POPULAR SCRIPTS IMPORTED BY SERVICE WORKERS

Script	Websites
OneSignal [5]	7,027
Workbox [8]	2,376
Firebase Messaging [4]	1,534
sendpulse [7]	988
pushprofit [6]	846

Messages are usually exchanged for requests or notifications, e.g., a SW sending a message to clients notifying them about a change in the SW so clients can refresh and get the update.

Websites can also use the `importScript` API to include another script into their SW. In contrast to the main SW script file, the included script is not limited to the website's domain; it can be retrieved from a different origin. In fact, it is a common practice for websites to import third-party scripts that implement Web Push as a service. Since many websites use *importScripts* to include external libraries, we collect and further analyze these scripts. By comparing the origin of the websites and imported scripts, we found 20,569 websites that use third-party scripts. Among the 2,107 unique scripts that we observed, 26 are used by more than 100 different websites. In Table II we present the top 5 scripts imported by the SWs that we identified. In their majority these scripts provide Web Push Notification functionality. Another popular script is Workbox, a library developed by Google for improving performance and adding offline support to web applications.

Overall, as can be seen in Table I, the number of websites that implement each functionality increases when we also visit additional pages. Interestingly, we found 913 websites with at least one additional SW being registered when visiting internal pages. In websites with multiple SWs, the one with the most narrow scope is activated when the user visits a page. This is either because those websites implement a different functionality in an additional SW (that is absent from the SW that is controlling the landing page), or because they only have one SW but some of its functionalities are only triggered when visiting a particular internal page. In the remainder of the paper we only consider domains that implement fetch and caching on the landing page when detecting susceptible resources for our attacks, as those can be directly identified by attackers.

Historical trends. Next, we explore how adoption of SWs has evolved over time. Prior work has demonstrated how the Internet Archive provides a unique looking glass for studying the evolution of practices on the web [34], [49], [41]. Similarly, we randomly select 9,000 of the domains detected in our study and leverage the Internet Archive for obtaining historical information. For each website, we identify the month where a SW first appeared in a snapshot. As shown in Figure 2, the oldest SWs identified in our sampled domains were from May 2016 when 24 domains first installed a SW. Starting from October 2016 we can see a steady increase in the number of SWs being added, with small jumps in the beginning and end of 2018. Based on reports on the emergence of progressive web apps [11], and projecting from our findings, it is safe to expect that SW adoption will continue to rise. As such, it is imperative that the security community continues to explore them and proactively identifies new attacks that become possible.

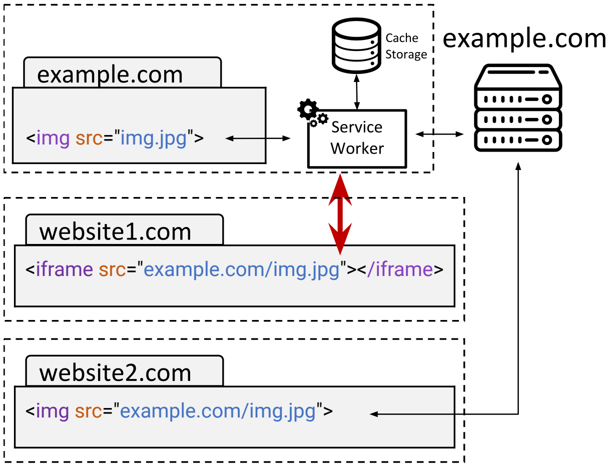

Fig. 3. Example site with a SW implementing a `FetchEvent` listener, which operates as a network proxy. A third-party (`website1`) includes an `iframe` that activates and uses `example.com`'s SW for fetching the resource.

IV. MISUSING SERVICE WORKERS: HISTORY SNIFFING

In this section we present the methodology and design of our proposed history sniffing attacks and detail the SW capabilities that we use as building blocks. In a nutshell, our attacks rely on the ability to infer the presence of a SW in the user's browser, by observing and measuring the side effects of their functionality when "probing" them. The presence of a SW in the user's browser shows that the user has visited a particular website in the past. Our ability to do so stems from the lack (or ineffective) isolation of SWs in modern browsers.

When a `FetchEvent` listener has been implemented in a SW, all HTTP requests (i.e., fetch events) that originate from a page within the SW's scope will go through it. The SW can either forward the request to the destination over the network, or return the requested resource from its cache storage. Figure 3 presents an overview of this functionality where `example.com` has a SW that uses the fetch API. When a request for `example.com/img.jpg` is issued while the user is navigating `example.com`, that request will go through the SW and be handled accordingly. When this image is requested as a cross-origin resource from `website2.com`, the request does not go through the SW of `example.com`, as `website2.com` is not within the SW's scope; instead, the request is directly sent to `example.com`. However, if the image's URL is used as the source of an iframe on a third-party website (i.e., website1.com), the image request will go through the SW. In other words, the iframe activates example.com's SW, which handles the request similarly to requests that originate from the first-party website. As such, the SW will return the resource from its cache, if it's cached, or fetch it from the first-party's server otherwise. In general, a SW acts as a proxy for all requests that originate from pages within its scope. In practice, however, it can also be activated by an iframe on a third-party website when the src attribute of the iframe is a URL within the SW's scope. This lack of proper isolation, creates a new attack vector for history sniffing attacks. Next, we detail our two main techniques.

A. *PerformanceAPI-based Attack*

In the first attack, the attacker's website attempts to load an iframe for one suitable resource (automatically identified by our tool described in Section IV-D) for each target website, in an attempt to activate their SWs and make them handle the resource fetching. When the resources are loaded, we utilize the PerformanceResourceTiming interface [15] to infer whether the resource was fetched through a SW. While we use a timing API, this attack is *not* a timing-based attack. The PerformanceResourceTiming interface allows web applications to retrieve detailed timing data regarding the loading of their resources. This API provides timing information for various steps of each resource's loading process, e.g., redirection, DNS lookup, TCP connection setup, etc. However, in the case of cross-origin resources, the API will by default return a value of zero for most attributes. According to the Resource Timing W3C Draft Specification [16]: "Cross-origin resources MUST be included as PerformanceResourceTiming objects in the Performance Timeline. If the timing allow check algorithm fails for a resource, these attributes of its PerformanceResourceTiming object MUST be set to zero: redirectStart, redirectEnd, domainLookupStart, [...]".

Crucially, however, there are still other attributes that can be used for inferring whether a resource was fetched through a SW. We have empirically found that the workerStart and nextHopProtocol attributes can be used for our attack. In more detail, the workerStart attribute returns a timestamp immediately before dispatching the SW's FetchEvent. If the request is not intercepted by a SW, the attribute will always return zero. In other words, if the value of the workerStart attribute is non-zero, it means that the request has been intercepted by a SW. While cache-based attacks are typically destructive (i.e., if a given resource does not already exist it will be fetched), this technique is not. We empirically found that the cache storage has a higher priority than the browser cache. Therefore, even if a resource exists in the browser cache, it will be retrieved by the SW from the cache storage, and the value of workerStart will be non-zero. When there is no SW, the value of workerStart will be zero, regardless of whether the resource is cached in the browser cache or not. Furthermore, the nextHopProtocol attribute returns a string value representing the network protocol used to fetch the resource. This attribute will contain an empty string when the resource is "retrieved from relevant application caches or local resources" [16]. Our experiments reveal that it always returns an empty string when a SW is used. Therefore, we can use the nextHopProtocol attribute, similarly to workerStart, to infer the presence of the SW in a user's browser.

This attack can test multiple websites in parallel without affecting its accuracy. This is done by loading multiple iframes for different target domains at the same time. Our attack prototype injects multiple *hidden* iframes in batches; after each batch finishes loading we inspect the resources' entries in the Performance table and call clearResourceTimings to clear the timing buffer.

