Navigating Murky Waters: Automated Browser Feature Testing for Uncovering Tracking Vectors

Anonymous Authors

***Abstract*—Modern web browsers constitute complex application platforms with a wide range of APIs and features. Critically, this includes a multitude of heterogeneous mechanisms that allow sites to store information that explicitly or implicitly alters client-side state or functionality. This behavior implicates any browser *storage*, *cache*, *access control*, and *policy* mechanism as a potential tracking vector. As demonstrated by prior work, tracking vectors can manifest through elaborate behaviors and exhibit varying characteristics that differ vastly across different browsing contexts. In this paper we develop CanITrack, an automated, mechanism-agnostic framework for testing browser features and uncovering novel tracking vectors. Our system is designed for facilitating browser vendors and researchers by streamlining the systematic testing of browser mechanisms. It accepts methods to read and write entries for a mechanism and calls these methods across different browsing contexts to determine any potential tracking vulnerabilities that the mechanism may expose. To demonstrate our system's capabilities we test 21 browser mechanisms and uncover a slew of tracking vectors, including 13 that enable third-party tracking and two that bypass the isolation offered by private browsing modes. Importantly, we show how two separate mechanisms from Google's highly-publicized and widely-discussed Privacy Sandbox initiative can be leveraged for tracking. Our experimental findings have resulted in 20 disclosure reports across seven major browsers, which have set remediation efforts in motion. Overall, our study highlights the complex and formidable challenge that browsers currently face when trying to balance the adoption of new features and protecting the privacy of their users, as well as the potential benefit of incorporating CanITrack into their internal testing pipeline.**

I. Introduction

The privacy of online activities is a growing concern to an increasing number of users [9], with a recent survey finding that 80% of users are worried about online tracking [11]. In recent years trackers have pivoted away from cookies, to a variety of alternative techniques as a response to cookie-based tracking countermeasures. In turn, this attracted increased scrutiny from the security community towards identifying new tracking vectors. At the same time, browsers have continued to evolve as complex application platforms by deploying new features and APIs that further complicate efforts towards restricting online tracking.

Even though new browser features offer novel functionality, they also increase the browser's attack surface, introducing new flaws and opportunities for misuse. One particularly problematic class of flaws involves mechanisms that can be abused for re-identifying and tracking users. Researchers have already demonstrated novel tracking techniques that leverage browser mechanisms which at face value do not resemble tracking mechanisms, such as, HSTS policies [70] and favicon caches [67]. These studies have showed that *any* mechanism that stores some form of data in the browser or affects client-side policies is a *potential* tracking vector.

This observation has serious implications for browser vendors' internal testing procedures, which may not include testing workflows for assessing this specific privacy risk. We argue that any client-side *caching*, *storage*, *access-control*, or *policy* mechanism should be thoroughly evaluated as a potential tracking vector prior to its public release. Moreover, once those mechanisms are actually deployed, security researchers may employ manual or ad hoc approaches that do not comprehensively test all pertinent aspects of a potential tracking vector's capabilities. As manual testing cannot scale, such endeavors will be limited to a small number of browser versions, thus being unable to uncover longitudinal patterns of vulnerability evolution over time [53, 58, 59, 64].

In this paper we present CanITrack, an automated framework designed to streamline the testing of browser features and assessing whether they can be misused as a tracking vector. The modular design of CanITrack provides all the necessary components for orchestrating browsers and web servers, and exploring multiple dimensions of tracking functionality; this includes the effects of first-party (1P) and third-party (3P) navigation, testing the isolation offered by incognito mode, and the impact of browsing data being cleared. To achieve our design goal of a *mechanism-agnostic* framework, CanITrack interacts with basic `write()` and `read()` user-provided functions, for the mechanism being tested. These functions implement a mechanism-specific action that allows our system to write and read, respectively, one or more bits of information that are used for storing and reconstructing a tracking identifier. For instance, if the developer wishes to test the suitability of using the favicon cache as a tracking vector [67], the `write()` function would simply require requesting a unique favicon, while the `read()` function would infer the presence of that favicon based on whether it is fetched over the network or returned from the internal cache. CanITrack then invokes the user-provided methods to assess how access to the mechanism is limited in multiple first-party (1P) and third-party (3P) browsing contexts, within the top-level context and from any embedded iframes. It also determines if a partitioning key is associated with the mechanism and, if so, infers the key's composition. Our system additionally develops redirection chains and evaluates methods for extending access to the mechanism from a single page, and also orchestrates a web server that can be used to host resources on multiple paths, ports, and domains. CanITrack manages the experimental pipeline

Network and Distributed System Security (NDSS) Symposium 2023
28 February - 4 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23xxx
www.ndss-symposium.org

using fresh browser instances for each experiment, which are controlled by simulating user interactions, and ensures that its results are consistent with end-user experience.

To experimentally evaluate CanITrack, we implement the necessary `write()` and `read()` functions for 21 browser mechanisms, including four mechanisms for which we provide the first exploration of their utility as a tracking vector. We then conduct a comprehensive evaluation of these mechanisms across a total of 126 versions of seven major browsers (i.e., Brave, Chrome, Edge, Firefox, Safari, Opera, and Tor) across a two-year period and find that all are vulnerable to *at least* one new tracking technique in their latest version. Crucially, we demonstrate how two mechanisms from Google's widely-discussed Privacy Sandbox initiative can be used for third-party tracking. Surprisingly, CanITrack also revealed a *new* behavior of the favicon cache in the latest versions of Chrome and Safari, which we leverage for demonstrating a novel history-sniffing attack. Overall, our experiments highlight that our framework streamlines the *comprehensive* and *systematic* testing of browser features as potential tracking vectors while requiring minimal effort from intended users.

