Statement from the NDSS 2021 Program Committee: NDSS is devoted to ethical principles and encourages the research community to ensure its work protects the privacy, security, and safety of users and others involved. While the NDSS 2021 PC appreciated the technical contributions of this paper, it was the subject of a debate in our community regarding the responsible disclosure of vulnerabilities for the Firefox web browser. The PC examined and discussed the ethics concerns raised and the authors' response. Although no harm came to users, the authors' oversight could have made a non-vulnerable browser vulnerable to the attack proposed in the paper. The PC does not believe the authors acted in bad faith. Nevertheless, we decided to add this note as well as the authors' response (in an Appendix) to the paper because the NDSS PC takes both the ethics of responsible disclosure and fairness towards the authors seriously. It is the PC's view that researchers must not engage in disclosure practices that subject users to an appreciable risk of substantial harm. NDSS will work with other conferences to further improve the transparency of vulnerability disclosure to reduce such errors in the future.

Tales of FAVICONS and Caches: Persistent Tracking in Modern Browsers

Konstantinos Solomos, John Kristoff, Chris Kanich, Jason Polakis
University of Illinois at Chicago
{ksolom6, jkrist3, ckanich, polakis}@uic.edu

***Abstract*—The privacy threats of online tracking have garnered considerable attention in recent years from researchers and practitioners. This has resulted in users becoming more privacy-cautious and browsers gradually adopting countermeasures to mitigate certain forms of cookie-based and cookie-less tracking. Nonetheless, the complexity and feature-rich nature of modern browsers often lead to the deployment of seemingly innocuous functionality that can be readily abused by adversaries. In this paper we introduce a novel tracking mechanism that misuses a simple yet ubiquitous browser feature: *favicons*. In more detail, a website can track users across browsing sessions by storing a tracking identifier as a set of entries in the browser's dedicated favicon cache, where each entry corresponds to a specific subdomain. In subsequent user visits the website can reconstruct the identifier by observing which favicons are requested by the browser while the user is automatically and rapidly redirected through a series of subdomains. More importantly, the caching of favicons in modern browsers exhibits several unique characteristics that render this tracking vector particularly powerful, as it is persistent (*not* affected by users clearing their browser data), non-destructive (reconstructing the identifier in subsequent visits does not alter the existing combination of cached entries), and even crosses the isolation of the *incognito* mode. We experimentally evaluate several aspects of our attack, and present a series of optimization techniques that render our attack practical. We find that combining our favicon-based tracking technique with immutable browser-fingerprinting attributes that do not change over time allows a website to reconstruct a 32-bit tracking identifier in 2 seconds. Furthermore, our attack works in all major browsers that use a favicon cache, including Chrome and Safari. Due to the severity of our attack we propose changes to browsers' favicon caching behavior that can prevent this form of tracking, and have disclosed our findings to browser vendors who are currently exploring appropriate mitigation strategies.**

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.2021.24202
www.ndss-symposium.org

I. Introduction

Browsers lie at the heart of the web ecosystem, as they mediate and facilitate users' access to the Internet. As the Web continues to expand and evolve, online services strive to offer a richer and smoother user experience; this necessitates appropriate support from web browsers, which continuously adopt and deploy new standards, APIs and features [76]. These mechanisms may allow web sites to access a plethora of device and system information [55], [21] that can enable privacy-invasive practices, e.g., trackers leveraging browser features to exfiltrate users' Personally Identifiable Information (PII) [24]. Naturally, the increasing complexity and expanding set of features supported by browsers introduce new avenues for privacy-invasive or privacy-violating behavior, thus, exposing users to significant risks [53].

In more detail, while cookie-based tracking (e.g., through third-party cookies [57]) remains a major issue [29], [9], [69], tracking techniques that do not rely on HTTP cookies are on the rise [63], [16] and have attracted considerable attention from the research community (e.g., novel techniques for device and browser fingerprinting [25], [18], [82], [23], [50]). Researchers have even demonstrated how new browser security mechanisms can be misused for tracking [78], and the rise of online tracking [52] has prompted user guidelines and recommendations from the FTC [20].

However, cookie-less tracking capabilities do not necessarily stem from modern or complex browser mechanisms (e.g., service workers [43]), but may be enabled by simple or overlooked browser functionality. In this paper we present a novel tracking mechanism that exemplifies this, as we demonstrate how websites can leverage *favicons* to create persistent tracking identifiers. While favicons have been a part of the web for more than two decades and are a fairly simple website resource, modern browsers exhibit interesting and sometimes fairly idiosyncratic behavior when caching them. In fact, the favicon cache (i) is a dedicated cache that is not part of the browser's HTTP cache, (ii) is not affected when users clear the browser's cache/history/data, (iii) is not properly isolated from private browsing modes (i.e., *incognito* mode), and (iv) can keep favicons cached for an entire year [26].

By leveraging all these properties, we demonstrate a novel persistent tracking mechanism that allows websites to re-identify users across visits even if they are in incognito mode or have cleared client-side browser data. Specifically, websites can create and store a unique browser identifier through a unique combination of entries in the favicon cache. To be more precise, this tracking can be easily performed by any website by redirecting the user accordingly through a series of subdomains. These subdomains serve different favicons and, thus, create their own entries in the Favicon-Cache. Accordingly, a set of N-subdomains can be used to create an N-bit identifier, that is unique for each browser. Since the attacker controls the website, they can force the browser to visit subdomains without any user interaction. In essence, the presence of the favicon for subdomain_i in the cache corresponds to a value of `1` for the i-th bit of the identifier, while the absence denotes a value of `0`.

We find that our attack works against all major browsers that use a favicon cache, including Chrome, Safari, and the more privacy-oriented Brave. We experimentally evaluate our attack methodology using common hosting services and development frameworks, and measure the impact and performance of several attack characteristics. First, we experiment with the size of the browser identifier across different types of devices (desktop/mobile) and network connections (high-end/cellular network). While performance depends on the network conditions and the server's computational power, for a basic server deployed on Amazon AWS, we find that redirections between subdomains can be done within 110-180 ms. As such, for the *vanilla* version of our attack, storing and reading a full 32-bit identifier requires about 2.5 and 5 seconds respectively.

Subsequently, we explore techniques to reduce the overall duration of the attack, as well as selectively assign optimal identifiers (i.e., with fewer redirections) to weaker devices. Our most important optimization stems from the following observation: while robust and immutable browser fingerprinting attributes are not sufficient for uniquely identifying machines at an *Internet-scale*, they are ideal for augmenting low-throughput tracking vectors like the one we demonstrate. The discriminating power of these attributes can be transformed into bits that constitute a portion of the tracking identifier, thus optimizing the attack by reducing the required redirections (i.e., favicon-based bits in the identifier) for generating a sufficiently long identifier. We conduct an in-depth analysis using a real-world dataset of over 270K browser fingerprints and demonstrate that websites can significantly optimize the attack by recreating part of the unique identifier from fingerprinting attributes that do not typically change over time [82] (e.g., Platform, WebGL vendor). We find that websites can reconstruct a 32-bit tracking identifier (allowing to differentiate almost 4.3 Billion browsers) in $\sim$2 seconds.

Overall, while favicons have long been considered a simple decorative resource supported by browsers to facilitate websites' branding, our research demonstrates that they introduce a powerful tracking vector that poses a significant privacy threat to users. The attack workflow can be easily implemented by any website, without the need for user interaction or consent, and works even when popular anti-tracking extensions are deployed. To make matters worse, the idiosyncratic caching behavior of modern browsers, lends a particularly egregious property to our attack as resources in the favicon cache are used even when browsing in incognito mode due to improper isolation practices in all major browsers. Furthermore, our fingerprint-based optimization technique demonstrates the threat and practicality of combinatorial approaches that use different techniques to complement each other, and highlights the need for more holistic explorations of anti-tracking defenses. Guided by the severity of our findings we have disclosed our findings to all affected browsers who are currently working on remediation efforts, while we also propose various defenses including a simple-yet-effective countermeasure that can mitigate our attack.

In summary, our research contributions are:

- We introduce a novel tracking mechanism that allows websites to persistently identify users across browsing sessions, even in incognito mode. Subsequently, we demonstrate how immutable browser fingerprints introduce a powerful optimization mechanism that can be used to augment other tracking vectors.
- We conduct an extensive experimental evaluation of our proposed attack and optimization techniques under various scenarios and demonstrate the practicality of our attack. We also explore the effect of popular privacy-enhancing browser extensions and find that while they can impact performance they do not prevent our attack.
- Due to the severity of our attack, we have disclosed our findings to major browsers, setting in motion remediation efforts to better protect users' privacy, and also propose caching strategies that mitigate this threat.

II. Background & Threat Model

Modern browsers offer a wide range of functionalities and APIs specifically designed to improve the user's experience. One such example are favicons, which were first introduced to help users quickly differentiate between different websites in their list of bookmarks [37]. When browsers load a website they automatically issue a request in order to look up a specific image file, typically referred to as the favicon. This is then displayed in various places within the browser, such as the address bar, the bookmarks bar, the tabs, and the most visited and top choices on the home page. All modern web browsers across major operating systems and devices support the fetching, rendering and usage of favicons. When originally introduced, the icon files had a specific naming scheme and format (*favicon.ico*), and were located in the root directory of a website [8]. To support the evolution and complex structure of modern webpages, various formats (e.g., *png, svg*) and sizes are supported, as well as methods for dynamically changing the favicon (e.g., to indicate a notification), thus providing additional flexibility to web developers.

To serve a favicon on their website, a developer has to include an `<link rel>` attribute in the webpage's header [84]. In general, the `rel` tag is used to define a relationship between an HTML document and an external resource like an image, animation, or JavaScript. When defined in the header of the

TABLE I: Example of Favicon Cache content and layout.

Entry ID	Page URL	Favicon ID	TTL	Dimensions	Size
1	foo.com	favicon.ico	50000	16 X 16	120
2	xyz.foo.com	fav_v2.ico	10000	32 X 32	240
3	foo.com/path	favicon.ico	25500	16 X 16	120

HTML page, it specifies the file name and location of the icon file inside the web server's directory [59], [83]. For instance, the code in Listing 1 instructs the browser to request the page's favicon from the "resources" directory. If this tag does not exist, the browser requests the icon from the predefined webpage's root directory. Finally, a link between the page and the favicon is created only when the provided URL is valid and responsive and it contains an icon file that can be properly rendered. In any other case, a blank favicon is displayed.

Listing 1: Fetching the favicon from a custom location.

```
<link rel="icon" href="/resources/favicon.ico"
 type="image/x-icon">
```

As any other resource needed for the functionality and performance of a website (e.g., images, JavaScript), favicons also need to be easily accessed. In modern web browsers (both desktop and mobile) these icons are independently stored and cached in a separate local database, called the Favicon Cache (*F-Cache*) which includes various primary and secondary metadata, as shown in Table I. The primary data entries include the Visited URL, the favicon ID and the Time to Live (TTL). The Visited URL stores the explicitly visited URL of the active browser tab, such as a subdomain or an inner path under the same base domain (i.e., eTLD+1). These will have their own cache entries whenever a different icon is provided. While this allows web developers to enhance the browsing experience by customizing the favicons for different parts of their website, it also introduces a tracking vector as we outline in §III.

Moreover, as with other resources typically cached by browsers, the favicon TTL is mainly defined by the *Cache-Control, Expires* HTTP headers. The value of each header field controls the time for which the favicon is considered "fresh". The browser can also be instructed to not cache the icon (e.g., `Cache-Control: no-cache/no-store`). When none of these headers exists, a short-term expiration date is assigned (e.g., 6 hours in Chrome [5]). The maximum time for which a favicon can be cached is one year. Finally, since favicons are also handled by different browser components, including the Image Renderer for displaying them, the F-Cache stores other metadata including the dimensions and size of each icon, and a timestamp for the last request and update.