B. *Timing-based Attack*

In this attack we determine if a user has visited a website by measuring the time that it takes to load a requested resource. If the user has visited the website previously and the requested resource is already cached by the SW, the resource will be retrieved from the cache storage instead of being fetched over the network. Since the loading time is significantly lower when the resource is cached locally, we can determine if a website's SW is installed in the user's browser or not.

For this attack we need to compare the measured loading time with a baseline value. A simple approach would be to compare the target resource's loading time to a fixed threshold. Our experiments showed that the performance of this approach can be affected by the user's network and browser load (i.e., multiple open tabs, multiple resources fetched simultaneously, etc). Another approach that is more robust to such external factors is to concurrently load the same resource twice, once through the SW's cache (if a SW is installed) and the other over the network, and to compare their loading times. To achieve this, our attack uses three iframes for each target resource. First, we use an iframe for booting up the SW; the sole purpose of this iframe is to remove the bootstrap delay from the timing measurements. Then, our JavaScript code simultaneously injects two more iframes in the page that separately load the same resource. While both iframes point to the resource's URL, one employs cache-busting and decorates the URL with a random parameter (e.g., src="example.com/img.jpg?v=random"). Since the decorated URL does not match a cache entry, the resource will be fetched directly from the server.

```
var iframe = document.createElement("iframe");
var body = document.getElementById("body");
body.appendChild(iframe);
iframe.onload = function(event) {
  duration = performance.now()-start;
  iframe.remove();
}
start = performance.now();
iframe.src = <URL>
```

Listing 5. Adding an iframe and measuring its loading time.

As shown in Listing 5, to estimate the resources' loading times our code calls performance.now() right before injecting the two iframes in the page and after each iframe is loaded. Since both iframes are injected in the page at roughly the same time, we expect both to be similarly affected by the browser's load, and that any significant difference in the loading times will be the result of one of them being retrieved from the cache and the other being fetched from the server. If the SW is not installed, both requests will end up on the network, and the resources' loading times will be comparable. We empirically found that if the loading time of the target resource is at most equal to 0.8 of the loading time of the control resource, we can deduce that the target resource is retrieved from the cache. Otherwise both resources were fetched from the network. After running this attack against a particular user, all the target resources will be stored in the browser cache. Thus, when running the attack again for the same user, we will not be able to detect if the target resources are coming from the cache storage or the browser cache. As such, we cannot have accurate results if we repeat the attack on the same user.

C. Browser Behavior

Some aspects of browsers' functionalities and operations are not standardized, and in certain cases browsers may behave differently. In this section, we compare how these different behaviors can affect the coverage of our attacks.

Security headers. Our attacks rely on `iframes` for loading cross-domain resources in the attacker's website. However, websites can restrict their resources from being *rendered* in `iframes` through the `X-Frame-Options` (i.e., '`deny`' or '`sameorigin`') or `Content-Security-Policy` (`frame-ancestors`) headers in their responses [42]. When such restricted resources are used in `iframes`, they are fetched, but the browser does not display them. With regards to the Performance API, Chromium-based browsers do not provide any `PerformanceEntries` (i.e., entries returned by the `PerformanceResourceTiming` interface) for such restricted cross-domain resources. Thus, such resources cannot be used for the PerformanceAPI-based attack if the victim browser is Chromium-based; however, we find that this affects less than 20% of susceptible domains. On the other hand, while Firefox respects the security headers and does not render such resources in `iframes`, it provides timing information for them through the Performance API. As such, in the case of Firefox, these response headers *do not prevent our attack.*

Interestingly, we observed a peculiar behavior for certain domains like `www.nytimes.com`. Specifically, we found that their resources have the security headers when a SW is not installed and the resource is fetched directly from the server, but they are absent when a SW is installed in the user's browser and the resource is loaded from its cache storage. After a more in-depth inspection, we deduced that the back-end server of these domains are configured to add such headers when the request originates from a third party but not for first-party requests. As a result, in the first visit to the website, where the resources are inserted into the cache storage, they are stored without the security headers. In such cases, since the headers are absent when the resource is loaded from a SW's cache, Chromium-based browsers handle them similarly to any other resource that is not restricted by headers and, accordingly, provides the timing information in the Performance API. Therefore, by observing those resources' entries in the `PerformanceResourceTiming` results we can infer that the service worker is installed.

It should be noted that while the security headers prevent cross-origin resources from being rendered in `iframes`, these resources are fetched by the browser, and thus an attacker can still estimate the time required for fetching them. Subsequently, the *timing-based attack is still possible* in all browsers even when the mentioned security headers prevent iframes from being loaded in the attacker's website.

Non-destructive attack. In Chromium-based browsers the `nextHopProtocol` attribute is empty not only when the response comes from the SW but also when the response comes from the browser cache. That is, when the `nextHopProtocol` attribute is used for inferring the user's visited websites, after running the attack once, the browser may add some of the fetched resources in the browser cache. This can occur for domains that did not have a SW installed in the user's browser. In such a case, running the attack at a later time can potentially result in a false positive detection where a domain is incorrectly identified as part of the user's browsing history. To avoid this issue we add a random parameter to each resource's URL when injecting the `iframes` in our website. In this way, the request bypasses both cache storage and browser cache. This request goes through the SW, which then fetches the resource from the network, and then sends it back to the attacker's website. The attacker is able to detect that the response has come from a SW by checking the value of the `nextHopProtocol` attribute. Interestingly, we observed that Firefox returns an empty string when the resource is retrieved from the cache storage by a SW, but not when loaded from the browser cache. This discrepancy in Firefox makes our attack non-destructive even without adding the random parameter.

D. Automated Resource Profiling

An important and challenging dimension of our work is to automatically identify resources that are susceptible to our attacks, which would enable running our attacks at scale. In this section, we describe our automated tool that relies on a differential analysis approach for identifying such resources.

This tool is built on top of the instrumented Chromium browser that we describe in Section III. Specifically, we use the instrumented browser for logging all the requests that are intercepted by the SW's `FetchEvent` listener. When visiting each website, our tool collects the URLs that have gone through the SW as well as the URLs of the resources that are stored in the website's cache storage. To identify cached resources we use Chrome's DevTools Protocol. By calling 'requestCacheNames' from the 'CacheStorage' domain we get the names of the caches, and 'requestEntries' returns the data that is stored in them. In Listing 6 we include Python code that uses Selenium Webdriver to collect the cached resources. Listing 7 shows our code for collecting the resources' URLs and their headers from the results of the DevTools Protocol.

```
caches = driver.execute_cdp_cmd(
  "CacheStorage.requestCacheNames",
  { "securityOrigin": <website_origin>})['caches']
allCacheStorages = []
for cache in caches:
  id = cache['cacheId']
  entries = driver.execute_cdp_cmd(
      "CacheStorage.requestEntries",
      {"cacheId": id,
      "skipCount": 0,
      "pageSize": 50,
      "pathFilter": ""})['cacheDataEntries']
  allCacheStorages.append(entries)
```

Listing 6. Using Chrome's DevTools protocol for collecting cached resources.

```
resources = []
for cacheStorage in allCacheStorages:
  for record in cacheStorage:
    reqUrl = record["requestURL"]
    headers = record["responseHeaders"]
    resources.append([reqUrl,headers])
```

Listing 7. Collecting the URL and headers of all the cached resources.

After identifying the URLs of all the fetched and cached resources for each website we filter out the URLs that correspond to third-party domains, as these cannot be used in our attacks. Finally, we test the suitability of the remaining URLs as described next.