In summary, we make the following research contributions:

- We develop CanITrack, a novel framework that streamlines the comprehensive testing of browser features as potential tracking vectors.
- We experimentally evaluate our system on 21 browser features, including four previously-untested mechanisms (two are from Google's Privacy Sandbox), and demonstrate their suitability as tracking vectors across different deployment scenarios and different browser versions.
- Due to the severe privacy implications of our research, we have disclosed our findings to all affected browsers to enable remediation.
- To ensure reproducibility, we will be open sourcing CanITrack upon publication of this paper.

II. System Design and Implementation

In this section, we first briefly outline the tracking behaviors and threat model that have guided the design of CanITrack. We then detail our framework's design and implementation.

First-party (1P) and Third-party (3P) Context. We consider the domain that a user visits to be the first-party (1P) entity, and any access to a browser mechanism made by the 1P entity or its subdomains as 1P access. On the other hand, we refer to any embedded elements (e.g., iframes) served from a different domain, or a different domain that the user visits in the future, as a third-party (3P) entity, and consider access to the browser mechanism made by these entities as 3P access.

Differentiating Tracking from Legitimate Access. Browser mechanisms may save and reuse state across browsing sessions for various purposes. While certain tracking behaviors (e.g., in 3P contexts) can be straightforwardly assessed as privacy-invasive, other scenarios may be more ambiguous. In our work we consider a capability to be privacy-invasive (i.e., suitable for tracking) based on two factors:

- *Intended Use.* While mechanisms like cookies, local storage, and indexedDB have been designed to store useful 1P information, their misuse in cross-origin contexts can result in privacy-invasive tracking behaviors. This scenario encompasses what we refer to as 3P tracking in the remainder of the paper.
- *Bypassing Existing Protections.* We consider the ability to re-identify users visiting from a private browsing mode session or after having cleared their browsing data to be a privacy-invasive tracking capability. This scenario encompasses what we refer to as 1P tracking in the remainder of the paper.

Threat Model. We assume that when a user visits a website the tested browser mechanism is used to store a unique 32-bit identifier in that browser instance. This identifier is then read back in future browsing sessions from the same browser instance. The attacker misusing the browser mechanism can be the visited website itself, or any included 3P entity (e.g., through the use of an iframe).

Figure 1 shows the major components of our framework, which we detail below. Given methods to interact with a browser mechanism, CanITrack curates contexts for various experimental scenarios under its test suite and handles the entire process for automating the testing pipeline. In a nutshell, a browser runner simulates user interactions within various browsing contexts, and generates page visits which execute the write and read methods for the browser mechanism within local browser instances. Following each experiment, the web server collects results in a database. Finally, these results are automatically gathered, analyzed, and filtered, before generating a vulnerability report for the mechanism.

❶ **Browser Mechanism**: CanITrack's inputs are JavaScript methods for interacting with browser mechanisms. These methods are formatted in such a way that CanITrack's client-side testing functions can independently store information (`write`) and access existing information (`read`) from the browser. While certain types of data can be directly accessed (e.g., data stored in local storage), other types of data can only be indirectly accessed (e.g., data fetched from a cache) or inferred based on the outcome of an action (e.g., the result of a client-side security policy being enforced by the browser). Using the framework for a new mechanism requires implementing the following two functions that will be called from within a client-side browser environment. Listing 1 provides example input methods that interact with a site's cookies.

1) **Write()**: A method that accepts a string *identifier* as input and translates that into a mechanism-specific action (or series of actions) that stores the input into the browser.
2) **Read()**: A method that implements a mechanism-specific action (or series of actions) that retrieves the previously-stored information from the browser. If successful, it will return the reconstructed *identifier*.

Abstracting the read and write methods allows CanITrack to be *agnostic* of the underlying mechanism. As new browser mechanisms that exhibit unique idiosyncrasies in terms of behavior and capabilities may be deployed in the future, our framework follows a modular design allowing it to be easily extended for handling additional mechanism-specific requirements that might arise. In the following paragraphs, we provide further details about our framework and elaborate on how the testing setup is modified depending on additional information about each browser mechanism.