Caching Policies. Once a resource is stored in a cache, it could theoretically be served by the cache forever. However, caches have finite storage so items are periodically removed from storage or may change on the server so the cache should be updated. Similar to other browser caches, F-Cache works under the HTTP client-server protocol and has to communicate with the server to add, update or modify a favicon resource. More specifically, there is a set of Cache Policies that define the usage of the F-Cache in each browser. The basic rules are:

Create Entry. Whenever a browser loads a website, it first reads the icon attribute from the page header and searches the F-Cache for an entry for the current page URL being visited. If no such entry exists, it generates a request to fetch the resource from the previously read attribute. When the fetched resource is successfully rendered, the link between the page and the favicon is created and the entry is committed to the database along with the necessary icon information. According to Chrome's specification [5] the browser commits all new entries and modifications of every linked database (e.g., favicon, cookies, browsing history) every 10 seconds.

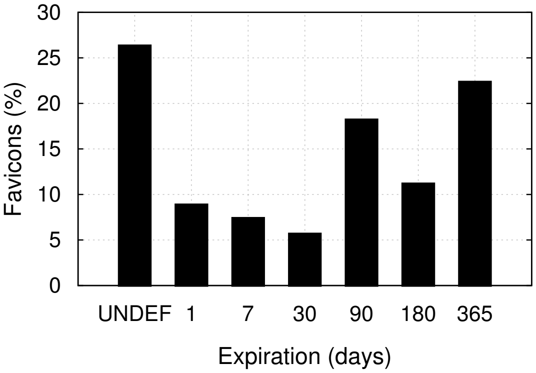


Fig. 1: Expiration of favicon entries in the top 10K sites.

Conditional Storage. Before adding a resource to the cache, the browser checks the validity of the URL and the icon itself. In cases of expired URLs (e.g., a 404 or 505 HTTP error is raised) or non-valid icon files (e.g., a file that cannot be rendered) the browser rejects the icon and no new entry is created or modified. This ensures the integrity of the cache and protects it from potential networking and connection errors.

Modify & Delete Entry. If the browser finds the entry in the cache, it checks the TTL to verify the freshness of the resource. If it has not expired, the browser compares the retrieved favicon ID with the one included in the header. If the latter does not match the already stored ID (e.g., `rel=``/fav_v2.ico"`) it issues a request and updates the entry if the fetch succeeds. This process is also repeated if the TTL has expired. If none of these issues occur, the favicon is retrieved from the local database.

Access Control and Removal. The browser maintains a different instance of the F-Cache for each user (i.e., browser account/profile) and the only way to delete the entries for a specific website is through a hard reset [33]. Common browser menu options to clear the browser's cache/cookies/history do not affect the favicon cache, nor does restarting or exiting the browser. Surprisingly, for performance and optimization reasons this cache is also used when the user is browsing in incognito mode. As opposed to other types of cached and stored resources which are completely isolated when in incognito mode for obvious privacy reasons [85], browsers only partially isolate the favicon cache. Specifically, the browser will access and use existing cached favicons (i.e., there is read permission in incognito mode), but it will not store any new entries (i.e., there is no write permission). As a result, the attack that we demonstrate allows websites to re-identify any incognito user that has visited them even once in normal browsing mode.

Favicon use in the wild. To better understand how favicons are used in practice, we conduct a crawl in the Alexa [12] top 10K using the Selenium automation framework [72], with Chrome. Since some domains are associated with multiple subdomains that might not be owned by the same organization or entity (e.g, wordpress.com, blogspot.com) we also explore how favicon use changes across subdomains. As such, for each website, we perform a DNS lookup to discover its subdomains using the c tool, and also visit the first 100 links encountered while crawling the website. Subsequently, we visit all collected URLs and log the HTTP requests and responses as well as any changes in the browser's favicon cache. We find that 94% of the domains (i.e., eTLD+1) have valid favicon resources, which is an expected branding strategy from popular websites.

Next, we use an image hashing algorithm [41] to measure how often websites deploy different favicons across different parts and paths of their domain. We find that 20% of the websites actually serve different favicons across their subdomains. While different subdomains may belong to different entities and, thus, different brands, the vast majority of cases are due to websites customizing their favicons according to the content and purpose of a specific part of their website. Figure 1 reports the expiration values of the collected favicons. As expected, favicon-caching expiration dates vary considerably. Specifically, 9% of the favicons expire in less than a day, while 18% expire within 1 to 3 months, and 22% have the maximum expiration of a year. Finally, for ∼27% of the favicons a cache-control directive is not provided, resulting in the default expiration date (typically 6 hours) of the browser being used.

A. Threat Model

Our research details a novel technique for tracking users by creating a unique browser identifier that is "translated" into a unique combination of entries in the browser's favicon cache. These entries are created through a series of controlled redirections within the attacker's website. As such, in our work the adversary is any website that a user may visit that wants to re-identify the user when normal identifiers (e.g., cookies) are not present. Furthermore, while we discuss a variation of our attack that works even when JavaScript is disabled, we will assume that the user has JavaScript enabled since we also present a series of optimizations that significantly enhance the performance and practicality of our attack by leveraging robust browser-fingerprinting attributes (which require JavaScript).

III. Methodology

In this section, we provide details on the design and implementation of our favicon-based tracking attack.

Overview & Design. Our goal is to generate and store a unique persistent identifier in the user's browser. At a high level, the favicon cache-based attack is conceptually similar to the HSTS supercookie attack [78], in that full values cannot be directly stored, but rather individual bits can be stored and retrieved by respectively setting and testing for the presence of a given cache entry. We take advantage of the browser's favicon caching behavior as detailed in the previous section, where different favicons are associated with different domains or paths of a base domain to associate the unique persistent identifier to an individual browser. We express a binary number (the ID) as a set of subpaths, where each bit represents a specific path for the base domain, e.g., domain.com/A corresponds to the first bit of the ID, domain.com/B to the second bit, etc. Depending on the attackers' needs in terms of scale (i.e., size of user base) the number of inner paths can be configured for the appropriate ID length. While the techniques that we detail next can also be implemented using subdomains, our prototype uses subpaths (we have experimentally verified that the two redirection approaches do not present any discernible differences in terms of performance).

Algorithm 1: Server side process for writing/reading IDs. This process runs independently for each browser visit.

```
Input: HTTPS traffic logged in web server.
Output: ID of visited browser.
ID_Vector =[N* 1] // init N-bit vector
read_mode=write_mode=False
if Request== GET : main page then
    if Next_Request == GET : favicon.ico then
        write_mode= True
    else
        read_mode= True
if write_mode==True then
                                        /* Write Mode */
    ID_Vector =Generate_ID
    // ID Bits mapping to Subpaths
    Redirection_Chain = Map [ID_Vector]
    foreach path in Redirection_Chain do
        Redirect_Browser (path)
        waitForRedirection()
        if Request == GET : faviconX.ico then
            // Write Bit
            Response = faviconX.ico
else if read_mode==True then
                                        /* Read Mode */
    foreach path in All_Paths() do
        Redirect_Browser (path)
        waitForRedirection()
        if Request == GET : faviconX.ico then
            // Log the absence of the Bit
            ID_Vector[path]=0
            Response = [404 Error]
return ID_Vector
```

Following this general principle, we first translate the binary vector into subpaths, such that every path represents a bit in the N-bit vector. For example, assume that we generate an arbitrary 4-bit ID as a vector: $\mathcal{ID}$ =<0101>. This vector has to be translated into a sequence of available paths, which requires us to define a specific ordering (i.e., sequence) of subpaths: $\mathcal{P}$ =<A, B, C, D>. The mapping is then straightforward, with the first index of $\mathcal{ID}$ - the most significant bit in the binary representation - mapped to the first subpath in $\mathcal{P}$. This one-to-one mapping has to remain consistent even if the attacker decides to increase the length of possible identifiers in the future, as doing so will allow the site to accommodate for more users (by appending additional subpaths in the $\mathcal{P}$ vector).

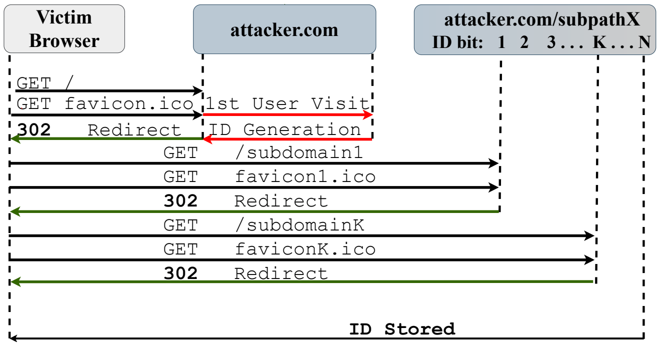


Fig. 2: Writing the identifier.

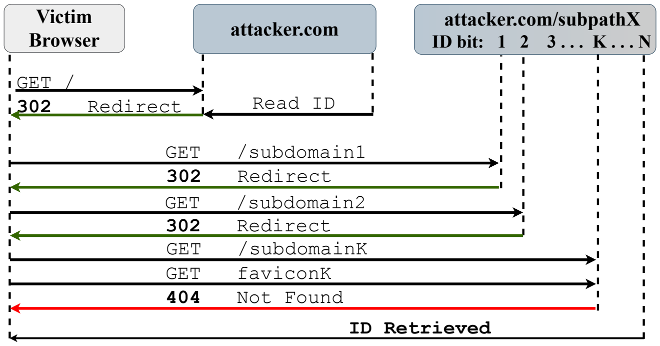


Fig. 3: Reading the identifier.

The next step is to ensure that the information carried by the identifier is "injected" into the browser's favicon cache. The key observation is that each path creates a unique entry in the browser favicon cache *if* it serves a *different* favicon than the main page. As such, we configure different favicons and assign them to the corresponding paths. Each path has its own favicon configured in the header of its HTML page, which is fetched and cached once the browser visits that page. The presence of a favicon entry for a given path denotes a value of `1` in the identifier while the lack of a favicon denotes a `0`.

To store the ID, a victim needs only to visit the paths `{B,D}`, which results in storing *faviconB.ico* and *faviconD.ico* (the customized favicons of each paths). In the visits, the user will be redirected through *all* subpaths. Since they have already visited the sub-pages (B, D), their favicons are stored in the browser's cache and will not be requested from the server. For the remaining domains (A, C) the browser will request their favicons. Here we take advantage of the browsers' caching policies, and serve invalid favicons; this results in no changes being made to the cache for the entire base domain and the stored identifier will remain unchanged.

In other words, our cache-based tracking attack is non-destructive and can be successfully repeated in all subsequent user visits. Finally, a core aspect of the attack is the `redirection threshold`, which defines the time needed for the browser to visit the page, request the favicon, store it into the cache and proceed to the next subpath. A high-level overview of our proposed methodology is given in Algorithm 1 and is further detailed in the next subsections.

A. Write Mode: Identifier Generation & Storage

In the write mode, our goal is to first make sure that the victim has never visited the website before and to then generate and store a unique identifier. Since we control both the website and the server, we are able to control and track which subpaths are visited as well as the presence or absence of specific favicons by observing the HTTP requests received by the server. The succession of requests during the write mode is illustrated in Figure 2. The first check is to see whether the favicon for the base domain is requested by the server when the user visits the page. If that favicon is requested, then this is the user's first visit and our system continues in write mode. Otherwise it switches to read mode. Next, we generate a new N-bit ID that maps to a specific path *Redirection Chain*. Specifically, we create a sequence of consecutive redirections through any subpaths that correspond to bits with a value of `1`, while skipping all subpaths that correspond to `0`. Each path is a different page with its own HTML file and each HTML page contains a valid and unique favicon.

The redirection chain is transformed to a query string and passed as a URL parameter. Each HTML page, then, includes JavaScript code that parses the URL parameter and performs the actual redirection after a short timing redirection threshold (`waitForRedirection()` in Algorithm 1). The redirection is straightforward to execute by changing the `window.location.href` attribute. For instance, for the ID `0101` we create the Redirection_Chain= [B→D] and the server will generate the query `domain?id=bd`. Finally, when the server is in write mode it responds normally to all the requests and properly serves the content. Once the redirection process completes, the ID will be stored in the browser's favicon cache.

B. Read Mode: Retrieve Browser Identifier

The second phase of the attack is the reconstruction of the browser's ID upon subsequent user visits. The various requests that are issued during the read mode are shown in Figure 3. First, if the server sees a request for the base domain without a corresponding request for its favicon, the server reverts to read mode behavior since this is a recurring user. When the server is in read mode, it *does not* respond to *any* favicon request (it raises a `404 Error`), but responds normally to all other requests. This ensures the integrity of the cached favicons during the read process, as no new F-Cache entry is created nor are existing entries modified.