Attack variant 1: PerformanceAPI-based. For each one of the resources our tool launches the instrumented browser (which already has the SW installed from the previous step) and a fresh instance of an unmodified browser. In both browsers, we open a website under our control that includes an `iframe` that loads the target resource. At this point, our modified browser checks whether (i) the requested resource goes through the `FetchEvent` and (ii) the response is obtained with the `respondWith()` function (described in Section II). In both browsers, our tool also inspects the resource's HTTP response for security headers such as `X-Frame-Options` and `Content-Security-Policy`. It also inspects the values of the `workerStart` and `nextHopProtocol` attributes in the Resource Timing API [15]. Comparing the results of the two browsers allows us to verify if the resource is suitable for the PerformanceAPI-based attack or not.

Attack variant 2: Timing-based. In the PerformanceAPI-based attack we can use all the URLs collected from a FetchEvent or the cache storage. For the timing-based attack, however, we can only use URLs from the cache storage. To identify resources suitable for the timing-based attack our tool performs the following process. Again we use two instances of our browser, with and without a SW, and open a website under our control. Our website has two `iframes` this time: one fetches the target URL, and the other fetches the same URL decorated with a random parameter. Comparing the loading times of all four resource requests allows us to understand whether (i) the SW processes both requests in the same way (i.e., the SW strips the random value), (ii) there is a CDN on route to the server (the URL with a random parameter in the fresh browser takes significantly longer time to be fetched), and (iii) the URL is detectable based on the differences in the loading times. To determine if a resource is a good candidate for our timing-based attack, we run this process 3 times and check whether the loading times follow a consistent pattern.

E. Vulnerable Browsers

As shown in Table III, all Chromium-based browsers and Firefox are vulnerable to both of our attacks. More specifically, for the first attack, which leverages the Performance API, we can use the `workerStart` and `nextHopProtocol` attributes. In Firefox, both attributes can be used to reveal if a requested resource is fetched through a SW. For Chromium-based browsers, the PerformanceAPI-based attack that uses the `nextHopProtocol` attribute works in all browsers, while the version that leverages `workerStart` works in all but Brave. Furthermore, all Chromium-based browsers and Firefox are vulnerable to the timing-based attack. Our attacks are not applicable against Safari as it correctly isolates SWs (i.e., a SW cannot be activated by an `iframe` on a third-party website). Interestingly, since iOS restricts browsers to use the Webkit [1] browser engine, they all behave the same as Safari. While browsers on iOS are not vulnerable to our attacks, they are on MacOS. Finally, in the Tor browser and Firefox's private browsing mode, websites cannot register a SW; therefore, they are not vulnerable to these attacks. Incognito mode in Chrome works like normal mode, and users are vulnerable to these attacks. However, since the two modes are isolated from each other, the attacker does not have access to SWs that had been installed in the browser's normal mode, and incognito SWs are removed after closing the window; thus the attacks can only detect websites that are currently open in different tabs. As such, the attacks have limited applicability in incognito mode.

TABLE III. BROWSERS THAT ARE VULNERABLE (●) TO OUR HISTORY SNIFFING ATTACKS. WS AND NHP STAND FOR WORKERSTART AND NEXTHOPPROTOCOL RESPECTIVELY.

Browser	Version	PerformanceAPI		Timing
		WS	NHP	
Firefox	72.0.2	●	●	●
Brave	1.3	○	●	●
Chrome	79	●	●	●
Edge	79	●	●	●
Opera	66	●	●	●
Safari	12.1.2	○	○	○

TABLE IV. DOMAINS SUSCEPTIBLE TO EACH ATTACK.

Attack	Firefox	Chromium-based
API-based	6,706 (100%)	5,507 (81.03%)
Timing-based	6,504 (96.98%)	6,504 (96.98%)
Combined	*6,706 (100%)*	*6,591 (98.28%)*

V. Experimental Evaluation

Here we present a series of experiments that explore practical aspects of our attacks and their privacy implications.

History-sniffing susceptibility. Our large-scale measurement study detected 8,895 SWs with a `FetchEvent` listener, which is the main requirement for our attacks. For our attacks we only consider websites with `Fetch` functionality on their landing page. Using our automated resource profiling tool we identified a total of 6,706 websites that have resources suitable for running our attacks. In Firefox, both variations of the PerformanceAPI-based attack work on all 6,706 websites. In Chromium-based browsers our PerformanceAPI-based attack can detect a total of 5,507 (81.03%) websites. Specifically, we identified 5,465 websites that have certain resources that do not include `X-Frame-Options` or `CSP` header, and 302 websites that have at least one resource that only contains these headers when it is requested directly by a third-party and not through a SW. Some of these cases overlap, resulting in 5,507 unique domains. Finally, we identified 6,504 (96.98%) websites that are susceptible to our timing-based attack. This attack has the same coverage in all vulnerable browsers. The other 202 (3.02%) websites have `FetchEvent` and requests are intercepted by the SW, but they do not actually cache any

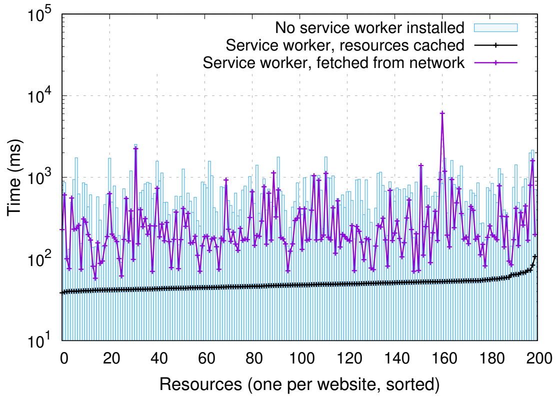

Fig. 4. Average loading times when the requested resources are retrieved from SWs' cache or fetched from the server. The latter occurs if (i) the resource is not matched with the cache's contents, or (ii) the SW is not installed.

resources. Certain websites are vulnerable to the API-based attack but not the timing-based attack; when combining both of our attacks we can detect 6,591 unique websites in Chromium-based browsers, accounting for 98.28% of all the susceptible websites. These results are summarized in Table IV.

Timing-based attack. Since the PerformanceAPI-based attack is always accurate by design (i.e., it does not have any false positives or negatives), here we evaluate the performance and practicality of our timing-based attack. To that end, we first run our automated resource profiling tool (i.e., timing-based mode, see Section IV-D) and identify a suitable resource on 200 randomly-selected websites that can be used in this attack.

Feasibility. First, we explore how loading time changes under the different scenarios the attacker can face, and present the average loading times for each resource (from the 3 runs performed by the tool) in Figure 4. Specifically, in this figure we aim to illustrate the discriminating effect of the presence of a SW combined with the caching of the resource. Our experiments show that when a resource is retrieved from the cache storage the loading times are significantly lower than the time that is spent for fetching the resource from the network (regardless of a SW being installed or not), demonstrating the effectiveness of using these resources in a timing attack.

Performance. Next, we assess the attack's accuracy in practice. To that end, we randomly visit N out of the 200 websites and emulate a user's browsing activity that installs SWs. Subsequently our user visits the attacker's website, which conducts the timing-based attack for inferring which pages the user has visited. We run this experiment 50 times each for $N = 20, 50$. As shown in Table V, our attack correctly detects 87.9% and 87.13% of the websites that have a SW installed, for browsing history of a size of 20 and 50, respectively. Also, in both cases our attack has a low false positive rate (i.e., websites that our attack incorrectly detects as visited) of around 1.5%, and is very precise, with an overall F1 score above 92%. After investigating our false negatives, we observed that most of them correspond to a small set of websites that our attack cannot detect across most (or all) of the runs where these websites had a SW installed. This is due to the loading times of the two requests being sufficiently similar for considering both as being fetched from the network.