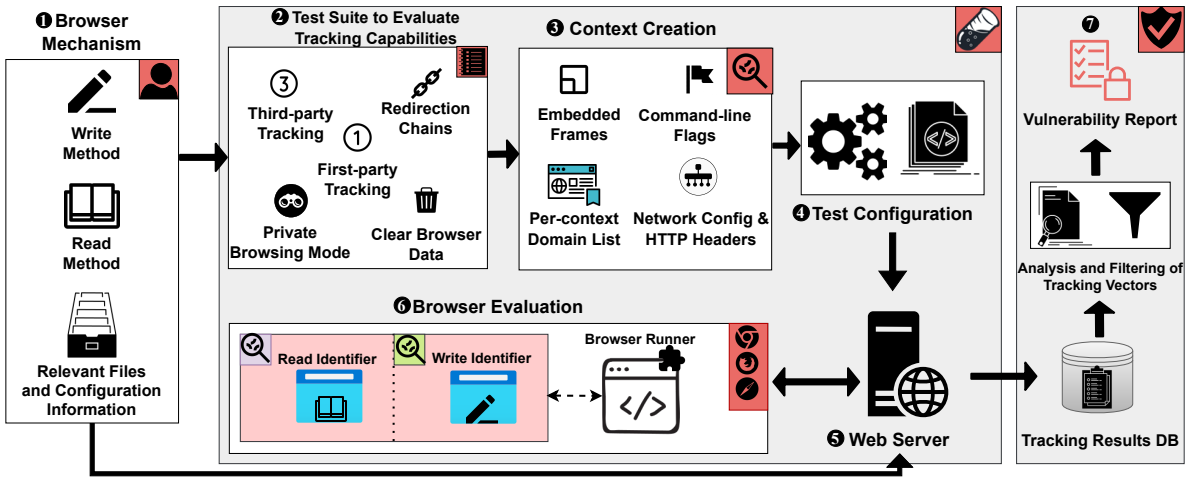

Fig. 1: An overview of CanITrack's architecture and testing pipeline, which evaluates browser mechanisms as potential tracking vectors across different browsing contexts.

```
write (identifier) {
    document.cookie = `identifier=${identifier};
        SameSite=None; Secure`;
}

read () {
   return document.cookie ? document.cookie.match
       (/identifier=(\S+)/)[1] : null;
}
```

Listing 1: *Methods to read/write an identifier with cookies.*

❷ **Testing Tracking Capabilities**: The tracking vectors that have been used in practice, or demonstrated by researchers, can vary significantly in terms of capabilities and suitability across different browsing contexts. As we aim to comprehensively and systematically assess whether a given mechanism can be misused for tracking, our framework must incorporate the necessary testing templates for uncovering all pertinent behaviors. To that end, we assembled a list of tracking capabilities and behaviors to be incorporated into our framework, inspired by the heterogeneous methods demonstrated by prior research [38, 67, 68, 70]. Next, we elaborate on the five dimensions of tracking explored by our framework's testing pipeline.

1P Tracking. Certain browser mechanisms provide a process for locally storing data on the client side based on previously observed resource requests. This data is typically associated with the specific domain that set it, and can usually be accessed by the domain itself in 1P contexts.

Websites may access browser mechanisms in legitimate, 1P contexts. The nuances of 1P accesses vary depending on the browser mechanism and the browser in question. While, conceptually, 1P access may not seem like a major privacy risk, implementation flaws that result in stored identifiers not being correctly purged, or websites being able to cross the isolation of private browsing [67], highlight the necessity of this dimension of our testing pipeline. CanITrack includes three 1P tracking tests that are applied to all browser mechanisms.

Additionally, the potential for 1P tracking indicates that the browser mechanism can further be exploited by colluding trackers adopting methods like redirection chains [2, 40] and cross-site leaks [55]. CanITrack evaluates three aspects of data access in 1P tracking, described below.

Can sites track in 1P top-level contexts? Data written by the browser mechanism can be read on subsequent visits to the same domain. CanITrack visits a website, *siteA.com*, and calls the `write()` method. On a subsequent visit to *siteA.com*, it calls the `read()` method, and observes if the same identifier can be accessed again.

Can sites track in 1P iframes? CanITrack checks if stored data can be read from 1P iframes, and is not restricted to the top-level browsing context (e.g., the favicon cache cannot be accessed from within an embedded iframe).

CanITrack visits a website, *siteA.com*, and calls the `write()` method. On a subsequent visit to *siteA.com* with an embedded 1P iframe (*siteA.com*), it calls the `read()` method from the iframe, and observes if the same identifier can be accessed again.

Can sites track across 1P subdomains? Websites that provide different services on different subdomains can have extended access to stored data that was added by different subdomains under the same top-level site. While we focus on tracking, recent work has explored security-related mechanisms within such a context [69]. CanITrack visits a subdomain, *sd1.siteA.com*, and calls the `write()` method. On a subsequent visit to a different subdomain, *sd2.siteA.com*, it calls the `read()` method, and observes if the same identifier can be accessed again.

3P Tracking. If data stored in the browser can be read or inferred by 3P domains (i.e., from a different origin than the one that set the data), then the mechanism can allow websites to track users across services. CanITrack evaluates three aspects of data access in 3P contexts.

Can sites track across 3P origins? If the browser mechanism is not partitioned and its stored data is globally accessible from all websites, it creates a significant privacy threat as it can potentially leak sensitive data, including information about the user's browsing history. CanITrack visits a website, *siteA.com*, and calls the `write()` method. On a subsequent visit to a differebt website, *siteB.com*, it calls the `read()` method, and observes if the same identifier can be accessed again.

Can sites track while embedded in different 3P contexts? Browsers that partition stored data with a single key may limit

access to the domain of the frame that added the entry. Without this protection, a site can read the same data entries while embedded across different 3P origins. For instance, assume that a browser visits *shopping.com*, which includes *ad.com* in an iframe. If *ad.com* adds data using the browser mechanism during this visit, the entry is keyed to *ad.com*. Thereafter, on a visit to a different site, say *news.com* which serves an ad from *ad.com* in an iframe, the embedded 3P, *ad.com*, can access the data it had stored during the browser's previous visit to *shopping.com* and identify the user. This allows the advertiser to track a user across websites.

Can sites track from different 3P iframes in the same top-Level context? Browsers that partition stored data with a single key may limit access to the domain of the top-level context. Such partitioning allows embedded 3P entities within the same site to share access to the entries of a browser mechanism. An example of a privacy-invasive attack in such a scenario would be as follows: say a browser visits *news.com* which serves an ad from *ad1.com* in an iframe. If *ad1.com* adds some entries using a browser mechanism during this visit, those entries are keyed to *news.com*. On a subsequent visit to *news.com*, a different ad in a different iframe from a different site *ad2.com* can read back the same entries that had been added by *ad1.com* and identify the user. This allows multiple advertisers to collude and track users, similar to cookie syncing [3, 36].