In practice, to reconstruct the ID we need to force the user's browser to visit all the available subpaths, and capture the generated requests. This is again possible since we control the website and can force redirections to all available subpaths in the Redirection_Chain through JavaScript. Contrary to the write mode, here the set of redirections contains all possible paths. In our example we would reconstruct the 4-Bit ID by following the full redirection chain [A→B→C→D].

In the final step, the server logs all the requests issued by the browser; every request to a subpath that is not accompanied by a favicon request indicates that the browser has visited this page in the past since the favicon is already in the F-Cache, and we encode this subpath as `1`. The other subpaths are encoded as `0` to capture the absence of this icon from the cache. Following

the running example where the ID is 0101, the browser will issue the following requests:
[GET /A, GET /faviconA, GET /B, GET /C, GET /faviconC, GET /D]. Notice here that for two paths we do not observe any requests (info bit: 1) while there are requests for the first and third path (info bit: 0).

Concurrent users. Since any website can attract multiple concurrent users, some of which may be behind the same IP address (e.g., due to NAT) in the first step when the user visits the website, we set a temporary "session" cookie that allows us to group together all incoming requests on the server that originate from the specific browser. It's important to note that our attack is not affected by the user clearing their cookies before and/or after this session (or are browsing in incognito mode) since this cookie is only needed for associating browser requests in this specific session. Furthermore, since this is a first-party session cookie it is not blocked by browsers' and extensions' anti-tracking defenses.

C. Scalability

Dynamic identifier lengths. As each subpath redirection increases the duration of the attack, websites can reduce the overall overhead by dynamically increasing the length of the N-bit identifier whenever a new user arrives and all possible identifier combinations (2^N) for the current length have already been assigned. This is trivially done by appending a new subpath in the sequence of subpaths and appending a "0" at the end of all existing user identifiers. In our running example, if the server goes from 4-bit identifiers to 5-bit identifiers, the subpath vector will become $\mathcal{P}$ =<A, B, C, D, E> and the identifier 0101 will become 01010, without any other changes necessary. This results in the website only using the minimum number of redirections necessary. While there is no inherent limitation to the maximum length of our identifier, we consider 32 bits suitable even for the most popular websites since 32 bits allow for almost 4.3 Billion unique identifiers.

D. Selective Identifier Reconstruction

As already discussed, our attack is not dependent on any stateful browser information or user activity, but only leverages the data stored in F-Cache. In general, the process of writing and reading the unique tracking identifier can be considered costly due to the page redirections that are performed. Especially the read phase which reconstructs the ID by redirecting through the full subpath sequence chain should only take place when necessary and not upon every user visit, i.e., when no other stateful browser identifier is available. This can be easily addressed by the use of a typical cookie that stores an identifier. This way, the website only needs to reconstruct the tracking identifier when the original request to the main page does not contain this cookie (e.g., because the user cleared all their cookies or is in incognito mode) thus removing any unnecessary overhead.

E. Vulnerable Browsers

We perform a series of preliminary experiments to identify which browsers are affected by our attack, and select the most popular browsers and major operating systems. For these experiments we visit our own attack website multiple times for each browser and OS combination and monitor the requests issued by the browser as well as the entries created in the favicon cache so as to identify potential inconsistencies.

TABLE II: Compatibility of the attack across different platforms and browsers. Combinations that do not exist are marked as N/A.

Browser	**Windows**	**macOS**	**Linux**	**Android**	**iOS**
Chrome (v. 86.0)	✓	✓	✓	✓	✓
Safari (v. 14.0)	N/A	✓	N/A	N/A	✓
Edge (v. 87.0)	✓	✓	N/A	✓	N/A
Brave (v. 1.14.0)	✓	✓	✓	✓	✓

TABLE III: Attack effectiveness under different scenarios: when the user is browsing in private mode (*Incognito*), after clearing the browser's user data (*Clear Data*), after installing anti-tracking extensions (*Anti-Tracking*), and using a *VPN*.

Browser	**Incognito**	**Clear Data**	**Anti-Tracking**	**VPN**
Chrome	✓	✓	✓	✓
Safari	✓	✓	✓	✓
Edge	✓	✓	✓	✓
Brave	✓	✓	✓	✓

Table II presents the browsers that we found to be susceptible to our attack. In more detail, our attack is applicable on all platform and browser combinations where the favicon cache is actually used by the browser (we detail a bug in Firefox next). Chrome, by far the most popular and widely used browser, is vulnerable to our attack on all the supported operating systems that we tested. We also identified the same behavior for Brave and Edge, which is expected as they are both Chromium-based and, thus, share the same browser engine for basic functionalities and caching policies. We note that since the F-Cache policies tend to be similar across different browser vendors, the attack is most likely feasible in other browsers that we have not tested.

Next, we also experimentally investigate whether our attack is affected by normal defensive actions employed by users. Specifically, we explore the effect of re-visiting a website in incognito mode, clearing the browser's user data (e.g., using the "Clearing Browsing Data" setting in Chrome) and installing popular anti-tracking and anti-fingerprinting extensions. As can be seen in Table III, the attack works against users in incognito mode in all the tested browsers as they all read the favicon cache even in private browsing mode (most likely for performance optimization reasons). Similarly, we find that the option for clearing the user's browsing data has no effect on the attack as the favicon cache is not included in the local storages that browsers clear. Moreover, we find that installing popular privacy extensions that are available in most platforms (i.e., Ghostery, UBlock, Privacy Badger[1]) does not protect users from our attack, which is expected since our attack presents the first privacy-invasive misuse of favicons. Finally, we also verify that if the user visits the website using a VPN the attack is still effective, as the user's IP address does not affect the favicon cache.

[1] Not available for Safari.

Firefox. As part of our experiments we also test Firefox. Interestingly, while the developer documentation and source code include functionality intended for favicon caching [27] similar to the other browsers, we identify inconsistencies in its actual usage. In fact, while monitoring the browser during the attack's execution we observe that it has a valid favicon cache which creates appropriate entries for every visited page with the corresponding favicons. However, it *never* actually uses the cache to fetch the entries. As a result, Firefox actually issues requests to re-fetch favicons that are already present in the cache. We have reported this bug to the Mozilla team, who verified and acknowledged it. At the time of submission, this remains an open issue. Nonetheless, we believe that once this bug is fixed our attack will work in Firefox, unless they also deploy countermeasures to mitigate our attack (we provide more details on our attack's disclosure in §VII).

IV. Attack Optimization Strategies

In this section we propose different strategies that can be applied to improve our attack's performance without affecting accuracy or consistency.

A. Weak devices: Identifier Assignment Strategy

Our first strategy is straightforward and aims to reduce the overhead of the write phase (i.e., storing the identifier) on a per-client basis. Specifically, our goal is to assign identifiers that require fewer redirections (i.e., have fewer `1`s) to resource-constrained devices. While this approach does not provide an optimization for the website at an aggregate level, since all identifiers for a given number of bits will be assigned to users, it allows the website to selectively/preferentially assign "better" identifiers to devices with computational constraints (e.g., smartphones) or devices that connect over high-latency networks (e.g., cellular) to reduce the redirections. Currently, websites can leverage the User-agent header for this, e.g., to infer if users are on mobile devices or have an older browser version. However, an experimental browser feature designed to optimize content selection and delivery, the Network Information API, is currently supported by several major browsers [6]), allowing websites to also decide based on the nature of the device's connection (e.g., if it is over a cellular network).

For this process, we need an algorithm for sorting IDs that creates a different arrangement of the `1` bits in an ID - the bits that are written through redirection- and assigns them accordingly. In the vanilla version of our attack, for each new client, we simply assign the next available binary identifier based on the number of identifiers assigned so far and increase the identifier's length when necessary. This assignment follows a simple decimal system enumeration, where the sequence of values follows a simple progression:

$$\mathcal{X}=[\texttt{01,10,11,100,101,110,111,1000,...}]$$

As such the ID represents the "arrival" order of each user's initial visit to the website. To put it simply, the first user is assigned the ID=`01`, the second ID=`10`, and so on. To optimize our ID assignment strategy we use a sorting heuristic. Having a constant number of bits in the ID, the "ascending" algorithm permutes the standard binary IDs and sorts them by the total number of `1`s. This results in generating the same set of IDs but in a different sequence. When new users visit the website, constrained devices will be assigned the next available identifier from the top of the sequence (i.e., with fewer `1`s) while more powerful devices or on high-speed networks are assigned from the bottom of the sequence (i.e., with more `1`s.) As we show in §V this approach can reduce the duration of the write phase for constrained devices, especially for websites with larger user bases that require longer identifiers.

B. Adaptive Redirection Threshold

While our previous optimization focuses on the write mode for weak devices and involves the internals of our attack, here we outline a different technique that optimizes the attack's overall performance. As defined in §III, the timing threshold between the visits of each path is directly connected to the attack's duration. Selecting this threshold is, thus, crucial since an unnecessarily large value (e.g., 1 second) will greatly affect the attack's stealthiness and practicality. On the other hand, if the redirection threshold is too low (e.g., 10 ms), there will be insufficient time for the browser to issue the request, receive a response from the server, and store the favicon. Various factors and constraints can affect the optimal threshold for a specific user, including the user's browser, network connection, and device characteristics. For instance, the attack should adopt a higher threshold for mobile devices on a cellular connection, compared to a desktop connecting from a residential network. Furthermore, as we extensively explore in §V, the attack can be further optimized by setting a lower threshold for clients in the same geographic region or network.

C. Leveraging Immutable Browser Fingerprints

Moving a step further, we outline another method that optimizes the attack's overall performance for *all* users. For this, we rely on our following key observation: while browser fingerprinting techniques do not typically provide sufficient discriminatory information to uniquely identify a single device at an Internet scale, they can be used to augment other tracking techniques by subsidizing part of the tracking identifier. Numerous studies have demonstrated various fingerprinting techniques for constructing a persistent identifier based on a set of browser and system attributes [82], [50], [17], [32], [71]. These attributes are commonly collected through JavaScript APIs and HTTP headers and form a set of system characteristics that vary across different browsers, devices and operating systems. Each of these features encodes different types of information, and Shannon's notion of entropy can be used to quantify the discriminatory power of the information that they carry (as bits of entropy). Intuitively, higher levels of measured entropy denote more information being stored in the variable. When focusing on fingerprinting attributes, high entropy represents features that encode information about larger spaces of potential values, while lower entropy is found in the features with smaller value ranges. For instance, features that store binary information (`Cookies Enabled, Use of Local Storage`) have lower entropy values in comparison to attributes that encode a wider range of values (e.g., `Platform/OS, WebGL metadata`).

However, one crucial characteristic of browser fingerprints is that certain fingerprinting attributes are volatile and frequently change, thus reducing their suitability for long-term tracking. Indeed, Vastel et al. [82] found that features with

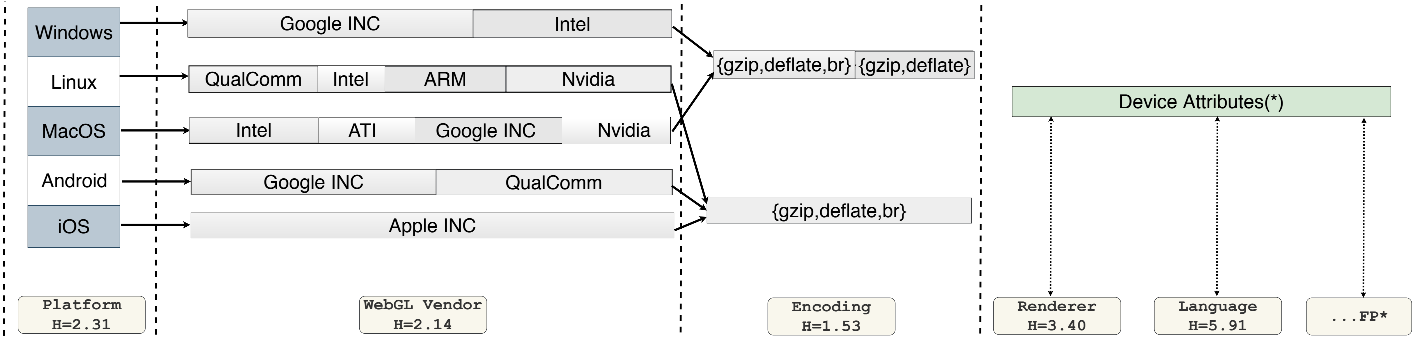


Fig. 4: Ordering of robust fingerprinting attributes and corresponding values from our real-world fingerprinting dataset (See §V). The "*" refers to the remaining attributes (FP) and which are not visualized here.

higher entropy, like the display resolution, timezone, browser fonts, plugins, are more likely to change due to common user behavior. Such examples can be users that travel a lot (different timezones) or install/disable plugins based on their needs. These changes are reflected in the attributes, thus, altering the browser fingerprint over time. To overcome this obstacle and enable long-term tracking, our strategy is to use a set of robust features that remain immutable over time, and use them as part of our tracking identifier. Table IV presents the browser attributes that rarely change along with the measured values of entropy, and the total entropy of those values accumulated, as reported by prior studies in the area. The reported entropy values vary as each study recruited different types and numbers of users (e.g., one study involves privacy-aware users), and implemented different approaches to collect those data.