TABLE V. DETECTION ACCURACY OF THE TIMING-BASED ATTACK.

	#SWs Installed	
Metric	$N = 20$	$N = 50$
True Positives (TP)	87.9%	87.13%
False Negatives (FN)	12.1%	12.86%
False Positives (FP)	1.48%	1.63%
F1 Score	92.8%	92.3%

In some cases our attack cannot determine correctly if a SW is installed or not. These cases can be attributed to (i) the way that particular SWs handle fetch events and (ii) the use of content delivery networks (CDNs) for caching first-party resources. In particular, with regards to the first case, we observed that some SWs fetch the resource from the network for both `iframes`, even though it is already stored in the cache. After examining their source code, we found that some SWs implement a network-first caching strategy for particular resources [9], where they first attempt to fetch the resource from the network, and if this is not possible (i.e., the user is offline) they serve a cached, and probably older, copy of the resource. Also, we came across cases of incorrect implementations, where the SW does not attempt to match the request with the cache, and always fetches the resource from the network. In those cases, our attack cannot detect the existence of a SW, as both resources are fetched over the network. We have also observed a small number of SWs that strip any parameters from requested URLs before attempting to match them with the contents in their cache. This also results in inconclusive loading times that prevent our attack.

The second problematic category is when websites utilize CDNs to serve their resources. The issue depends on the CDN's behavior, and can occur when there is no SW installed in the user's browser. Specifically, if a SW is installed, the resource that does not include a random parameter in its source URL will be retrieved from the SW's cache, while the other resource will most likely not exist in the CDN's cache due to the random parameter. In this case our attack correctly identifies that a SW exists. In the case, however, where there is no SW installed, both requests will be sent to the CDN, and most probably one of the responses will be served from the CDN's cache while the other will be retrieved from the first-party's server by the CDN (i.e., the CDN cache may miss because of the random parameter). Due to the significant difference in the resources' loading times our attack will consider the effects of the CDN caching as being caused by a SW, i.e., a false positive.

Attack duration. An important dimension of our attacks, which determines their practicality, is the time required for the attack to complete. Thus, we run an experiment that measures the duration of each attack for various numbers of tested domains. In each iteration of the experiment we visit our website 6 times. In 3 of the visits we have 10% of the websites with a SW installed. In the other 3 visits we do not install any SW, simulating the worst case scenario where all resources need to be fetched from the network. Furthermore, our website is configured to perform the timing-based attack during the first visit, and the Performance-API-based attack during the other two visits. The difference between the latter two visits,

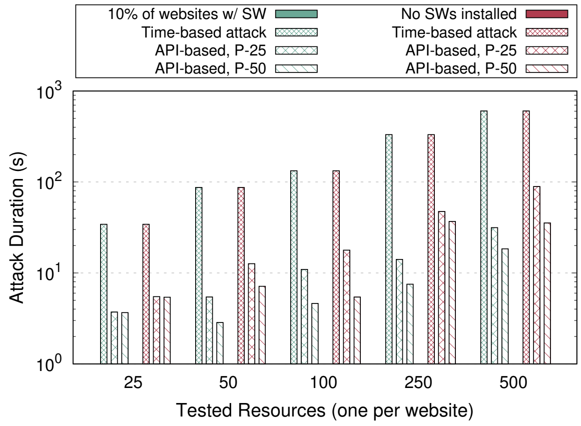

Fig. 5. Time required for performing the proposed attacks.

is that in one of them the attack loads 25 `iframes` in parallel while in the other one it loads 50. This allows us to compare our attacks both in the presence and absence of SWs. Finally, we run the experiment for different number of websites to be tested, to assess the scalability of our attacks. In each run we choose a random subset of resources to be tested, and test the same set of resources in all six visits.

Figure 5 presents the average attack duration over 10 different runs, both with SWs installed and without, for a varying number of resources (we test up to 500 websites). The timing-based attack takes much longer to complete; this is expected, as the attack tests the resources one-by-one, to avoid interference that can affect the loading times. Also, since the timing-based attack always fetches at least one resource from the network (the one with the random parameter), the duration of the attack does not decrease significantly when SWs are installed. On the other hand, the API-based attack issues multiple requests in parallel and is much faster than the timing-based one. Indicatively, when parallelizing 50 requests this attack tests 500 domains in less than 18 and 35 seconds, depending on SWs being installed or not. While the timing-based attack is not optimal for testing a large number of domains, in practice, the attacker only needs to use it for the cases that cannot be detected by the Performance-API-based attack (see Table IV). Apart from combining the attacks, attackers could also follow a more targeted approach and compile a list of websites that reveal sensitive information, or websites that belong to specific categories of interest.

Classifying detectable websites. To better understand the privacy implications of the proposed attacks, we used McAfee's website categorization tool to categorize the 6,706 websites that are vulnerable to our attacks, and check whether any sensitive ones are included. This allowed us to categorize 6,412 websites, which were assigned to 78 different categories. We manually inspected these categories and combined certain sensitive categories that are closely related, resulting in 72 categories. For instance, we consider the categories of *Health*, *Pharmacies* and *Drugs* as a single category in our analysis.

Figure 6 presents a subset of the categories that reveal information about the user's interests and preferences, as well as personal and sensitive information. As expected, many websites are related to *Online Shopping* and *Merchandising* (1,690 and 507 respectively). While the non-sensitive categories may not directly reveal private information about the user, they are typically part of Ad Preference Manager profiles [17] and can be leveraged by advertisers for user targeting [30]. Moreover, websites that are susceptible to fine-grained history sniffing (see Section VI-C) could enable the inference of sensitive data. Regarding the sensitive categories, we observe a considerable number of websites that are related to *Health* and *Pornography* (i.e., 112 and 90, respectively), and a smaller number of websites that are related to *Religion*, *Dating* and *Politics*. In total, we found 403 (6%) of the detectable websites that are associated with sensitive categories, that reveal user information, and can be misused by attackers. Interestingly, 124 of these websites are also susceptible to our fine-grained history sniffing attack, potentially revealing highly sensitive information about the user. Finally, we find that our API-based attack in Chrome can detect 294 of the 403 sensitive websites, as they do not use `x-frame-options` or `CSP` headers.

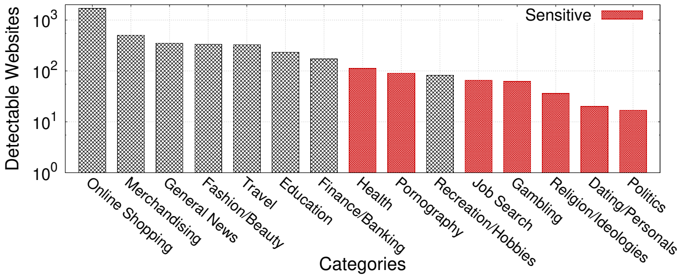

Fig. 6. Categorization of websites detectable by our attacks.

VI. Additional Attacks and Use Cases

Due to the idiosyncrasies of SWs' caching behavior, our attack methodology is not limited to stealing users' browsing history, but can also be used to infer additional, potentially more sensitive information, by targeting specific pages and resources. Here we present a series of use cases that highlight the additional capabilities of our attack techniques.

A. Registration Inference

During our experiments we observed that certain websites fetch and store additional resources when users are logged into their account. Two interesting examples that illustrate the privacy implications of this behavior are Tinder (a popular dating site) and Gab (a site that attracts "alt-right users, conspiracy theorists, and trolls, and high volumes of hate speech" [58], [57]). When the user visits these websites for the first time and SWs are registered, they do not populate the cache with all the needed resources; some of them are fetched and cached only after the user is authenticated to the service.