Partitioning key. Once vendors recognized the use of certain mechanisms as tracking vectors, browsers deployed mitigations by keying each resource entry to the context within which it was accessed. In addition to 1P and 3P tests, CanITrack includes additional tests that help determine both, the number of elements added to each browser mechanism's partitioning key, as well as the domain level (i.e., site (eTLD+1), subdomain, or port) of each element considered while constructing the partitioning key. These tests help determine the extent to which each mechanism provides tracking capabilities, and the limits of tracking use that each browser permits.

Redirection chains. Mechanisms that are limited to 1P access may not be directly available within 3P contexts. As a method to circumvent such restrictions, sites can redirect browsers through multiple domains, each of which accesses data in a 1P context before moving on to the next domain in the redirection chain. This way sites can access data for 3P tracking even when such entries are only available in 1P contexts.

Despite prior research [56] showing the use of redirections as a popular tracking vector, browser vendors vary in their mitigation strategies; while Safari attempts to block cross-site redirections [4], Google Chrome does not consider the vector to be a privacy-relevant issue. Regardless, past disclosures of tracking vulnerabilities that used redirection chains have demonstrated their practicality for tracking without significantly impacting the end-user's browsing experience [67].

CanITrack traverses the redirection chain by updating the value of `window.location` on the client-side. The *Test Enumeration* phase considers the minimum tests required to write and read a 32-bit identifier. These redirection chains can be traversed as both *top-level redirections* (i.e., navigating users through multiple domains) and *frame redirections* (i.e., only redirecting embedded iframes while the user accesses content on the top-level frame).

Private browsing mode. Browsers offer their users the option of accessing domains in a private browsing mode (i.e., *incognito mode*), intended for ensuring the user's privacy during that browsing session. This session is partitioned from normal browser storage and uses separate storage spaces whose lifetime is limited to that specific session. Data stored during regular browsing sessions is typically (but not always [67]) inaccessible within private browsing sessions, and data stored during the private browsing session is intended to be purged once the session ends [19]. If browsers omit clearing access to stored data before/after the use of private mode, or do not correctly isolate the use of stored policies [70], trackers may be able to correlate the activity of a private-mode browsing session with a regular session, thereby impacting users' privacy. CanITrack includes tests for detecting the leakage of stored data or use of client-side policies from, to, and within private browsing mode sessions.

Clear browser data. Regardless of the scope and use of a browser mechanism, browsers are expected to respect user decisions, especially when a user explicitly requests that stored browsing data be cleared. While browsers may allow such requests from extensions via browsing data APIs [17, 20], from developer tools [5], and even from websites themselves using a `Clear-Site-Data` header in their HTTP response [22], they place special emphasis on such requests being received via the user interface. For instance, the Trust Token API [45] mechanism checks if the request to clear tokens has been received from a user. However, it does not clear tokens even if the domain that issued them were to send a *Clear-Site-Data* HTTP header in subsequent responses [8].

CanITrack uses PyAutoGUI [34] to simulate user interaction. For instance, while testing against Chromium-based browsers, it "presses" `Ctrl+Shift+Del` to open the "Clear Browsing Data" menu. CanITrack first calls the `write()` method on one visit, clears the browser data, and then calls the `read()` method on a subsequent visit. It, therefore, tests if identifiers written by a browser mechanism can be read even after a user explicitly clears their browsing data.

❸ **Context Creation**: CanITrack's *Context Creation* component curates information relevant to the browsing context required for each experiment under the tests described above. It uses available configuration information as input, which includes details about the browser vendor, domains hosted by the web server, and information available about the browser mechanism. Next, we detail the different aspects that are considered while defining each context.

Embedded frames. The tests for *1P tracking*, *3P tracking*, *Paritioning key*, and *Redirection chains*, each include experiments involving embedded iframes. The context creation phase first determines the domains required for both the top-level context and iframe for each experiment (i.e., the 1P or 3P site, subdomain, or port).

Per-context domain list. In addition to determining domains for frames, certain mechanisms require a list of domains to access resources and to send network requests. CanITrack ensures that the list of domains that receive such requests remain consistent while writing and reading the identifier, and

these domains vary between tests for 1P and 3P tracking. Moreover, the *Context Creation* phase creates an additional list for redirection chains, which is comprised of a curated list of domains to traverse.

Network configuration and HTTP headers. CanITrack handles both, 1P and 3P requests by including a default header set that responds to cross-origin requests. It also listens to requests made on multiple domains and ports, which can be incorporated into tests for numerous browser mechanisms as-is. Nonetheless, if the existing defaults do not suffice, our framework offers flexibility for such accommodations. Network configuration changes may include handling server-side requests, serving hosted files, customizing headers for each request, and even setting up a parallel server on a different port on the web server.

Command-line Flags. One of CanITrack's most useful features is its ability to test experimental browser mechanisms accessible in browsers through specific command-line flags. For any such browser mechanism, the *Context Creation* phase accepts and includes these flags, to be later read by the *Browser Runner* before starting browser instances for relevant tests. In our experiments (§ IV) we use command-line flags to evaluate mechanisms within Google's Privacy Sandbox, as well as older browser versions that do not support Alt-Svc-based protocol updates to HTTP/3 requests by default.