Nonetheless, the general properties of each attribute remain consistent, e.g., binary attributes have the lowest entropy. Moreover, as shown in prior work [82] the first 4 attributes are constant over time, while the remaining 6 rarely change for a small amount of users (≈10%) over a one-year period. This is expected considering the fact that if the Platform or any of the WebGL features alter, in essence the device becomes different and cannot be treated as the same browsing instance. Moreover, users' browsing preferences, like disabling an AdBlocker [50], [22], [32] or accepting cookies are unlikely to change.

We use these robust attributes and the numbers reported in representative prior work to calculate the total entropy of these features in bits, which we will use to create a K-bit immutable identifier that will be used to subsidize K bits from our favicon-based identifier, thus reducing the number of redirections required during our attack. Based on Table IV, the entropy that we can obtain from these robust attributes varies between 21-26 bits. While this approach adds a new layer of complexity to the attack, it significantly optimizes the performance of our attack as we demonstrate in §III.

TABLE IV: Persistent browser attributes and their entropy reported in the AmIUnique [50], (Cross-)Browser fingerprinting [17] and Hiding in the Crowd [32] studies.

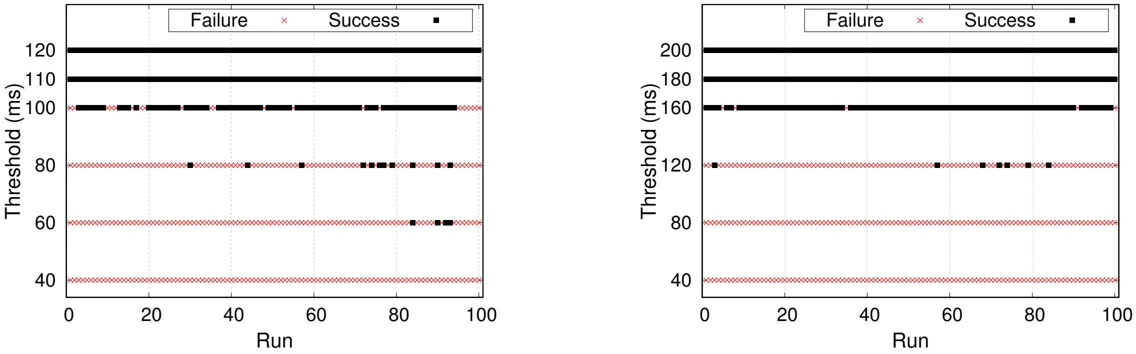
Attribute	AmIUnique	Cross-Browser	Crowd
Cookies Enabled	0.25	0.00	0.00
Local Storage	0.40	0.03	0.04
Do Not Track	0.94	0.47	1.19
Ad Blocker	0.99	0.67	0.04
Platform	2.31	2.22	1.20
Content Encoding	1.53	0.33	0.39
Content Language	5.91	4.28	2.71
WebGL Vendor	2.14	2.22	2.28
WebGL Renderer	3.40	5.70	5.54
Canvas	8.27	5.71	8.54
Total	**26.14**	**21.63**	**21.93**

Combining favicons and fingerprints. Having identified that browser attributes can be used to decrease the favicon identifier's size, we further investigate this strategy and provide a concrete methodology. In more detail, each attribute encodes information that is common with a number of other devices which results in the formation of anonymity sets, i.e., multiple devices with the same fingerprint. In our case, where we use a subset of the 17 attributes that are usually collected to form a fingerprint, the chance of creating a signature that is not unique is higher. Also, it is important to note that the collection of certain device attributes may be blocked by privacy-oriented browser extensions or even as part of a browser's normal operation (e.g., in Brave).

This necessitates a methodology for generating identifiers that dynamically decides how many identifier bits will be obtained from a specific browser's fingerprints based on their availability and discriminating power (i.e., their entropy as calculated for a website's aggregate user base). More concretely, we define the following:

- $\mathcal{V}$: set of browser attributes.
- $\mathcal{W}$: distribution of values of vector V.
- $\mathcal{FP}_{\mathcal{ID}}$: fingerprint-based ID with a length of K bits.
- $\mathcal{FV}_{\mathcal{ID}}$: favicon-based ID with a length of J bits.
- $\mathcal{T}_{\mathcal{ID}}$: unique tracking ID with a length of N bits.

In general, we assume that each attribute in $\mathcal{V}$ has a range or set of possible values that are not uniformly distributed. For instance, in our dataset (described in §V) most users are actually on a Linux platform and, of those, the vast majority (∼ 80%) has a specific WebGL Vendor. These frequency distributions are expressed as a normalized weight $\mathcal{W}$ that captures the portion of the data that each value has, over the entire set of possible values. While in our analysis we use



(a) Desktop

(b) Mobile Device

Fig. 5: Favicon-caching outcome for different redirection thresholds.

per-attribute entropy calculations based on prior studies and a real-world dataset, we assume that individual websites will tweak those values based on their own user base allowing them to more accurately infer $\mathcal{H}$. Since set $\mathcal{V}$ may not contain all ten browser attributes for certain users, its measured entropy $\mathcal{H}$ will vary based on the availability of attributes. Taking all these variables into consideration, we define the following relationships:

$$\mathcal{H}(\mathcal{V},\mathcal{W}) \rightarrow \mathcal{FP}_{\mathcal{ID}}$$
$$\mathcal{FP}_{\mathcal{ID}} \mathbin{+\!\!+} \mathcal{FV}_{\mathcal{ID}} \rightarrow \mathcal{T}_{\mathcal{ID}}$$

Our proposed attack relies on the generation of the two different identifiers with a combined length of N, where N=[2,32] depending on the size of each websites' user base.

In practice, the website will generate the unique tracking ID $\mathcal{T}_{\mathcal{ID}}$ as follows. First, the website will define a standard ordering of the attributes which are used as fingerprint inputs – an example of conceptual visualization along with corresponding attribute values is shown in Figure 4. When a new user arrives, the website will retrieve all available attributes $Attr_i \in \mathcal{V}$ and obtain their hashed representation. These hashes are concatenated into a single string following the aforementioned standard ordering, with any missing attributes skipped, and then converted into a hash. Subsequently, the website calculates the total discriminating power (i.e., entropy) of the available attributes for that specific user and rounds that down to the next whole bit to calculate K. Then it truncates the hash to its K most significant bits to create $\mathcal{FP}_{\mathcal{ID}}$ which, essentially, is a coarse identifier that corresponds to the pool of users that share that specific set of fingerprinting-attribute values (i.e., an anonymity set). Finally, the website calculates $\mathcal{FV}_{\mathcal{ID}}$ to match the next available to-be-assigned identifier $\mathcal{T}_{\mathcal{ID}}$ of length N, and stores the favicon entries that correspond to $\mathcal{FV}_{\mathcal{ID}}$ in the user's favicon cache as described in §III.

V. Evaluation

In this section we provide an experimental evaluation of our attack that explores the practicality and performance of several dimensions of our attack under different realistic scenarios, and also measures the performance improvement obtained by our optimization techniques.

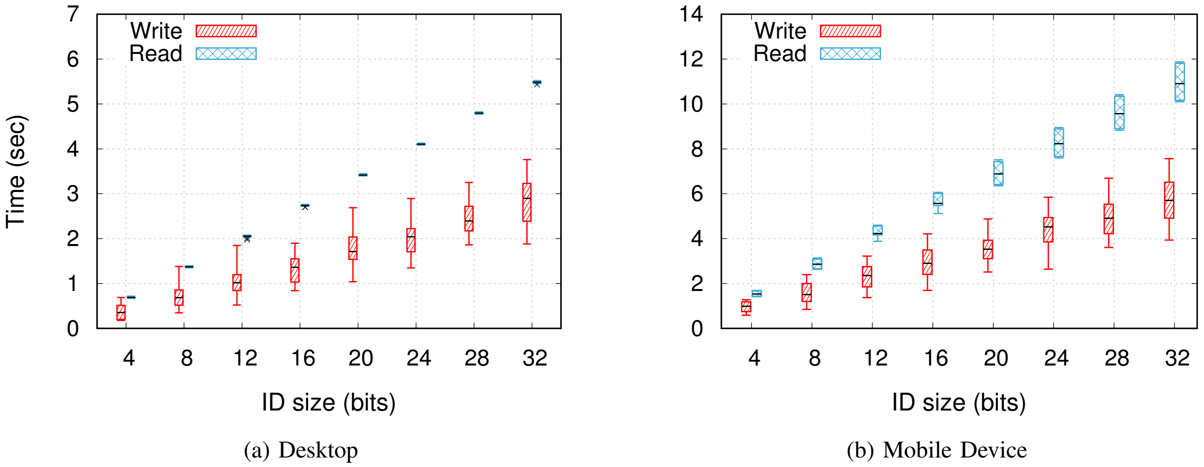
A. Experimental Setup and Methodology

Server & Frameworks. To perform our experiments we first deploy an attack website in the AWS Lightsail environment [11]. We use a dedicated Virtual Machine to minimize potential overhead due to congested system resources. Specifically, our server was built on top of a Quad Core Intel i7-7700 with 32GB of RAM. We also registered a domain name to ensure that our measurements include the network latencies of a realistic attack scenario (i.e., DNS lookup etc).

We opted to locate our VM and DNS zone in the same geographical region with our user devices, to replicate a reasonable scenario where the tracking website leverages a geographically-distributed CDN infrastructure to minimize the distance between their servers and users. However, since AWS does not offer a hosting service in our own state, we select the closest option (distance ∼350 miles) for our main experiments.

We implemented our website using the Python Flask Framework [79], a popular and lightweight framework for deploying web applications.The web application is configured under an Nginx server [7] that acts as a reverse proxy and load balancer, and communicates with the main application and the browser. Our server runs on Ubuntu 18.04 LTS, using a dedicated static IP address. To make the website accessible for all the tested devices and frameworks, we registered an official domain with a valid HTTPS certificate. We believe that even a modest website can recreate (or even significantly augment) this setup by deploying more powerful servers and different combinations of web development tools and frameworks.

Clients. We leveraged Selenium [72] to orchestrate browsers that pose as desktop users that visit our attack website. We used an off-the-shelf desktop with a 6-core Intel Core i7-8700, 32GB of RAM, connected to our university's network. Every experiment consists of the automated browser visiting the attack website two distinct times so as to capture both phases of the attack; in the first visit the website generates and stores the tracking identifier (*write mode*), while in the second visit it reconstructs it (*read mode*). For every phase we measure the time required for the users' browser to complete the chain of redirections through the base domains subpaths and the server to write or read the identifier. Since we do not include any other resources on the website, the favicon is fetched once the request for the main page completes



(a) Desktop

(b) Mobile Device

Fig. 6: Performance evaluation for the two stages of the attack for a desktop and mobile device.

successfully. For the mobile device experiments, we used a low-end mobile device (Xiaomi Redmi Note 7) connected to the cellular network of a major US provider. To automate the experiments we use the Appium framework [28], which allows the automation of both real and emulated mobile devices. All the measurements that we present, consist of 500 repetitions for each given configuration, unless stated otherwise.