While these post-login resources are not sensitive, detecting them in the user's cache reveals not only that the user has visited the website at some point, but that they also have an account on that service. We also observed that these resources are not deleted from the cache storage after the user logs out; as such, this works even if the user is not currently logged in. It is important to note that we are not able to identify such post-login resources in a fully automated manner, since that would require the ability to automatically register and create

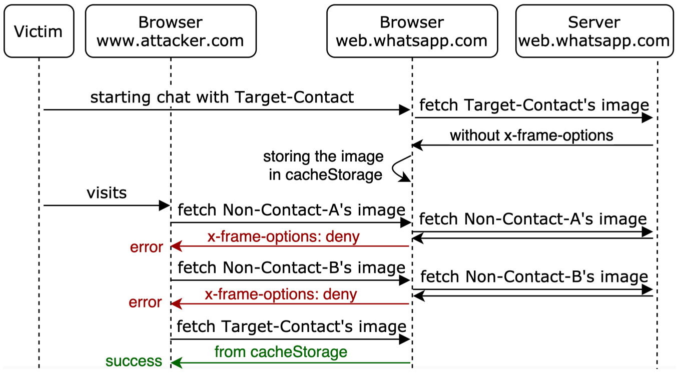
Victim
Browser www.attacker.com
Browser web.whatsapp.com
Server web.whatsapp.com
starting chat with Target-Contact
fetch Target-Contact's image
without x-frame-options
storing the image in cacheStorage
visits
fetch Non-Contact-A's image
fetch Non-Contact-A's image
x-frame-options: deny
error
fetch Non-Contact-B's image
fetch Non-Contact-B's image
x-frame-options: deny
error
fetch Target-Contact's image
from cacheStorage
success

Fig. 7. Social graph inference attack on the web application of WhatsApp.

accounts. Nonetheless, our automated resource profiling tool can be used to detect such resources once an account is created, by comparing resources that are cached pre- and post-login.

B. Application-level Inference

Our attacks can also be used to reveal application-level personal or sensitive information. We use the web version of WhatsApp (`web.whatsapp.com`) to illustrate such an attack. When the victim logs into WhatsApp the SW populates the cache storage with profile pictures of the contacts that the victim has communicated with, as well as images of the groups that they are a member of. Furthermore, user actions that result in a thumbnail image being shown to the user will cause the image to be cached by the SW. For example, when the user searches for a contact, or when they click on the "New chat" button and the contact list is displayed, all the images that appear as thumbnails in the page end up in the cache storage. These images can be used for our attacks, similar to any other resource that is stored in a SW's cache. In this case, the resources not only reveal that the victim has visited the WhatsApp's website, but that particular individuals are among the victim's contacts. To that end, by searching for multiple resources, the attacker can (partially) reconstruct the victim's social graph. We present an overview of our attack in Figure 7.

Initially, the victim performs an action such as sending a message to one of their contacts, prompting the application to fetch and cache the contact's image. It should be noted that the server's response does not include the `X-Frame-Options` header when returning this image. When the victim visits the attacker's website, the website uses multiple `iframes` to request images of various users that could potentially be contacts of the victim. If a specific user is not among the victim's contacts (i.e., the image is not in the cache) the SW fetches that image from the server. In this case the server's response includes the restricting `X-Frame-Options` header to prevent the browser from displaying the image in an `iframe`. This difference in the response headers of cached and non-cached images allows the attacker to easily distinguish targeted contacts that are indeed among the victim's contacts. Similarly, requesting group images instead of individual users' pictures allows the attacker to infer whether the victim is a member of any of the tested groups. We also experimented with our timing-based attack to test the resources' loading time and found that we can again distinguish cached resources.

Identifying WhatsApp resources. Constructing the attack website requires knowledge of the URLs of the targeted contacts' profile images. The attacker can obtain that image URL through the following web server endpoint: `web.whatsapp.com/pp?t=s&u=<number>&i=<timestamp>` While this endpoint requires a phone number and timestamp combination, the attacker can trivially obtain those by adding the phone number as a contact in their phone. As such, the attacker can conduct a targeted attack probing the victim for specific users (e.g., a law enforcement agency searching for connections to known criminals) or simply brute-force a large number of URLs collected in a preparatory phase (e.g., create city-based hit lists and serve based on victim's IP address).

Partial user deanonymization. We also present an additional, more challenging attack that can be deployed against WhatsApp users for inferring that they belong to a given WhatsApp group. While targeting individual contacts is straightforward, inferring group membership requires the phone number and timestamp of its creator, and an *additional* timestamp that corresponds to the groups creation time. To obtain all the necessary information the attacker needs to be a member of the group (or collude with a member). However, if that information is available, the attacker can map the user visiting their website to one of the members of a specific group, thus partially deanonymizing them. While the attacker could also, *theoretically*, fully deanonymize them by reducing the set of potential users by probing combinations of different groups with different intersections of users (similar to the techniques in [53]), we do not consider this a likely threat.

C. Fine-grained History Sniffing

While some websites use SWs' cache storage only for storing necessary resources, other websites' SWs dynamically store additional resources when the user navigates to different pages on that domain. While both strategies reveal that the user has visited the specific website, the latter one also provides fine-grained information about the navigation of the user within the visited website. To detect websites that implement this type of caching strategy, we evaluate all websites with SWs that use the cache storage API (see Section III). We first crawl each one and collect 20 URLs, and then use Selenium to visit each page in Chrome and compare the contents of the cache storage before and after visiting each URL. If at least half of the visited pages add new resources to the cache storage, we flag that website as a candidate for additional inspection.

Our system flagged 1,964 websites that follow the aforementioned caching strategy. We randomly selected 200 of these websites and manually inspected them, and found that 157 (78.5%) are indeed vulnerable to this attack. In the remaining 43 websites new resources are added but they are not unique to each page that is visited. For the vulnerable websites, an attacker can determine exactly which pages the user has visited, which can reveal the user's preferences and interests or other sensitive information. This information can be used to infer private user traits and attributes [28]. An example website that is vulnerable to such an attack is `spokeo.com`. This website aggregates information about people from various sources and allows users to search for an individual's information. We found that this website stores all user's search queries into the cache storage, thus allowing an attacker to

```
self.addEventListener('fetch',function(event){
referrer = (new URL(event.request.referrer)).host;
if(referrer==self.location.hostname ||
   referrer.match(<allowlist-item>)!=null){
       /*Remaining SW functionality goes here*/ }});
```

Listing 8. Controlling access to SWs by leveraging the referrer.

infer whether the victim has searched for specific individuals. Another interesting example of a website that is susceptible to this attack is pleasurestore.in. This website, which sells sex paraphernalia, fetches and stores the images of the products that appear in every page the user visits. As such, attackers not only infer that the user has visited this store, but can also learn more sensitive information (e.g., infer the user's gender or sexual preferences and orientation).

VII. Attack Mitigation

The root cause that enables our attacks is the improper isolation of SWs in browsers, which allows iframes on third-party websites to use the SWs of other origins for fetching resources. This can be prevented by redesigning site isolation mechanisms to prevent the activation of SWs from third-party websites. We have disclosed our findings to vulnerable browsers, which are currently working towards fixing the underlying problem. However, such non-trivial changes will require a considerable amount of time before being deployed.