❹ **Test Configuration**: The *Test Configuration* component interprets details about the context of each experiment and feeds them into existing scripts that perform the evaluations for each test. The test scripts are broadly comprised of two parts; first, the server-side scripts handle the creation of the HTML body for each experiment, which includes embedding and serving iframes, and setting any global variables needed by the client-side scripts. Second, the client-side scripts call the mechanism's abstracted `write()` and `read()` methods, and perform redirections if needed. The results from each invocation of the `write()` and `read()` methods are returned to the web server.

❺ **Web Server**: The server accepts requests for multiple domains and on multiple ports, and hosts all the logic and scripts relevant to communicating with the *Browser Mechanism*, the *Test Configuration*, and the *Tracking Results DB*.

Browser Mechanism. The *Web Server* makes the client-side scripts that include the `write()` and `read()` methods for the browser mechanism available when invoked by the *Test Configuration* scripts. In addition, it hosts resources and files, along with the logic provided by the mechanism to handle any network requests.

Test Configuration. The *Web Server* accepts network requests on behalf of the test scripts, and also makes the context of the request available for assisting test scripts.

Tracking Results DB. The *Test Configuration*'s client-side scripts send the results of each experiment back to the *Web Server*, which parses them, and adds them to the *Tracking Results DB*.

❻ **Browser Evaluation Steps**: The *Browser Evaluation* phase is primarily handled by a script, the *Browser Runner*, that interprets the configuration information, executes fresh browser instances, and creates new and appropriate contexts for each experiment. Prior to each experiment the runner makes sure that the browser has been completely closed and its state has been cleared, ensuring that each experiment is executed fresh and the operations performed in one experiment do not affect another. The *Browser Runner* then opens a new browser instance with any command-line flags specified. It opens a new window within a regular or private session, depending on the context of each experiment, before visiting different links to first *write* and then to *read* an identifier. The test scripts running within each visit send the results of these operations to the server using network requests. If an experiment requires clearing browsing data between writing and reading an identifier, the *Browser Runner* module executes a PyAutoGUI [34] script to simulate keyboard and mouse events that perform this operation. At the end of each experiment the *Browser Runner* closes the browser instance, and repeats the process for the next experiment.

```
"Chrome-v100": {
    "Overall": {
        "Track in 1P Contexts": true,
        "Track in 3P Contexts": false,
        "Redirections": n/a,
        "Track Into or From Private Browsing Mode
            ": false,
        "Track Despite Clearing Browsing Data":
            true
    }
    "1P Tracking": {
        "Track in 1P Top-level Contexts?": true,
        "Track in 1P iframes?": true.
        "Track Across 1P Subdomains?" : false
    }
    "3P Tracking": {
        "Track Across 3P Sites?": false,
        "Track While Embedded in Different 3P
            Contexts?": false,
        "Track From Different 3P iframes in the
            Same Top-
        Level Context?": false
    }
    "Partitioning Key": {
        "Number of Elements in Paritioning Key": 2
            ,
        "Key Composition": [
            {
                "frameLevel": "IFrame",
                "domainLevel": "Origin (Subdomain)
                    "
            },
            {
                "frameLevel": "Top-Level",
                "domainLevel": "Site (eTLD+1)"
            }]}}
```

Listing 2: *Example vulnerability report for the CORS Preflight Cache on Chrome v100.*

❼ **Vulnerability Analysis**: The *Tracking Results DB* contains fine-grained entries that include configuration information for the mechanisms, the browser configuration for each operation, and the domains and frames used within each experiment, in addition to the absolute values of the 32-bit identifier that were written in each context and the values that were retrieved from local data, as a result of the write and read operations.

Once the *Browser Evaluation* has been completed for all tests, an analysis script parses the individual entries from the *Tracking Results DB*. It creates a list of successful read

operations (i.e., where the read operations were able to reconstruct the identifier), and separates them from unsuccessful experiments, while taking the context of each experiment into consideration. The script compiles the results of these experiments into a simple, computer- and human-readable report, indicating the scenarios within which the browser mechanism can be used as a tracking vector. Listing 2 provides an example of our framework's output.

III. Exploring Browser Mechanisms

In this section we provide additional details about the browser mechanisms that we explore in our experimental evaluation. We gathered 21 browser mechanisms that were included based on three factors: first, we ensured that they were supported by at least one major browser vendor. Second, websites under our control could interact with entries in the mechanism by altering the DOM, calling a client-side Web API, or using HTTP response headers. Third, the entries in the mechanism persisted across subsequent visits to the same domain within the same browser instance. In addition to the `read()` and `write()` methods, the mechanisms that we evaluated comprised a diverse set of requirements, which needed both, server- and client-side setup, an overview of which is shown in Table I. We provide details about their individual read and write actions in Table VI in the Appendix, along with a more detailed example implementation to provide an estimation of the workload required to integrate a new mechanism in CanITrack. Due to space constraints, here we focus on the four mechanisms that have *not* been studied by prior work – Trust Token API, FLEDGE API, CORS Preflight Cache, and Client Hint Headers.

Google Privacy Sandbox. Google recently announced their plans to mitigate 3P cookie-based tracking and to experiment with and release a slew of different technologies (all part of their *Privacy Sandbox* initiative [26]) that aim to offer more privacy-preserving alternatives for numerous aspects of the web ecosystem, including online advertising and ad bidding (currently planned for late 2023 [18]). Given Google's dominant positioning, coverage and power within the web ecosystem, as well as Chrome's prevalence among browsers, this initiative can have severe and long-lasting privacy implications. As such, our exploration of browser mechanisms that can be misused for tracking also includes *two* of the main components of Google's Privacy Sandbox, the *Trust Token API* and the *FLEDGE API*, that have been rolled out and are currently supported by certain major browsers.