B. Redirection Threshold Selection

First, we need to identify a suitable value for the threshold between redirections, as too small a value can result in the browser not fetching and caching the favicon while larger values will unnecessarily increase the attack's duration. As such, we experimentally explore different threshold values, as shown in Figure 5, using both our desktop and mobile device setups. Here we label a specific iteration as a `Success` if (i) the browser visits the landing page and successfully requests the favicon, (ii) the server issues a valid response, and (iii) the browser stores the favicon and redirects to an inner path. If any of those steps fail, we label this iteration as a `Failure`. Our results show that a threshold of 110 ms is sufficient for the browser to always successfully request and store the favicon resource on the desktop device. Comparatively, an increased redirection threshold of 180 ms is optimal for the mobile device – this is expected due to differences in the computational capabilities and network connections between the two setups. We use these threshold values in the remainder of the experiments, unless stated otherwise.

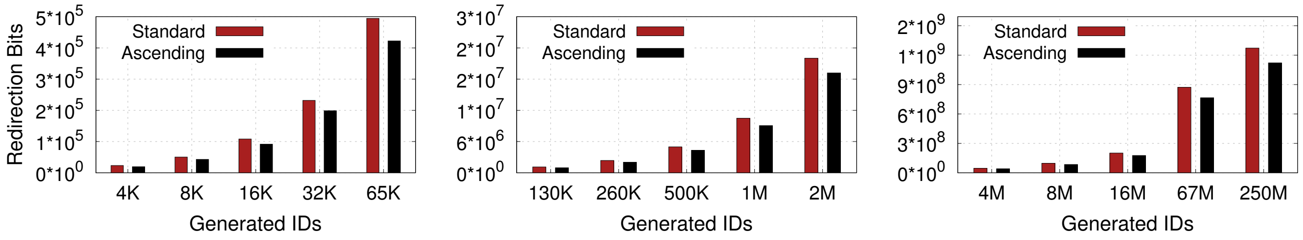
C. Attack Performance

Next we measure various aspects of our attack's performance. For our experiments we use Chrome as it is the most prevalent browser. First, we conduct 500 successive runs of our attack for varying identifier lengths between 2 and 32 bits – recall that websites can dynamically increase the length to accommodate an increasing user base. The results are illustrated in Figure 6.

Desktop browser. The performance measurements for the desktop browser are given in Figure 6a. Considering the nature of the attack, the time required for the write phase is affected by the number of `1` bits as that denotes the number of redirections. This is clearly reflected in the distribution of execution times for each ID size, with the range of variance also slightly increasing as the ID length increases. Nonetheless, even for a 32-bit identifier the median time needed to store the identifier is only 2.47 seconds. While the write phase poses a one-time cost, since this is only needed the first time the user visits the website, our optimization techniques can vastly improve performance. If the website leverages the user's browser fingerprints, assuming 20 bits of entropy are available (which is less than what has been reported by prior studies, as shown in Table IV), then the 12 remaining identifier bits can be stored in the favicon cache in approximately one second.

Figure 6a also reports the experimental results for the read phase for the complete range of ID sizes. As opposed to the distribution of the write phase durations, here we see a very narrow range of values for all ID sizes, where all measurements fall right around the median value. This is expected as the read phase requires that the user's browser traverses the entire redirection chain, which is also apparent by the effect of the ID size on the attack's duration. The minimum time needed to read a 4-bit ID, is $\leq$1 second, and it proportionally grows as the length of the ID increases. Considering again the scenario where the website also leverages the browser fingerprints, the attacker can reconstruct the unique tracking identifier in less than two seconds (median: 1.86 seconds).

Mobile browser. The duration of the two phases when using a mobile device is shown in Figure 6b. As one might expect, there is an increase in the attack's duration for both attack phases and all identifier lengths, due to the reduced computational power of the mobile device. As such, the importance of the optimization techniques is even more necessary for mobile devices – we further explore their effect in the next subsection, and find that for the mobile devices in our dataset at least 18 bits of entropy are always available, which would allow our attack to complete in $\sim$4 seconds.



(a) ID Range : [4K, 65K]

(b) ID Range : [130K, 2M]

(c) ID Range : [4M, 250M]

Fig. 7: Total number of redirections for the two different ID generation algorithms, for the first 250 Million identifiers.

D. Optimization Effect: ID Assignment Algorithm

In §IV-A we presented an alternate ID generation algorithm that creates an "optimized" sequence of identifiers for a given length N, by permuting the order of "1"s in the ID. The goal is to assign better identifiers to users behind resource-constrained devices or slower connections. To quantify its effect, we simulate an execution scenario where new users visit the website and the number of ID bits increases accordingly. To measure the effect of this optimization technique we compare the total number of write bits generated when using the two different identifier-generation algorithms (Standard/Ascending), especially for larger numbers of generated IDs. To better quantify the benefit for weaker devices, all devices are assigned the next available identifier in the sequence (i.e., for the Ascending algorithm we always assign the next available identifier from the top of the sequence). We generate a variety of IDs that range from 12 to 28 bits in length, in order to capture the potential effect of the algorithm(s), under a realistic number of users for both popular and less popular websites.

Figure 7 illustrates the total number of redirection bits for 250 Million IDs for different ID lengths. Even though the number of IDs in each bin remains stable, since it represents the users visiting the websites, we can clearly observe a reduction in the total number of write bits used by the ascending algorithm each time. Specifically, for the first set of IDs (Figure 7a) the measured average decrease is $\sim$16% across different sizes of IDs. Similarly, for the reported IDs in the other ranges shown in Figures 7b, 7c, the total number of redirection bits is reduced by $\sim$15%. Overall, the ascending algorithm optimizes the ID when a new bit (subdomain) is appended to the original schema and more permutations of `1` become available. Compared to the standard approach, this algorithm can considerably improve the attack's write performance for weaker devices.

E. Optimization Effect: Leveraging Browser Fingerprints

Next we explore various aspects of our optimization that relies on the presence of robust browser-fingerprinting attributes. As detailed in §IV-C, retrieving such attributes and computing their entropy allows us to effectively reduce the required length of the favicon-based identifier. Due to the default behavior of certain browsers or the potential presence of anti-tracking tools, the availability of each attribute is not uniform across browsers. Furthermore, different combinations of available browser attributes will result in anonymity crowds of varying sizes. As such we conduct an analysis using a real-world fingerprinting dataset.

We contacted the authors of [50] who provided us with a dataset containing real browser fingerprints collected from `amiunique.org` during March-April 2020. To measure the distribution of the various fingerprinting attributes and their values across different browsers, we filter the dataset and only store instances that have available data for the immutable features in Table IV. In more detail, we reject any entries where all attributes are either empty or obfuscated. We consider as a valid fingerprint any entry that has a stored value for at least one of the attributes. For example, entries that only contain a Platform attribute will be kept, even if the remaining attributes are unavailable or obfuscated. This leaves us with 272,608 (92.7%) entries, which we consider in our subsequent analysis. The removed entries also include platforms that are found infrequently (e.g., Smart TVs, gaming consoles).

Since we do not know a-priori which features are available for each device, we conduct a more in-depth analysis of the dataset and measure the availability of browser fingerprints and the sizes of the anonymity sets that they form. For each device, we read each attribute, and based on the corresponding entropy we sum the entropy values for the available set of immutable features. We find that in this dataset the attributes that were most commonly unavailable due to obfuscation were the WebGL metadata; however, this occurred in $\leq$ 0.05% of the fingerprinting instances indicating that the effect of missing fingerprints would be negligible in practice.

A break down of our results for the various platforms is given in Figure 8. For desktop platforms, the lowest measured entropy from available attributes is 16 bits, revealing that most immutable attributes are always available. Interestingly, for more than half of the devices running Windows we gain 24 bits of entropy, while comparatively Linux and MacOS devices expose 19 bits. These numbers demonstrate that leveraging traditional fingerprinting attributes provides a significant performance optimization as a website would only need between 6-14 favicon-based identifier-bits for $\sim$99.99% of the devices. We can also see that for iOS devices approximately 90% of the devices provide a little over 18-bits of entropy, while Android devices tend to expose attributes with more discriminating power resulting in half the devices having more than 21 bit of entropy. As such, in practice, while the attack's duration can be significantly reduced for all types of mobile devices, Android devices present increased optimization benefits.

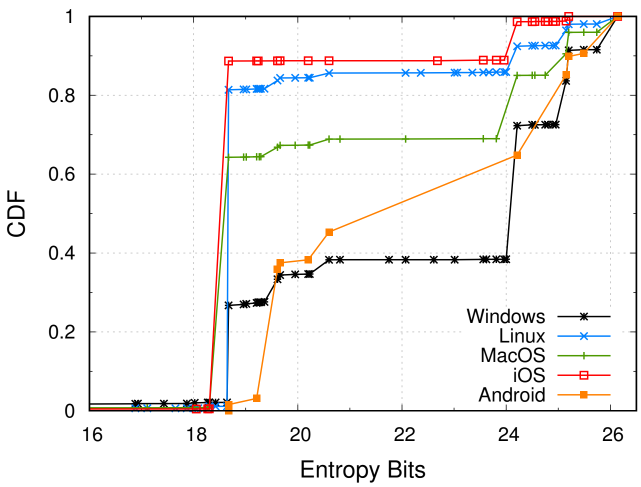


Fig. 8: Fingerprint size for desktop and mobile devices.

TABLE V: Time required for reading each fingerprinting attribute, and amount of time saved due to the reduction in the number of necessary redirections.

	Time Spent (ms)		**Time Saved (ms)**
	mean (μ)	stdev (σ)	
Cookies	2.83	2.86	0
Storage	2.56	2.99	0
DNT	0.32	0.95	110
Ad Blocker	8.21	8.29	110
Platform	0.20	0.08	220
HTTP Metadata	0.42	0.60	770
WebGL Metadata	74.21	13.22	550
Canvas	105.96	11.64	880
All	200.19	20.23	1,760

Attribute effect. Next, we conduct a more in-depth exploration of these benefits by conducting a cost-benefit analysis of each fingerprinting attribute by measuring the time required to obtain it and the corresponding improvement of the provided entropy, shown in Table V. In more detail, attributes that are retrieved through the `Navigator` object (Cookies Enabled, Storage) can be retrieved almost instantaneously, whereas more complex attributes like Canvas and WebGL need at least 100 ms to be processed. The retrieval of each attribute, depending on its internal properties and the gained entropy, decreases the required length of the favicon-based identifier and, thus, the redirection time needed for reading the browser ID. For example, the existence of the DNT attribute provides almost 1 bit of entropy which saves 1 identifier bit (i.e., one redirection) resulting in a 31-bit FV_{ID}. Similarly, the HTTP-metadata provide 7 bits of information, thus needing 7 fewer redirections (840 ms); this would optimize the total attack performance by 22%. Obtaining all the aforementioned attributes from a specific browser instance, requires 200ms. If we add this overhead to the duration of the favicon attack reported in Figure 6a for 12-bit identifiers, we find that in practice our optimized attack requires ∼2 seconds for reconstructing a 32-bit tracking identifier when 20 bits of entropy are available. This can be further optimized using the adaptive threshold.