As such, we propose a mitigation and build a tool that can assist web developers with fortifying their SWs against our attacks. Our solution is based on implementing access control logic inside SWs to restrict them from responding to incoming requests that originate from unauthorized domains. In more detail, requests that originate from an authorized domain (e.g., the first-party domain) will be processed normally while those that originate from a non authorized one will bypass the SW and the resources will be fetched directly from the network. This can be implemented by using the referrer header of the request. As shown in Listing 8, by using the referrer header we can allow access to the relevant functionalities of the SW only to pages of the first-party website (or other allowed domains). If the referrer is not from an authorized origin (or is an empty string) the request will go through the network, as would happen if the SW was not installed.

To bypass this countermeasure, attackers might try to spoof the referrer in the requests issued by the attack website. To the best of our knowledge, this cannot be done directly with JavaScript, as the referrer is set and controlled directly by the browser, and is a read-only attribute. In this context, an attacker can only effectively spoof the referrer header if the target website supports open redirections, by specifically crafting a request that redirects to the requested resource. For example, considering a target website example.com, the attacker would need to set the iframe's source to example.com/?redirect=example.com/img.jpg. This sets example.com as the referrer for the resource request, instead of the attacker's domain. In this case, the attack can be prevented if the target website deploys our proposed countermeasure and also uses the appropriate X-frame-options or CSP headers to restrict frames. It should be noted though that the use of the X-frame-options or CSP headers cannot prevent the time-based attack when our countermeasure is not in place, as the frames are actually fetched but the browser does not render them, and thus the attacker is still able to measure their loading times.

```
let orig_f = EventTarget.prototype.addEventListener;
EventTarget.prototype.addEventListener = function(){
 if (arguments[0] == 'fetch'){
  let handler = arguments[1]
  arguments[1] = function(){
   let event = arguments[0];
   if (event.request.referrer){
    let referrer =
             (new URL(event.request.referrer)).host;
    if(referrer==self.location.hostname ||
             referrer.match(<allowlist-item>)!=null)
     return handler.apply(this,arguments)
   }
   //else it will be fetched from the network
  }
 }
 return orig_f.apply(this,arguments);
}
```

Listing 9. Adding the access control to SWs by overriding the addEventListener's prototype.

To assist developers in implementing this countermeasure, we will release our tool that automatically incorporates these checks in the SW's code. Given the SW's source code and a file with a list of authorized domains, our tool injects a function at the beginning of the SW's source file, to be executed first, that overrides the addEventListener function which exists in the prototype of the EventTarget interface. Listing 9 shows how we override this function. If the first argument of the addEventListener function is fetch, we include our access control mechanism in the function that handles the event. This makes the addEventListener('fetch', handler) function in the SWs behave like the function that is shown in Listing 8. That is, if the referrer of the intercepted request is in the allowlist, we run the event handler, otherwise the request will be fetched directly from the network. We note that this approach works correctly even in the case of obfuscated SWs or SWs that import a third-party library for implementing the fetch functionality.

VIII. Discussion

Limitations. Our crawler initially only visits websites' landing page for inferring the presence of a SW, and only further analyses websites if one is found on the landing page. We made this decision to render the overhead of our measurement study more manageable, and because we observed that the majority of websites install their SW on their landing page. Furthermore, our system does not log into websites that support user accounts. As such, our measurements present a *lower bound* of vulnerable domains, since websites that require login may install SWs on other parts of their domain or SWs may only cache resources after users login.

The process of finding websites that are vulnerable to the *Registration Inference* and *Application-level Inference* attacks cannot be completely automated as this would require an account on each website. It also requires extensive manual effort for understanding the nature and *purpose* of different cached resources (e.g., knowing that the cached images in WhatsApp correspond to user photos) as opposed to the history sniffing attacks which do not require such knowledge. While we present a series of interesting use cases, our goal is to demonstrate the feasibility and severity of such attacks, not to provide a complete manual evaluation of all such services.

The API-based attack is efficient as it can issue requests for multiple resources in parallel without affecting its accuracy, where batches of 500 websites can be tested every 10-20 seconds, depending on the level of parallelization. On the other hand, the timing-based attack is not optimal for testing a large number of domains as it cannot be parallelized; thus it is better suited for more targeted attacks (e.g., only sensitive websites).

Ethics and disclosure. Our experiments were conducted using our own browsers and test accounts. We did not interact with, or affect, actual users. Due to the severe privacy implications of our attacks we disclosed our findings and techniques to all vulnerable browser vendors and WhatsApp (in January and February, 2020). Chrome split our report into two bugs: one for the PerformanceAPI, which has been assigned a CVE (`Blink>PerformanceAPIs`) and one for the site isolation (`Internals>Sandbox>SiteIsolation`). Chromium releases that follow our disclosure have fixed the issues that are related to the Performance API. Specifically, for cross-origin iframes the PerformanceAPI now returns the value of zero for the `workerStart` attribute and an empty string for `nextHopProtocol`; this prevents our API-based attack in Chromium-based browsers. Firefox also fixed the PerformanceAPI issues, following our disclosure report, by restricting the `workerStart` and `nextHopProtocol` attributes. However, their fix actually introduced a new issue that re-enables our attack: while in previous versions the `duration` attribute always returned the request's duration, in the newer version this attribute returns zero when the request is intercepted by a SW and the actual duration otherwise. We have reported this new issue to Firefox.

Our attacks are possible due to a design flaw in the browsers' site isolation mechanism that allows third-party websites to use other parties' SWs. Unlike the PerformanceAPI-based attack that can be prevented by restricting specific attributes of the API, the timing-based attack requires the re-design of the site isolation mechanism. This task is not trivial, and these issues will most likely take a considerable amount of time to be fixed. Specifically, Chrome's feedback about these issues stated that "this requires web API changes" and that "a fix is likely quite a ways off". Since the underlying issues have not been fixed yet, and our timing-based attack is still possible, our countermeasure will allow websites to protect their users until browsers redesign their systems. Finally, WhatsApp fixed the issues that allowed our application-level inference attack by restricting cross-origin requests from accessing the SW cache, similarly to our proposed countermeasure.

IX. Related Work

Here we discuss prior work on history sniffing, and pertinent studies on the security implications of browser features.

History sniffing. Various attacks have been demonstrated for sniffing users' browsing history. Several of those were through CSS features, with the `visited` pseudoclass being one of the first features misused for inferring whether the user has visited a specific URL based on the color of the rendered hyperlink [19]. Janc and Olejnik [24] demonstrated a practical implementation of this attack and conducted a study on over 270K users. To prevent such attacks, browsers have stopped providing DOM mechanisms for directly detecting element styles. Recently, Smith et al. [45] leveraged the CSS Paint API, and also showed how the bytecode script cache can be misused in Chrome and Brave. As with all cache-based techniques, the attack's practicality and robustness can be considerably affected by external factors that result in the browser evicting targeted resources (we note that not all attacks in [45] are cache-based). On the contrary, the SW cache that we exploit is solely under the SW's control and no eviction occurs unless the device or browser runs out of disk storage. Lee et al. [32] evaluated the susceptibility of eight websites to an attack that infers the caching of resources in the HTML5 App Cache. This cache has since been deprecated, and developers are urged to use service workers instead [13]. While these cache-based attacks are typically destructive we also demonstrate a non-destructive variant of our attack.