Trust Token API. To allow advertisers to differentiate trusted users from bots when serving ads, Google introduced the Trust Token API as a cross-origin mechanism for websites to communicate *trust* within a browser instance [45]. For example, consider a user visiting *shopping.com*, which embeds Google Ads. During this visit, Google can use its reCAPTCHA mechanism [31] to identify that the user of the current browser instance can be "trusted" as a real user, and can therefore be served advertisements. Google can issue multiple *Trust Tokens* that are stored within the browser as a way to remember such trust in the future. Following this, if the user visits a different website *travel.com* which embeds Facebook Ads, before Facebook actually displays an advertisement it can request the browser to provide a *Trust Token* from Google, if one exists. It can then send this token to Google, and redeem it. This way, Facebook can learn that Google has already verified the user, and serve advertisements without needing to perform such a verification again.

```
write (uniqueID, domainList) {
    for (let i = 0; i < domainList.length; i++) {
        if(uniqueID[i] == '1') {
            fetch(`https://${domainList[i]}/tokens
                `, {
                method: "POST",
                trustToken: {
                type: "token-request",
            }}}}

read (domainList) {
    let uniqueID = '';
    for (let i = 0; i < domainList.length; i++) {
        let ifExists = await document.
            hasTrustToken(`https://${domainList[i
            ]}`);
        if (ifExists) {
            uniqueID += '1';
        } else {
            uniqueID += '0';
        }}
    return uniqueID;}
```

Listing 3: *Example read and write methods used to evaluate the Trust Token API with CanITrack.*

The Trust Token API uses the Privacy Pass protocol [43] as an underlying cryptographic primitive, which ensures that tokens are unlinkable (i.e., when Facebook redeems a token Google does not learn which exact browser instance the token belongs to). A service using the Trust Token API additionally needs to set up TLS-based cryptographic functions on its end and advertise its public key commitments at a *Well-Known URI* [51]. Google has also placed additional limits on the number of tokens each website can redeem, allowing only 2 calls to be made per top-level browsing context, in order to prevent malicious actors from exhausting all tokens [35].

Despite restrictions, the API is fundamentally a cross-origin communication mechanism, made especially easy by having each token associated with an origin. Google also included `document.hasTrustToken(<origin>)`, a client-side API call that can be used to query the existence of a 3P token, without the intricacies of the cryptographic operations put in place by the Privacy Pass protocol for redeeming trust tokens. This creates a mechanism for writing and reading a unique identifier, based on a unique set of origins, to be used as a tracking vector. Listing 3 shows an example implementation of the `read()` and `write` mechanisms.

Writing an identifier. A trust token can be issued by adding an additional attribute to one of three existing methods, a Fetch request, an XML sHTTPRequest, or an iframe tag. As stated in Chrome's documentation, "these APIs are not restricted to being called in any particular origin's context" [8]. With two such issuance requests allowed under each top-level browsing context, a total of 16 redirections would be required to write a 32-bit identifier.

Reading the identifier. Issuing a call to `document.hasTrustToken(<origin>)` returns a Promise that resolves to `True` if a token exists for the `<origin>` or `False` if no such token exists. With a

TABLE I: *Overview of the diverse browser-mechanism setups that CanITrack supports. ● denotes a requirement for a browser mechanism, ◐ denotes partial requirements for browser mechanisms. Specific to the Routing Setup, rows that include multiple ● can be evaluated with any one such setup.*

Mechanism	DOM Interaction	Web API	Network Requests	File Resources	HTTP Headers	Server Configuration	Command-line Flags	Routing Setup			
								Paths	Ports	Subdomains	Sites (eTLD+1)
Cookies		●									
Local Storage		●									
indexed DB		●									
Cache Storage		●	●								
Stylesheet Cache	●		●	●	●			●	●	●	●
Font Cache	●		●	●	●			●	●	●	●
Image Cache	●		●	●	●			●	●	●	●
HTTP Disk Cache			●	●	●			●	●	●	●
Favicon Cache	●		●	●	●			●	●	●	●
Service Worker Variable Scope		●		●				●			
Service Worker Cache		●		●				●			
Alt-Svc			●		●	●	◐		●	●	●
HSTS				●		●				●	●
HTTP Auth			●		●				●	●	●
CORS Preflight			●		●				●	●	●
Accept-CH			●		●					●	●
NEL			●		●					●	●
Filesystem API		●									
WebSQL		●									
FLEDGE API		●		●			●			●	●
Trust Tokens		●					●		●	●	●

restriction of two such calls under each top-level browsing context, a total of 16 redirections would be required to reconstruct a 32-bit identifier.

FLEDGE API. Google proposed this API to facilitate remarketing and advertising to custom audiences in the absence of 3P cookies [46]. FLEDGE helps advertisers save user interests in the browser and read these interests back when placing bids for showing advertisements in future visits across different sites. Consider, the user visiting *shopping.com*; this website can add the user to an Ad Interest Group named "sneakers enthusiast", using a call to `navigator.joinAdInterestGroup()`. When the user visits a different website, say *news.com*, which sells ad space, the website can call another FLEDGE API, `navigator.runAdAuction()` with a list of buyers, including *shopping.com*, that can bid for the ad space. Here, *shopping.com* can access any Ad Interest Groups that it had previously saved in the browser instance. If it finds that the browser belongs to a specific interest group it can place a higher bid for showing a relevant advertisement during the current visit to *news.com*.