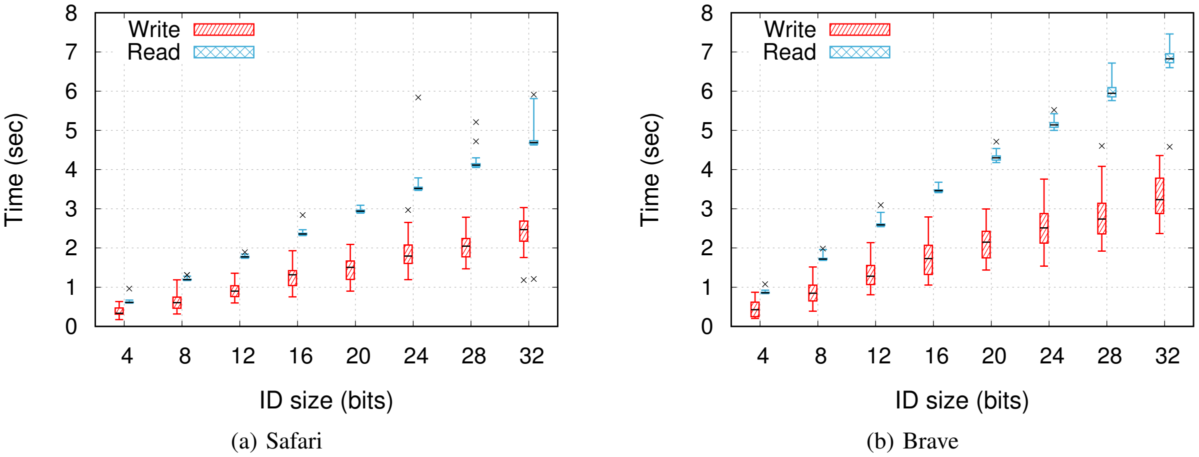
TABLE VI: Popular anti-fingerprinting tools, their defense technique, and the number of entropy bits available from fingerprinting attributes when they are present in the user's browser. Here ⊗ denotes that access to the attributes is blocked by the tool and ⊖ that the attribute values are randomized.

	Users	**Strategy**	**Remaining Entropy (bits)**
CanvasFingerprintBlock [13]	5K	⊗	18
Canvas Fingerprint Defender [87]	10K	⊖	18
Canvas Blocker [42]	9K	⊗	18
WebGL Fingerprint Defender [45]	4K	⊗	21
Brave browser [15]	8M	⊖ ⊗	12

Anti-fingerprinting defenses. Next, we explore the implications of users leveraging anti-fingerprinting defenses. For both the desktop and mobile datasets we observe a high availability of fingerprinting attributes, indicating that such defenses are not commonly deployed. In practice, this could be partially influenced by the nature of the dataset, which originates from users of `amiunique.org`, as users may decide to deactivate any anti-fingerprinting defenses when testing the uniqueness of their system. However, recent work [22] has found that only a small number of users employ such privacy-preserving tools, which may also be ineffective in practice. Specifically, browser extensions that obfuscate and randomize attributes such as the Platform, HTTP headers or session storage, may fail to effectively mask the values. This lack of widespread deployment is also due to the fact that popular anti-tracking tools (e.g., AdBlock, Ghostery, uBlock, Privacy Badger) focus on detecting and blocking 3rd-party domains that are potentially malicious or used by trackers and do not actively defend against fingerprinting; as such we expect a similar availability of fingerprints in practice.

Nonetheless, browsers like Brave have recently adopted built-in anti-fingerprinting techniques which can affect our attack's performance (while Tor has done so for years, we do not consider it in our experiments since it is not susceptible to our favicon attack). In more detail, Brave's documentation [15] reports two different defenses against WebGL and Canvas fingerprinting; the standard defense mode includes the randomization of certain fingerprinting attributes to avoid breaking websites' functionality, while the strict mode blocks these API calls which can potentially break website functionality. In our analysis, we use Brave's strict mode.

To quantify the effect of such privacy-preserving mechanisms on our attack's performance, which would stem from missing fingerprinting attributes, we select the most popular extensions that defend Canvas and WebGL fingerprinting from Google's web store, and the Brave browser. Table VI reports the number of available entropy bits when each tool (or browser) is used. Specifically, we consider that if any tool either randomizes or blocks a specific fingerprinting API the corresponding attributes are unavailable. Interestingly, we observe that none of the anti-fingerprinting extensions affect the immutable attributes that we use for our attack optimization. Out of the 26 bits of entropy that the website could potentially obtain if the entire fingerprinting vector was available, the Canvas-based defenses will end up removing 8 bits. The WebGL-based defense is less effective as 21 bits



(a) Safari

(b) Brave

Fig. 9: Attack performance for alternative browsers.

of entropy will still be available. Brave actually achieves the highest reduction as only 12 bits are left. Nonetheless, even in this case, reading the remaining 20-bits using our favicon-based attack would require ∼3.1 seconds. Overall, while the presence of anti-fingerprinting defenses could result in a less optimized (i.e., slower) performance, our attack's duration remains acceptable.

It is also important to note that while blocking a specific fingerprinting call may be considered a stronger defense, in this case it works in the favor of the attacker since they can easily ignore that specific attribute. On the other hand, using a randomized value will result in the website calculating different identifiers across visits. As such, websites can leverage extension-fingerprinting techniques [71], [44], [77], [74] to infer the presence of these extensions and ignore the affected attributes when generating the FP_{ID}. For Brave, websites simply need to check the User-agent header.

F. Evaluating Browser Performance

As shown in Table II, several browsers across different operating systems are vulnerable to our attack. To explore whether different browsers result in different attack durations, we repeat our experiment with two additional browsers and the user connected to a residential network, as illustrated in Figure 9. Specifically, we evaluate Safari as it is a popular choice for MacOS users, and Brave as it is Chromium-based and privacy-oriented. Surprisingly, while Brave's writing performance is comparable to that of Chrome (Figure 6a), there is a measurable increase when reading the identifier (the median attack for a 32-bit ID is 1.35 seconds slower than Chrome). For Safari we observe that the attack's overall performance is similar to Chrome and even slightly better for some ID sizes. Our experiments show that differences in browser internals can affect the performance of straightforward operations like caching and reading favicons, even when powered by the same engine (as is the case with Brave). As such, the benefit of our fingerprint-based optimization will be even more pronounced for Brave users.

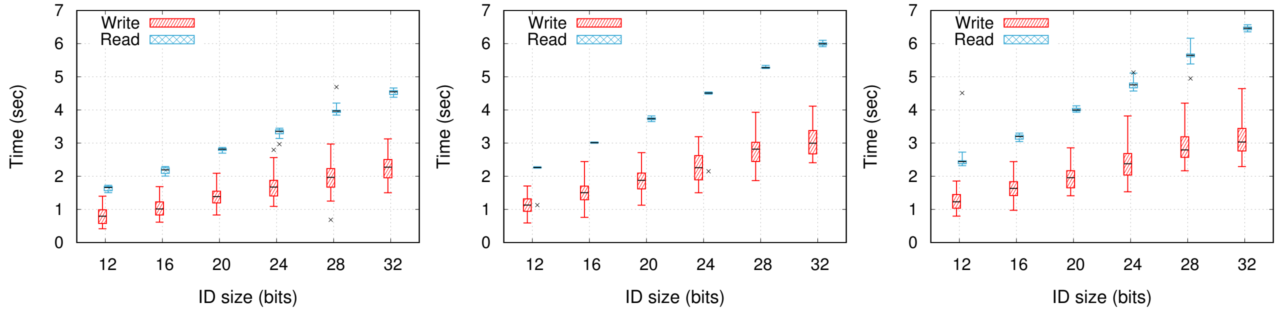
G. Evaluating Network Effects

To measure the effect that different network and infrastructure conditions can have on the attack's performance, we conduct a series of experiments that explore alternative server and client network setups.

Server location. First, we aim to measure how our attack's performance is affected for different web server locations. For this set of experiments, we use our vanilla attack with a consistent redirection threshold value of 110ms. We then compare the attack's duration for a selection of identifier sizes, for three different locations, as shown in Figure 10. *Same_City* captures the scenario where the victim and web server are located within the same city; since AWS does not offer any hosting options in our area, we host the server in our academic institution's computing infrastructure. The *State_A* scenario uses a server hosted on Amazon AWS in a different geographic region (distance $\sim$ 850 miles), while the *State_B* experiment uses an AWS server located in a distant region (distance $\sim 2,000$ miles).

As one might expect, we find that the attack requires less time to complete when the server and client are located in the same city. Specifically, for a 32-bit ID size the median value is ∼27% faster for writing the identifier compared to the other locations, while the reading time is decreased by 35% compared to the distant server *State_B*. This experiment verifies that there is a noticeable performance impact for distant locations, however the attack maintains a practical duration in all scenarios.

To further measure the effect of the server's location on the performance, we repeat our *threshold selection* experiment using the server deployed in our academic institution and the desktop client connecting from a residential network in the same city. Under these conditions, we find that a redirection threshold of 70 ms is sufficient for the browser to successfully request and store the favicons, which significantly reduces the attack's overall duration; e.g., for a 32-bit identifier the median read and write values are 1.5 and 3.14 seconds respectively. Overall, our experiments demonstrate that attackers with access to a distributed infrastructure resources (which is a



(a) Same_City (b) State_A (c) State_B

Fig. 10: Attack performance evaluation for servers located in different regions.

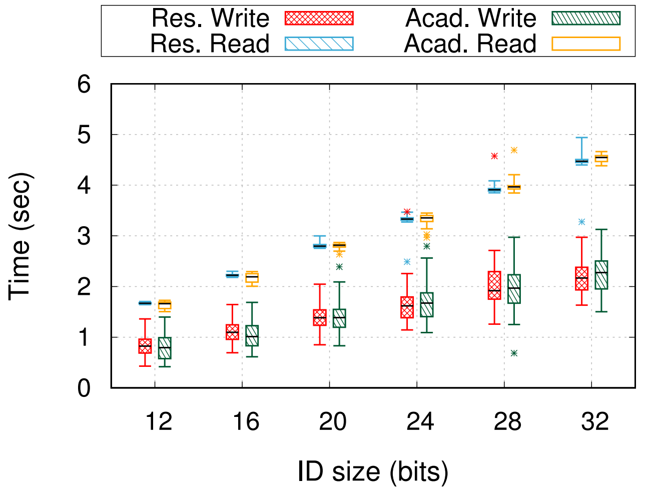


Fig. 11: Performance evaluation for clients connecting from two different networks: a high speed network in our academic institution (*Acad.*) and a residential network (*Res.*).

reasonable assumption for modern attackers) can considerably reduce the attack's duration by using CDNs and dedicated machines across different locations.

Client network. We explore how the attack's performance changes depending on the type of the user's network. To that end, we use the server deployed in our academic institution (to reduce the effect of the server's location) and test two different client network setups. In the first case, we explore a more ideal scenario where the user is connected to the same (academic) network, while the second setup showcases a scenario where the user is connected to a different (residential) network. As shown in Figure 11, the performance is consistent across networks for approximately half of the attack runs. For smaller identifier sizes there is no discernible difference during the writing phase, while there is a small improvement in the reading phase for approximately 25% of the attacks when the client is on the academic network. Additionally, when the user is on the residential network approximately 25% of the runs exhibit a small increase in the attack's duration. For larger identifiers we see a higher variance in the write phase for the user on the academic network, while we again observe that the reading phase exhibits less variance when the user is on the academic network. Overall, we do not observe a considerable difference in the attack's performance even when the client is on a residential network, further demonstrating the robustness of using favicons as a tracking vector in real-world scenarios.

VI. Mitigations and Countermeasures

Here we propose mechanisms for mitigating the tracking vector enabled by favicon caching. We outline necessary browser modifications and potential countermeasures, and discuss the limitations they face in practice. Due to the nature of the underlying browser functionality leveraged by our attack, there is no straightforward countermeasure that can prevent the attack without affecting the user's browsing experience.

Incognito mode. Browsers currently allow the favicon cache to be read even when the user is browsing in incognito mode. While this is allowed for performance reasons, it is a design flaw in the browsers' isolation mechanism that should be addressed. Similar to other cached content, a separate isolated instance of the cache should be created for each browsing session. Even if this will introduce a small additional delay for new favicons that need to be fetched, we consider the overhead reasonable as the underlying browser functionality is not affected and there is an obvious privacy gain for users.

Cookie-tied favicon caching. The main defense against our attack is to remove the suitability of the favicon cache as a tracking vector. Specifically, by tying the use of cached favicons to the presence of a first-party cookie, the browser can basically invalidate any additional benefit of using the favicon cache to track the user; if cookies are already present then the website can obviously match the user to previous browsing sessions. A straightforward way to implement this defense is to simply clear the favicon cache whenever the user deletes the cookie jar and other local storages and caches (e.g., through the "Clear browsing data" option in Chrome). The downside of this countermeasure is the potential performance penalty; if F-Cache entries are deleted frequently or after every browser session, favicons will need to be re-fetched every time users revisit each website. Nonetheless, this overhead is not prohibitive even for cellular networks, since fetching a favicon is an asynchronous and non-blocking operation.