Kotcher et al. [29] proposed a timing-based attack that measured the time required for rendering CSS filters, which could be used for sniffing pixels rendered on the user's screen. One of the presented use cases was for a history sniffing attack; however, accuracy was low and the overall attack impractical as it required a considerable amount of time for checking a single URL and suspicious visual actions that would alert users (i.e., expanding a pixel to the size of the entire screen). Timing-based attacks were proposed as early as 2000, with Felten and Schneider demonstrating how this could be achieved by measuring the time for performing a cross-site request [22]. The approach of Bortz and Boneh could infer if a user was currently logged in a website but not whether it had been accessed in the past [18]. Sanchez-Rola et al. [43] measured the time required for server side computation to complete an HTTP request carrying cookies; this attack works if the cookies for a given domain have not expired or been deleted, and if the sameSite cookie flag has not been set for at least one cookie. They also performed the largest evaluation of a history sniffing technique up to that point, with $\sim 10K$ websites. Comparatively, we analyzed the top one million Alexa sites for the presence of SWs, and evaluated the susceptibility of over 30K domains. Dabrowski et al., proposed a different cookie-based attack, where a rogue captive portal could infer websites the user has visited in the past [20].

In other side-channel techniques, Kim et al [26] aimed to infer browsing history information based on changes in the browser's storage. Their attack is prone to false positives since a multitude of online resources (e.g., banners, images, scripts) regularly fetched from different domains can have the same storage footprint. This is reflected in the attack's low accuracy despite their experiments being conducted on a few popular sites. A more realistic number of sites (i.e., moving towards an open-world setup) would significantly increase false positives. All major browsers fixed this issue by partitioning the browser's cache using the top frame's origin [3].

Lee et al. [33] showed that the lack of appropriate memory protection in GPUs could allow the extraction of rendered webpage textures, but evaluated their attack when only two tabs were open in the victim's browser (randomly choosing from the top 100 Alexa websites). In any realistic deployment setting, where users have numerous tabs open, this technique would suffer from many misclassifications. Van Goethem et al. [52] showed how browser features can be leveraged for obtaining timing measurements to estimate the size of cross-origin resources, which can lead to the inference of private information. While they used a SW, their attack could be launched through simple JavaScript that inserts files in the common cache without the use of a SW, i.e., their attack did not require SW-specific functionality. Weinberg et al. [56] demonstrated how interactive tasks (e.g., captchas, games) could be used to trick users into revealing websites in their browsing history. Apart from the practical challenge of requiring user interaction, this attack exfiltrates an extremely limited number of websites. Complimentary to history sniffing, Su et al. [50] demonstrated how a user's browsing history could be linked to social media profiles and deanonymize users.

Browser APIs. As new browser APIs are rolled out, novel attack vectors emerge. Snyder et al. [47], [46] explored the usage of browser APIs and features in the wild, and measured the security vs usability trade-off of removing rarely used features. Olejnik et al. [38] explored how the adoption of seemingly innocuous features like the Battery API can lead to privacy threats (i.e., user tracking). Recently, Das et al. [21] and Marcantoni et al. [37] presented large-scale measurements on the use of mobile-specific HTML5 WebAPI calls that enable a plethora of attacks. Tian et al. [51] demonstrated how the HTML5 screen-sharing API could be used for various attacks; the proposed history sniffing attack requires the target URLs to actually be rendered on the user's screen, presenting an obstacle for the practicality of the attack and limiting the number of target URLs that can be tested. Karami et al. [25] showed how the Performance API can be used to detect what browser extensions a user has installed.

Service Workers are a relatively recent browser feature that has not received much scrutiny. Papadopoulos et al. [39] explored their use for malicious client-side computations like cryptomining, while Franken et al. [23] briefly explored SWs in the context of cookie-carrying third-party requests and found that SW-initiated requests are often not blocked by privacy extensions. Watanabe et al. [55] proposed a persistent man-in-the-middle attack that exploits SWs. In this attack, malicious websites can register a SW in the scope of a rehosting website. By using the `fetch` event listener this malicious SW can intercept and manipulate any requests and responses issued from the rehosting website. Lee et al. [31] focused on the security threats of web push functionality. They also proposed using SWs for a history-sniffing attack which, however, had completely unrealistic assumptions and requirements. Specifically, the attack could only happen if victims visited the attacker's website while they *did not have Internet connectivity*. Furthermore, the victims needed to have already visited the attacker's site in the past so that a malicious SW would already be installed in their browser; the attack was also not applicable to Chromium-based browsers and has since been fixed. Finally, their study was limited to the presence of push and caching functionality, and did not provide a comprehensive view of SW API use.

X. Conclusions

In this paper we investigated an emerging trend in web app development, namely the use of service workers. We conducted a large-scale measurement study and found that the adoption of SWs has steadily increased in recent years, with almost 6% of the top 100K websites leveraging their rich functionality. Subsequently, we conducted an exploration of the threat that SWs pose to users, and presented a series of novel privacy-invasive attacks that exploit their capabilities in most modern browsers. Initially, we demonstrated two variants of history sniffing attacks that bypass current site isolation strategies and allow an attacker to infer the presence of third-party SWs through cross-origin requests hidden in `iframes`. We then presented a more in-depth assessment of the implications of our techniques, through a series of use cases that showcase the feasibility of more privacy-invasive attacks, such as inferring members of a user's social circle or the existence of an account in a "sensitive" web service, or obtaining clues about the users' sexual preferences through cached application-level information. We also presented an experimental evaluation that demonstrates the practicality of our attacks. In an effort to protect users, we disclosed our findings to affected vendors and remediation efforts are currently taking place, including plans for exploring a redesign of Chromium's site isolation mechanism. Finally, we also developed an access-control-based countermeasure to mitigate our impactful attacks while browsers' remediation efforts are underway. Overall, our work sheds light on an emerging and severe threat and we hope that it incentivizes additional research on the risks posed by SWs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the DARPA ASED Program and AFRL (FA8650-18-C-7880), and NSF (CNS-1934597). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government.

References

[1] "Apple Developer - App Store Review Guidelines," https://developer.apple.com/app-store/review/guidelines/#software-requirements, accessed on 2020-02-04.

[2] "Chrome Platform Status - Partition the HTTP Cache," https://www.chromestatus.com/feature/5730772021411840.

[3] "Chrome Platform Status - Split HTTP auth cache by NetworkIsolationKey," https://www.chromestatus.com/feature/5739996117991424.

[4] "Firebase Documentation - Send messages to topics on Web/JavaScript," https://firebase.google.com/docs/cloud-messaging/js/topic-messaging.

[5] "OneSignal Service Worker," https://documentation.onesignal.com/docs/onesignal-service-worker-faq.

[6] "PushProfit," https://www.pushprofit.net.

[7] "SendPulse," https://sendpulse.com.

[8] "Workbox," https://developers.google.com/web/tools/workbox/modules/workbox-sw.

[9] "Workbox - workbox.strategies.NetworkFirst," https://developers.google.com/web/tools/workbox/reference-docs/v4/workbox.strategies.NetworkFirst.

[10] "Service workers and the Cache Storage API," 2018, https://web.dev/service-workers-cache-storage/.

[11] "Forbes - How Progressive Web Apps Will Change Online Business," 2019, https://www.forbes.com/sites/theyec/2019/10/23/how-progressive-web-apps-will-change-online-business.

[12] “Google Developer Docs - Offline Storage for Progressive Web Apps,” 2019, https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/offline-for-pwa.

[13] “MDN Web Docs - Using the application cache,” https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache, March 2019, accessed on 2020-01-05.

[14] “Serviceworkercontainer.controller,” https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerContainer/controller, November 2019, accessed on 2020-01-06.

[15] “Using the Resource Timing API,” https://developer.mozilla.org/en-US/docs/Web/API/Resource_Timing_API/Using_the_Resource_Timing_API, March 2019, accessed on 2020-01-14.

[16] “Resource Timing Level 2 - W3C Editor’s Draft,” https://w3c.github.io/resource-timing, January 23, 2020, accessed on 2020-01-30.