Google has placed a few privacy-focused restrictions on the FLEDGE API. Each browser instance regularly queries two advertiser-controlled endpoints within an interest group: the *dailyUpdateURL* used by advertisers for periodically updating interest group information en masse, and the *renderingURL* from where the browser fetches an individual advertisement. The API restricts the use of these two components by requiring that the same endpoints be observed by at least 100 other browser instances. No such restriction exists on the entire Ad Interest Group. Additionally, an ad auction that results in a winning bid is returned as an opaque source (example: `urn:uuid:c3697...`), a value that can only be deciphered by a new, sandboxed, HTML Element called *Fenced Frames* [24]. However, the API allows any origin to have the browser join Ad Interest Groups, including 3P iframes with a Permissions-Policy directive [13]. Moreover, ad auctions can be run with a single buyer bidding for the advertising space. While a successful auction returns an opaque source to the seller, an auction that ends without a winner returns a `NULL` value. The FLEDGE API Explainer itself points out that "this non-opaque return value leaks one bit of information to the surrounding page" [25]. Listing 4 shows an example implementation of the corresponding `read()` and `write` mechanisms.

Writing an identifier using the FLEDGE API. A browser can be added to an Ad Interest Group by passing the interest group object as an argument to *navigator.joinAdInterestGroup()*. If a 3P element like an iframe makes the API call it requires a Permissions-Policy directive of "join-ad-interest-group". In order to set a 32-bit identifier, up to 32 origins will make API calls (i.e., all origins corresponding to a value of "1" in the identifier), adding the browser to at least one Ad Interest Group from each origin.

Reading the identifier. A site can run multiple ad auctions on a single page visit by making calls to `navigator.runAdAuction()`. Each auction can involve a single buyer. If a buyer can access their respective Ad Interest Group from the browser, they can use it to bid in an auction which will return an opaque source. If they find no such Ad Interest Group, the bid can end without a winner and return a NULL value. The results from 32 such auctions can be used to reconstruct a 32-bit identifier. While no restrictions are currently in place for the number of calls to `navigator.runAdAuction()`, Google is experimentally evaluating an 8-auction limit per page visit, behind an additional flag [21]. If this limit were to be turned on by default, reading a 32-bit identifier would require either 4 redirections or 4 page reloads, which would marginally increase the effort required for using this tracking vector.

CORS Preflight Cache. When websites request resources from an origin other than the top level browsing context (referred to as Cross Origin Resource Sharing or CORS), browsers can issue so-called "preflight" requests, if they determine that these requests may be sensitive (e.g. AJAX requests with custom HTTP headers). These requests use the OPTIONS method and are used to ask for permission from the 3P server, before they send the actual request that the cross-origin site intended. If the server responds with the appropriate permission headers, the full cross origin request

```
write (uniqueID, domainList) {
  for (let i = 0; i < domainList.length; i++) {
    if (uniqueID[i] == '1') {
        let iframe = document.createElement('
            iframe');
        iframe.src = `https://${domainList[i]}`;
        /*  Within each iframe:
            navigator.joinAdInterestGroup({
                owner: `https://${domainList[i]}`
                ...
            }, 3600*24*30))
        */
        let body = document.getElementById('body'
            );
        body.appendChild(iframe);}}
read (domainList) {
    uniqueID = '';
    for (let i = 0; i < 32; i++) {
        adAuctionResult = await navigator.
            runAdAuction({
            interestGroupBuyers: [`https://${
                domainList[i]}]
            ...
        });
        if (adAuctionResult == null) {
            uniqueID += '0';
        } else {
            uniqueID += '1';
        }
    }
    return uniqueID; }
```

Listing 4: *Example read and write methods used to evaluate the FLEDGE API with CanITrack.*

can be fired. This is an effective defense against CSRF attacks and prevents unauthorized requests from causing side-effects on the server [65]. All major browsers cache the preflight requests for cross origin resources for performance reasons, thus matching our description of a potential tracking mechanism. However, we note that while other mechanisms described in this section have extended lifetimes, the CORS Preflight Cache is shortlived, and only persists for 24 hours on Firefox and 2 hours on Chrome [15].

Writing an identifier using CORS Preflight cache. Every time a cross-origin resource is requested, the browser fires an OPTIONS request to the server. The response of this OPTIONS request will be cached for future requests for the same resource. In order to write a 32-bit identifier, the client-side code can generate a bitmap representing the identifier to be written, and assign a single cross-origin resource for each bit in the bitmap. Subsequently, the client will issue cross origin requests corresponding to the appropriate bit in the identifier. This will result in the identifier being encoded in the browser's preflight cache and available for later use.

Reading the identifier. In order to read a previously stored identifier, the client will re-generate the same bitmap that was created in the write phase. Then, the client will issue preflight requests for each of the 32 resources mapped to the bits of the identifier. Meanwhile the server maintains a set of resources for which it receives an OPTIONS request, which indicates that the preflight cache was cold for those. These resources correspond to the `0` bits of the identifier, and are used for re-constructing the previously written identifier.

Client Hint Headers. To optimise content delivery based on device and network characteristics, Chromium-based browsers support *client hints*, wherein the browser includes information about the client along with the HTTP requests (sent in the form of HTTP headers). The `Accept-CH` header in a HTTP response allows the server to request specific client hint headers from the browser. For instance, the `Accept-CH: Viewport-Width` response header directs the browser to supply the width of the client viewport in the `Viewport-Width` header on subsequent requests.