Navigation-based favicon caching. Browsers can potentially employ an alternative strategy for preventing our attack, where the caching of favicons is managed based on the navigation's transition type [3]. Specifically, if a navigation occurs to a different subpath or subdomain for which a F-Cache entry does not exist, the browser will fetch the favicon and create the entry *only* if the user initiated the navigation. While this strategy does not introduce the (negligible) performance overhead of the previous caching strategy, it could potentially be bypassed if the website slowly recreates the identifier throughout the user's browsing session where each click on a link is used to obtain one identifier bit. Naturally, such an attack strategy would face the risk of incomplete identifier reconstruction in short user sessions, and would be more suitable for certain categories of websites (e.g., e-commerce). We further discuss the attack strategy of stealthily reconstructing the identifier in §VII.

VII. Discussion

Attack detection. URL redirections have been proposed in various prior studies as a signal for detecting malicious activities (e.g., malware propagation [58], SEO poisoning [54], click jacking [88]). However, in such cases the redirection typically involves redirection to *different* domains or hosts, which does not occur in our attack. Nonetheless, one could potentially deploy a detection mechanism that also checks intra-domain redirections. In such a case, a website could potentially opt for a stealthier strategy where the chain of redirections is completed in phases over the duration of a user's browsing session. Based on statistics regarding average user behavior within a given domain (e.g., session duration, number of links clicked) a website could optimize this process by creating partial redirection chains that are completed each time a user clicks on a link or navigates within the website. Especially for websites like social networks, search engines, e-commerce and news websites, where common browsing activity involves clicking on numerous links and visiting many different pages, the website could trivially include one or two additional redirections per click and avoid any redirection-based detection. When taking into consideration our optimization strategies, such a website could trivially reconstruct the 12-bit favicon-based identifier without a considerable impact on the attack's coverage (i.e., aggregate user stats would allow the website to fine-tune a stealthy attack so only a very small percentage of users terminates a browsing session without completing the entire redirection chain).

Tracking and redirections in the wild. A recent study on the page latency introduced by third-party trackers for websites that are popular in the US [36] reported that only 17% of pages load within 5 seconds while nearly 60% and 18% of pages require more than 10 and 30 seconds respectively. Their analysis also highlighted the dominating effect that trackers have on the overall page-loading latency, with an average increase of 10 seconds. When taking their reported numbers into consideration, it becomes apparent that the cost of our attack is practical for real-world deployment. Furthermore, Koop et al. [47] just recently studied how redirections to third-party trackers are commonly employed as a means for them to "drop" tracking cookies in users' browsers. As such, the underlying mechanism that drives our attack, resembles behavior that is already employed by websites, reducing the likelihood of our attack being perceived as abnormal behavior.

Deception and enhancing stealthiness. While redirections are already part of "normal" website behavior and, thus, may not be perceived as concerning or malicious by average users, several deceptive strategies can be employed to further enhance the stealthiness of our attack. For instance, websites can employ various mechanisms for distracting the user (e.g., a popup about GDPR compliance and cookie-related policies [81]). Additionally, JavaScript allows for animations and emoticons to be encoded in the URL [67]. An attacker could use such animated URL transitions to obscure the redirections. Finally, Chrome is currently experimenting with hiding the full URL in the address bar and only showing the domain [4] as a way to combat phishing attacks [68]. If Chrome or other browsers permanently adopt this feature where only the main domain is shown by default, our attack will be completely invisible to users as it leverages redirections within the same domain.

Anti-fingerprinting. Our work presents an attack which, conceptually, is a member of the side-channel class of attacks. One important implication of our work, with respect to browser fingerprinting and online tracking, is that such an attack composes well with any number of entropy bits available from traditional browser fingerprinting; while browsers are working to decrease the aggregate entropy down to prevent unique device identification, e.g., Brave [15], the remaining bits are still incredibly powerful when composed with a side-channel tracking technique.

In more detail, while even the vanilla version of our attack is well within the range of overhead introduced by trackers in the wild [36], leveraging immutable browser-fingerprinting attributes significantly reduces the duration of the attack. As such, while browser fingerprints typically do not possess sufficient discriminating power to uniquely identify a device, they introduce a powerful augmentation factor for any high-latency or low-bandwidth tracking vector. Furthermore, while our attack remains feasible even without them, other tracking techniques may only be feasible in conjunction with browser fingerprints. As such, we argue that to prevent the privacy impact of as-yet-undiscovered side-channel attacks and tracking vectors, anti-fingerprinting extensions and browser vendors should expand their defenses to include *all* the immutable fingerprinting attributes we leverage in our work instead of focusing on a single (or small set) of attributes.

Favicon caching and performance. Recently certain browsers have started supporting the use of `Data URI`s for favicons. Even though this technique can effectively serve and cache the favicon in the user's browser almost instantaneously, it cannot currently be used to optimize our attack's performance. In more detail, the write phase does not work since the browser creates different cache entries for the base-64 representations of the favicons, and stores a different sequence of icons than those served by the page. Moreover, since there are not requests issued by the browser for such resources, reading those cached-favicons would not be possible. Finally, we also experimented with the HTTP2 Server Push mechanism, but did not detect any performance benefit.

Ethics and disclosure. First we note that all of our experiments were conducted using our own devices and no users were actually affected by our experiments. Furthermore, due to the severe privacy implications of our findings we have disclosed our research to all the browser vendors. We submitted detailed reports outlining our techniques, and vendors have confirmed the attack and are currently working on potential mitigations. In fact, among other mitigation efforts, Brave's team initially proposed an approach of deleting the Favicon-Cache in every typical "Clear History" user action, which matches our "Cookie-tied favicon caching" (see §VI) mitigation strategy that can work for all the browsers. The countermeasure that was eventually deployed adopts this approach while also avoiding the use of favicon cache entries when in incognito mode. Additionally, the Chrome team has verified the vulnerability and is still working on redesigning this feature, as is the case with Safari. On the other hand, the Edge team stated that they consider this to be a non-Microsoft issue as it stems from the underlying Chromium engine.

VIII. Related Work

Online Tracking. Numerous studies have focused on the threat of online tracking and the techniques that are employed for tracking and correlating users' activities across different websites. These can be broken down into stateful [69], [64], [25], [86] and stateless tracking techniques [23], [10], [9], [63], [62], [65], [73]. One of the first studies about tracking [57] measured the type of information that is collected by third parties and how users can be identified. Roesner et al. [69] analyzed the prevalence of trackers and different tracking behaviors in the web, while Lerner et al. [52] provided a longitudinal exploration of tracking techniques. Olejnik et al. [64] investigated "cookie syncing", a technique that provides third parties with a more complete view of users' browsing history by synchronizing their cookies. Englehardt and Narayanan [25] conducted a large scale measurement study to quantify the use of stateful and stateless tracking and cookie syncing. Numerous studies have also proposed techniques for blocking trackers [39], [35], [38], [86]. On the other hand, out paper demonstrates a novel technique that allows websites to re-identify users. Conceptually, our work is closer to "evercookies" – Acar et al. [9] investigated their prevalence and the effects of cookie re-spawning in combination with cookie syncing. The HSTS mechanism has also been abused to create a tracking identifier [30]. Klein and Pinkas [46] recently demonstrated a novel technique that tracks users by creating a unique set of DNS records, with similar tracking benefits to ours, which also works across browsers on the same machine (our technique is bound to a single browser). However, their attack is not long-term due to the limited lifetime of caching of DNS records at stub resolvers (between a few hours and a week) whereas favicons can be cached for an entire year.

Browser fingerprinting. While stateful techniques allow websites to uniquely identify users visiting their site, they are typically easier to sidestep by clearing the browser's state. This has led to the emergence of stateless approaches that leverage browser fingerprinting techniques [32], [50], [23], [60]. A detailed survey on techniques and behaviors can be found in [49]. Nikiforakis et al. [63] investigated various fingerprinting techniques employed by popular trackers and measured their adoption across the web. Acar et al. [10] proposed `FPDetective`, a framework that detects fingerprinting by identifying and analyzing specific events such as the loading of fonts, or accessing specific browser properties. Also, Cao et al. [17] proposed a fingerprinting technique that utilizes OS and hardware level features to enable user tracking across different browsers on the same machine. Recently, Vastel et.al. [82], designed `FP-STALKER`, a system the monitors the evolution of browser fingerprints across time, and found that the evolution of fingerprints strongly depends on the device's type and utilization. Other defenses also include randomization techniques and non-deterministic fingerprints [62], [48].

Cache-based attacks. Prior studies have extensively explored security and privacy issues that arise due to browser's caching policies of different resources [70], [34], [80], often with a focus on history-sniffing [75], [51], [14]. Nguyen et al. [61] conducted an extensive survey of browser caching behavior by building a novel cache testing tool. Bansal et al. [14] extended history sniffing attacks using web workers and cache-timing attacks. In a similar direction, Jia et al. [40] exploited browsers' caches to infer the geo-location information stored in users' browsing history. While our attack similarly leverages browsers' caching behavior, we find that the favicon cache exhibits two unique characteristics that increase the severity and impact of our attack. First, this cache is not affected by user actions that clear other caches, local storages and browsing data, enabling the long-term tracking of users. Next, while browsers fully isolate other local storages and caches from the incognito mode that is not the case for the favicon cache, allowing our attack to track users in incognito mode.

Favicons have not received much scrutiny from the research community. In one of the first studies, Geng et al. [31] used favicons to successfully differentiate between malicious and benign websites. Their method had high accuracy, and this work was the first that evaluated and characterized favicon usage in the wild. Chiew et al. [19] also proposed the use of favicons for the detection of phishing pages. Finally, favicons have been used as part of other types of attacks, such as man-in-the-middle attacks [56], inferring whether a user is logged into certain websites [2], distributing malware [1] or stealthily sharing botnet command-and-control addresses [66].

IX. Conclusion

As browsers increasingly deploy more effective anti-tracking defenses and anti-fingerprinting mechanisms gain more traction, tracking practices will continue to evolve and leverage alternate browser features. This necessitates a proactive exploration of the privacy risks introduced by emerging or overlooked browser features so that new tracking vectors are identified before being used in the wild. In this paper we highlighted such a scenario by demonstrating how favicons, a simple yet ubiquitous web resource, can be misused as a powerful tracking vector due to the unique and idiosyncratic favicon-caching behavior found in all major browsers. In fact, cached favicons enable long-term, persistent user tracking that bypasses the isolation defenses of the incognito mode and is not affected by existing anti-tracking defenses. Furthermore, we analyzed a real-world dataset and illustrated how immutable browser fingerprints are ideal for optimizing low-bandwidth tracking mechanisms. When leveraging such fingerprints our attack can reconstruct a unique 32-bit tracking identifier in 2 seconds, which is significantly less than the average 10-second overhead introduced by trackers on popular websites [36]. To address the threat posed by our technique, we disclosed our findings to browser vendors and remediation efforts are currently underway, while we also outlined a series of browser changes that can mitigate our attack.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (CNS-1934597). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the National Science Foundation.

References

[1] "Getastra - favicon (.ico) virus backdoor in wordpress, drupal," https://www.getastra.com/e/malware/infections/favicon-ico-malware-backdoor-in-wordpress-drupal.

[2] "Robin linus - your social media footprint," https://robinlinus.github.io/socialmedia-leak/.

[3] "Chrome developers: Transition types," https://developers.chrome.com/extensions/history#transition_types, 2019.

[4] "Chromium blog - helping people spot the spoofs: a url experiment," https://blog.chromium.org/2020/08/helping-people-spot-spoofs-url.html, 2020.

[5] "Git repositories on chromium," https://chromium.googlesource.com/chromium/chromium/+/4e693dd4033eb7b76787d3d389ceed3531c584b5/chrome/browser/historyhistory_backend.cc, 2020.

[6] "Mdn web docs - network information api," https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API, 2020.

[7] "Nginx plus," https://nginx.org/en/, 2020.

[8] Wikipedia, "Favicon," https://en.wikipedia.org/wiki/Favicon, 2009.