[17] M. A. Bashir, U. Farooq, M. Shahid, M. F. Zaffar, and C. Wilson, “Quantity vs. quality: Evaluating user interest profiles using ad preference managers.” in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*.

[18] A. Bortz and D. Boneh, “Exposing private information by timing web applications,” in *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 621–628.

[19] A. Clover, “Css visited pages disclosure,” 2002, https://lists.w3.org/Archives/327Public/www-style/2002Feb/0039.html.

[20] A. Dabrowski, G. Merzdovnik, N. Kommenda, and E. Weippl, “Browser history stealing with captive wi-fi portals,” in *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 234–240.

[21] A. Das, G. Acar, N. Borisov, and A. Pradeep, “The Web’s sixth sense: A study of scripts accessing smartphone sensors,” in *Proceedings of the 25th ACM Conference on Computer and Communication Security (CCS)*. ACM, 2018. [Online]. Available: https://doi.org/0.1145/3243734.3243860

[22] E. W. Felten and M. A. Schneider, “Timing attacks on web privacy,” in *Proceedings of the 7th ACM conference on Computer and communications security*, 2000, pp. 25–32.

[23] G. Franken, T. Van Goethem, and W. Joosen, “Who left open the cookie jar? a comprehensive evaluation of third-party cookie policies,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 151–168.

[24] A. Janc and L. Olejnik, “Web browser history detection as a real-world privacy threat,” in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 215–231.

[25] S. Karami, P. Ilia, K. Solomos, and J. Polakis, “Carnus: Exploring the privacy threats of browser extension fingerprinting,” in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[26] H. Kim, S. Lee, and J. Kim, “Inferring browser activity and status through remote monitoring of storage usage,” in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 410–421.

[27] B. Kondracki, A. Aliyeva, M. Egele, J. Polakis, and N. Nikiforakis, “Meddling middlemen: Empirical analysis of the risks of data-saving mobile browsers,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 810–824.

[28] M. Kosinski, D. Stillwell, and T. Graepel, “Private traits and attributes are predictable from digital records of human behavior,” *Proceedings of the national academy of sciences*, vol. 110, no. 15, pp. 5802–5805, 2013.

[29] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, “Cross-origin pixel stealing: timing attacks using css filters,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 1055–1062. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516712

[30] M. Lecuyer, R. Spahn, Y. Spiliopoulos, A. Chaintreau, R. Geambasu, and D. Hsu, “Sunlight: Fine-grained targeting detection at scale with statistical confidence,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 554–566.

[31] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, “Pride and prejudice in progressive web apps: Abusing native app-like features in web applications,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1731–1746.

[32] S. Lee, H. Kim, and J. Kim, “Identifying cross-origin resource status using application cache.” in *Network and Distributed System Security Symposium, NDSS*, 2015.

[33] S. Lee, Y. Kim, J. Kim, and J. Kim, “Stealing webpages rendered on your browser by exploiting gpu vulnerabilities,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 19–33.

[34] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, “Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016,” in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[35] X. Lin, P. Ilia, and J. Polakis, “Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 507–519.

[36] F. Marcantoni, M. Diamantaris, S. Ioannidis, and J. Polakis, “A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks,” in *The World Wide Web Conference*, 2019, pp. 3063–3071.

[37] ——, “A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks,” in *30th International World Wide Web Conference, WWW ’19*. ACM, 2019.

[38] L. Olejnik, S. Englehardt, and A. Narayanan, “Battery status not included: Assessing privacy in web standards.” in *IWPE@ SP*, 2017, pp. 17–24.

[39] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, “Master of web puppets: Abusing web browsers for persistent and stealthy computation,” in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[40] J. R. R. Fielding, M. Nottingham, “Hypertext transfer protocol (http/1.1): Caching,” https://httpwg.org/specs/rfc7234.html#heuristic.freshness, June 2014, accessed on 2020-01-05.

[41] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, “Complex security policy? a longitudinal analysis of deployed content security policies.” in *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[42] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, “Busting frame busting a study of clickjacking vulnerabilities on popular sites,” in *Web 2.0 Security and Privacy*. IEEE, 2010.

[43] I. Sanchez-Rola, D. Balzarotti, and I. Santos, “Bakingtimer: privacy analysis of server-side request processing time,” in *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, 2019, pp. 478–488.

[44] P. Skolka, C.-A. Staicu, and M. Pradel, “Anything to hide? studying minified and obfuscated code in the web,” in *The World Wide Web Conference*, 2019, pp. 1735–1746.

[45] M. Smith, C. Disselkoen, S. Narayan, F. Brown, and D. Stefan, “Browser history re:visited,” in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/smith

[46] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, “Browser feature usage on the modern web,” in *Proceedings of the 2016 Internet Measurement Conference*. ACM, 2016, pp. 97–110.

[47] P. Snyder, C. Taylor, and C. Kanich, “Most websites don’t need to vibrate: A cost-benefit approach to improving browser security,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 179–194.

[48] T. Steiner, “What is in a web view: An analysis of progressive web app features when the means of web access is not a web browser,” in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 789–796.

[49] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the web tangled itself: Uncovering the history of client-side web (in) security,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 971–987.

[50] J. Su, A. Shukla, S. Goel, and A. Narayanan, "De-anonymizing web browsing data with social networks," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 1261–1269.

[51] Y. Tian, Y. C. Liu, A. Bhosale, L. S. Huang, P. Tague, and C. Jackson, "All your screens are belong to us: attacks exploiting the html5 screen sharing api," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 34–48.

[52] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1382–1393.

[53] G. Venkatadri, A. Andreou, Y. Liu, A. Mislove, K. P. Gummadi, P. Loiseau, and O. Goga, "Privacy risks with facebook's pii-based targeting: Auditing a data broker's advertising interface," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 89–107.

[54] P. Walton, "Building faster, more resilient apps with service worker (chrome dev summit 2018)," November 2018, accessed on 2020-01-05.

[55] T. Watanabe, E. Shioji, M. Akiyama, and T. Mori, "Melting pot of origins: Compromising the intermediary web services that rehost websites," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[56] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 147–161.

[57] S. Zannettou, B. Bradlyn, E. De Cristofaro, H. Kwak, M. Sirivianos, G. Stringini, and J. Blackburn, "What is gab: A bastion of free speech or an alt-right echo chamber," in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 1007–1014.

[58] S. Zannettou, T. Caulfield, J. Blackburn, E. De Cristofaro, M. Sirivianos, G. Stringhini, and G. Suarez-Tangil, "On the origins of memes by means of fringe web communities," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 188–202.

Appendix

Table VI provides a list of all the API calls that can be used in a SW, and how we map them to the different categories of functionality reported in Section III.

TABLE VI. Service worker capabilities and the corresponding API calls.

Functionality	API calls
Caching	cache.add cache.addAll cache.delete cache.keys cache.match cache.matchAll cache.matchAll cache.put CacheStorage.Delete CacheStorage.has CacheStorage.keys CacheStorage.match CacheStorage.open
Web Push	NotificationEvent.notification PushEvent.data PushManager.getSubscription PushManager.permissionState PushManager.subscribe PushManager.supportedContentEncodings PushMessageData.json PushMessageData.text PushSubscription.endpoint PushSubscription.expirationTime PushSubscription.getKey PushSubscription.options PushSubscription.toJSON PushSubscription.unsubscribe
Fetch	FetchEvent.clientId FetchEvent.preloadResponse FetchEvent.request FetchEvent.respondWith FetchEvent.resultingClientId
Sync	SyncEvent.tag SyncManager.getTags SyncManager.register
SW to client Message	Client.postMessage
Client to SW Message	ServiceWorker.postMessage
importScripts	ServiceWorkerGlobalScope.importScripts