[9] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 674–689.

[10] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "Fpdetective: Dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1129–1140. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516674

[11] Amazon, "Amazon lightsail virtual servers, storage, databases, and networking for a low, predictable price." https://aws.amazon.com/lightsail/, 2020.

[12] Amazon, "The top 500 sites on the web," https://www.alexa.com/topsites, 2020.

[13] appodrome.net, "Canvasfingerprintblock," https://chrome.google.com/webstore/detail/canvasfingerprintblock/ipmjngkmngdcdpmgmiebdmfbkcecdndc, 2020.

[14] C. Bansal, S. Preibusch, and N. Milic-Frayling, "Cache Timing Attacks Revisited: Efficient and Repeatable Browser History, OS and Network Sniffing," in *IFIP International Information Security and Privacy Conference*. Springer, May 2015, pp. 97–111.

[15] Brave, "Fingerprinting protections," https://brave.com/whats-brave-done-for-my-privacy-lately-episode-4-fingerprinting//-defenses-2-0/, 2020.

[16] T. Bujlow, V. Carela-Español, J. Sole-Pareta, and P. Barlet-Ros, "A survey on web tracking: Mechanisms, implications, and defenses," *Proceedings of the IEEE*, vol. 105, no. 8, pp. 1476–1510, 2017.

[17] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via os and hardware level features," in *Proceedings of Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2017.

[18] Y. Cao, S. Li, E. Wijmans *et al.*, "(cross-) browser fingerprinting via os and hardware level features." in *NDSS*, 2017.

[19] K. L. Chiew, J. S.-F. Choo, S. N. Sze, and K. S. Yong, "Leverage website favicon to detect phishing websites," *Security and Communication Networks*, vol. 2018, 2018.

[20] F. T. Commission, "Consumer information - online tracking," https://www.consumer.ftc.gov/articles/0042-online-tracking.

[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 ACM CCS, October 2018*, 2018.

[22] A. Datta, J. Lu, and M. C. Tschantz, "Evaluating anti-fingerprinting privacy enhancing technologies," in *The World Wide Web Conference*, 2019, pp. 351–362.

[23] P. Eckersley, "How unique is your web browser?" in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, ser. PETS'10, 2010, pp. 1–18.

[24] S. Englehardt *et al.*, "Automated discovery of privacy violations on the web," 2018.

[25] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 1388–1401.

[26] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc2616: Hypertext transfer protocol–http/1.1," 1999.

[27] Firefox, "Resources for developers, by developers," https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/mozIAsyncFavicons, 2020.

[28] J. Foundation, "Automation for apps," http://appium.io/, 2020.

[29] 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.

[30] B. Fulgham, "Webkit - protecting against hsts abuse," https://webkit.org/blog/8146/protecting-against-hsts-abuse/, 2018.

[31] G.-G. Geng, X.-D. Lee, W. Wang, and S.-S. Tseng, "Favicon - a Clue to Phishing Sites Detection," in *Proceedings of the 2013 APWG eCrime Researchers Summit*. IEEE, September 2013, pp. 1–10.

[32] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *Proceedings of the 2018 world wide web conference*, 2018, pp. 309–318.

[33] Google, "Reset chrome settings to default," https://support.google.com/chrome/answer/3296214?hl=en, 2020.

[34] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page cache attacks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 167–180.

[35] D. Gugelmann, M. Happe, B. Ager, and V. Lenders, "An automated approach for complementing ad blockers blacklists," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 282–298, 2015.

[36] M. Hanson, P. Lawler, and S. Macbeth, "The tracker tax: the impact of third-party trackers on website speed in the united states," Technical report, 2018. Available at: https://www. ghostery. com/wp-content , Tech. Rep., 2018.

[37] J. Hoffman, "How we got the favicon," https://thehistoryoftheweb.com/how-we-got-the-favicon/.

[38] M. Ikram, H. J. Asghar, M. A. Kaafar, A. Mahanti, and B. Krishnamurthy, "Towards seamless tracking-free web: Improved detection of trackers via one-class learning," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 1, pp. 79–99, 2017.

[39] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *In Proceedings of the 37th IEEE Symposium on Security and Privacy*, ser. S&P '20, 2020.

[40] Y. Jia, X. Dong, Z. Liang, and P. Saxena, "I know where you've been: Geo-inference attacks via the browser cache," *IEEE Internet Computing*, vol. 19, no. 1, pp. 44–53, 2014.

[41] Johannes Buchner, "An image hashing library written in python," https://pypi.org/project/ImageHash/, 2020.

[42] joue.quroi, "Canvas blocker (fingerprint protect)," https://chrome.google.com/webstore/detail/canvas-blocker-fingerprin/nomnklagbgmgghhjidfhnoelnjfndfpd, 2020.

[43] S. Karami, P. Ilia, and J. Polakis, "Awakening the Web's Sleeper

Agents: Misusing Service Workers for Privacy Leakage," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[44] 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)*. The Internet Society, 2020.

[45] Keller, "Webgl fingerprint defender," https://chrome.google.com/webstore/detail/webgl-fingerprint-defende/olnbjpaejebpnokblkepbphhembdicik, 2020.

[46] A. Klein and B. Pinkas, "Dns cache-based user tracking." in *NDSS*, 2019.

[47] M. Koop, E. Tews, and S. Katzenbeisser, "In-depth evaluation of redirect tracking and link usage," *Proceedings on Privacy Enhancing Technologies*, vol. 4, pp. 394–413, 2020.

[48] P. Laperdrix, B. Baudry, and V. Mishra, "Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 97–114.

[49] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020.

[50] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 878–894.

[51] S. Lee, H. Kim, and J. Kim, "Identifying cross-origin resource status using application cache." in *NDSS*, 2015.

[52] 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.

[53] 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 (CCS)*, 2020, pp. 507–519.

[54] L. Lu, R. Perdisci, and W. Lee, "Surf: detecting and measuring search poisoning," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 467–476.

[55] 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 *30th International World Wide Web Conference, WWW '19*. ACM, 2019.

[56] M. Marlinspike, "More tricks for defeating ssl in practice," *Black Hat USA*, 2009.

[57] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 413–427. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.47

[58] H. Mekky, R. Torres, Z.-L. Zhang, S. Saha, and A. Nucci, "Detecting malicious http redirections using trees of user browsing activity," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1159–1167.

[59] Microsoft, "How to Add a Shortcut Icon to a Web Page," https://technet.microsoft.com/en-us/windows/ms537656(v=vs.60).

[60] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, pp. 1–12, 2012.

[61] H. V. Nguyen, L. Lo Iacono, and H. Federrath, "Systematic Analysis of Web Browser Caches," in *Proceedings of the 2nd International Conference on Web Studies*. ACM, October 2018, pp. 64–71.

[62] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving fingerprinters with little white lies," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015, pp. 820–830. [Online]. Available: https://doi.org/10.1145/2736277.2741090

[63] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 541–555. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.43

[64] L. Olejnik, T. Minh-Dung, and C. Castelluccia, "Selling off privacy at auction," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[65] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at internet scale." in *NDSS*, 2016.

[66] T. Pevnỳ, M. Kopp, J. Křoustek, and A. D. Ker, "Malicons: Detecting payload in favicons," *Electronic Imaging*, vol. 2016, no. 8, pp. 1–9, 2016.

[67] M. Rayfield, "Animating urls with javascript and emojis," https://matthewrayfield.com/articles/animating-urls-with-javascript-and-emojis/, 2019.

[68] J. Reynolds, D. Kumar, Z. Ma, R. Subramanian, M. Wu, M. Shelton, J. Mason, E. M. Stark, and M. Bailey, "Measuring identity confusion with uniform resource locators," 2020.

[69] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228315

[70] 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*, 2019, pp. 478–488.

[71] I. Sanchez-Rola, I. Santos, and D. Balzarotti, "Clock around the clock: time-based device fingerprinting," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1502–1514.

[72] Selenium, "Selenium automates browsers," https://www.selenium.dev.

[73] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 639–656.

[74] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 329–336.

[75] 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

[76] 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.

[77] O. Starov and N. Nikiforakis, "Xhound: Quantifying the fingerprintability of browser extensions," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 941–956.

[78] P. Syverson and M. Traudt, "Hsts supports targeted surveillance," in *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI '18)*, 2018.

[79] F. D. Team, "Flask," https://palletsprojects.com/p/flask/, 2020.

[80] T. Tiwari and A. Trachtenberg, "Alternative (ab) uses for http alternative services," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[81] C. Utz, M. Degeling, S. Fahl, F. Schaub, and T. Holz, "(un) informed consent: Studying gdpr consent notices in the field," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 973–990.

[82] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-stalker: Tracking browser fingerprint evolutions," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 728–741.

[83] W3C, "How to add a favicon to your site." https://www.w3.org/2005/10/howto-favicon, 2020.

[84] W3Schools, “Html link rel attribute,” https://www.w3schools.com/tags/att_link_rel.asp, 2020.

[85] Y. Wu, P. Gupta, M. Wei, Y. Acar, S. Fahl, and B. Ur, “Your secrets are safe: How browsers’ explanations impact misconceptions about private browsing mode,” in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 217–226.

[86] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, “Tracking the trackers,” in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW ’16. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 121–132. [Online]. Available: https://doi.org/10.1145/2872427.2883028

[87] Yubi, “Canvas fingerprint defender,” https://chrome.google.com/webstore/detail/canvas-fingerprint-defend/lanfdkkpgfjfdikkncbnojekcppdebfp, 2020.

[88] M. Zhang, W. Meng, S. Lee, B. Lee, and X. Xing, “All your clicks belong to me: investigating click interception on the web,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 941–957.

Appendix

Here we provide additional details and context to clarify our disclosure process.

Initially this project started by trying to understand how favicons work in browsers, including how they are fetched, their characteristics, how they are stored etc. One of our earliest discoveries was that in the case of Firefox even basic favicon functionality didn’t seem to work correctly and thats why we submitted a bug report to Mozilla in February 2020. As such, we focused our exploration on other browsers where basic favicon functionality appeared to be working correctly. As we continued our exploration, and after other ideas didn’t pan out, we devised the tracking technique detailed in the paper. This was a fairly straightforward technique to test and we ran experiments to see which browsers were susceptible. We informed all vulnerable browsers through reports in June 2020, detailing our technique and the underlying issue – as we wrote in the paper “we have disclosed our findings to all affected browsers”. This disclosure took place even before we submitted our paper to NDSS in July 2020. We opted for a pre-submission disclosure to give vendors ample time to address the issues we had uncovered. We made our findings public many months later (February 2021) after the camera ready version of our paper was accepted by NDSS. Not notifying Firefox at that time was definitely an oversight, which was a result of us focusing on which browsers were vulnerable and notifying those. This wasn’t done to harm users in any way, nor was it done to improve the paper’s impact as claimed by certain people online. This is further corroborated by two facts:

- We already explicitly mentioned in the paper that our attack didn’t work against at least one browser (Tor).
- We didn’t notify Firefox even after our paper was accepted, at which point whether Firefox was vulnerable or not would obviously not improve the paper or change anything for us. Again, this shows that we simply (and unwisely) didn’t take Firefox into account during the disclosure process because it wasn’t susceptible to the attack. It wasn’t due to some elaborate or malicious plan.

We agree that we should have proactively notified Firefox to make sure that the attack wouldn’t work no matter what changes they eventually made to their favicon functionality. In hindsight, this seems like a painfully obvious thing to do, but this past year has been far from normal. We hope that the conversations we had with the vulnerable vendors to get this security bug fixed shows that we have the best interests of web users at heart. This can also be seen by our track record of responsibly disclosing our research, including two other disclosures that Firefox was a part of in the past year alone. Without an understanding of the larger context around our work, its definitely understandable that people would get the impression that we wanted Mozilla to introduce a weakness in their browser, but we trust that members of our research community will be able to better evaluate the situation. Neglecting to circle back with Mozilla about the security issue that could potentially be introduced in their browser was an oversight, and thankfully the overarching favicon caching flaw is now well known and any fix to Firefox will include appropriate proactive mitigations further ensuring that no users will be affected by this vulnerability.