TABLE II: *Statistics about browsers tested using CanITrack.*

Browser	Versions	Period	Market Share [10]	Tested Mechanisms
Brave	20 (v1.3 -v1.37)	02/2020 - 04/2022	<1%	21
Chrome	20 (v80 - v100)	02/2020 - 04/2022	67.17%	21
Edge	20 (v80 - v100)	02/2020 - 04/2022	9.14%	21
Firefox	20 (v80 -v99)	08/2020 - 04/2022	7.87%	15
Opera	20 (v67 - v86)	02/2020 - 04/2022	2.89%	21
Safari	4 (v12.1.2 - v15.4)	07/2019 - 04/2022	9.63%	13
Tor	22 (v9.0 - v11.0.10)	10/2019 - 04/2022	–	15

Writing an identifier using Client Hint Headers. The value of the `Accept-CH` response header is stored in the browser for future requests, and hence can be used for storing an identifier. Upon receiving a request the server generates a bitmap of the identifier which needs to be written, and maps a single client-hint to each bit of the identifier. Now, the server populates the `Accept-CH` header only with client-hints for which the bit value is `1`, and responds to the client. The client stores this `Accept-CH` directive for future requests, which can be used to reconstruct the identifier.

Reading the identifier. The server can retrieve an existing identifier by reading the client-hint headers that were sent to it along with a request. The server re-generates the mapping of bit positions and client hints that were used and checks which client-hints were sent in the requests. These correspond to the identifier-bits set to `1` allowing the server to reconstruct the identifier that was written.

IV. Evaluation

In this section we evaluate CanITrack, and identify browser mechanisms that can be misused as tracking vectors. We choose 126 versions of seven major browsers (see Table II) as a representative sample of the browser ecosystem over a 2-year period, during which browsers deployed a series of countermeasures (including redesigning their origin-partitioning architectures). We present our findings for six of the tested browsers below, and separately provide some observations on the Tor browser.

1P Tracking. Initially, we explore which of the tested mechanisms can be used as a tracking vector in a 1P context. As can be seen in Table III, these mechanisms are overwhelmingly accessible in top-level contexts and can be used to write and read identifiers across visits, each affecting the latest version of *at least* one browser. CanITrack also evaluates the mechanisms that can be accessed within 1P iframes. Such access can help websites separate the context used for tracking from the context used for their user-facing services. Table III shows that apart from the favicon-cache all the other evaluated browser mechanisms that can be accessed in top-level contexts can also be accessed within 1P iframes. Additionally, we check whether browsers provide unified access to the mechanisms from all subdomains under the main domain (eTLD+1). This access allows sites that provide a large number of services to share tracking identifiers across their subdomains even when a

TABLE III: *Number of browser mechanisms that can be used for 1P tracking across different scenarios.*

Browser	Versions	Number of Vulnerable Mechanisms		
		1P Top-level	Site-wide	1P IFrame
Brave	1.3-1.15	17	8	16
	1.17-1.37	16	7	16
Chrome	80-83	19	8	18
	84-90	20	9	19
	91-100	21	10	20
Edge	80-83	19	8	18
	84-90	20	9	19
	91-100	21	10	20
Firefox	80-100	15	7	14
Opera	67-69	19	8	18
	70-76	20	9	19
	77-88	21	10	20
Safari	12-15	12	6	12

TABLE IV: *Non-standard APIs supported by Chromium-based browsers.*

Mechanism	Version Introduced	Status
Accept-CH	Chrome(46), Edge(79), Opera(33)	Enabled
FLEDGE	Chrome(91), Edge(91), Opera(77)	Experimental
File System API	Chrome(13), Edge(79), Opera(20), Brave(0.57)	Enabled
Network Error Log	Chrome(71), Edge(79), Opera(58)	Enabled
Trust Tokens	Chrome(84), Edge(84), Opera(70)	Experimental
WebSQL	Chrome(4), Edge(79), Opera(10.5), Brave(0.57)	Deprecated

user is not registered with each service individually. All tested browsers support at least six such mechanisms in their latest versions that enable site-wide tracking capabilities.

Non-standard or deprecated mechanisms. We also observe that Chromium-based browsers have adopted non-standard APIs and extended support for deprecated APIs long after the plans for deprecation were made public, as detailed in Table IV. Google Chrome still supports WebSQL and the legacy version of the File System API (via *window.webkitRequestFileSystem()*) despite both APIs being deprecated for over 3 years [39, 52]. Support for such APIs can be observed across the Chromium-based browser family (including Brave, Edge and Opera) expanding the viability of its use for tracking to users of those browsers as well. An additional example is Network Error Logging [41], which uses an older version of the Reporting API [42]. While browsers like Firefox hide the API behind flags [32], Chrome enabled support for the now-deprecated mechanism by default in 2018, and continues to support its use in the latest version [61]. While Brave blocks use of the header by default due to its potential for misuse, other popular Chromium-based browsers (e.g., Edge, Opera) do not include such restrictions. Mechanisms made available using Google's Privacy Sandbox, as described in §III, remain unadopted by other major browsers like Firefox and Safari. While Brave blocks access, Edge and Opera support them, thus increasing the number of users affected by the tracking vectors that these protocols enable.

The curious case of Safari. While Chromium-based browsers are affected by the adoption of non-standard mechanisms, Safari takes a conservative approach even in its implementation of standard browser mechanisms. Safari has added support for HTTP/3 as an experimental feature in its Technology Preview Version [27], but does not support upgrades from HTTP/1.1 to HTTP/3 based on the Alt-Svc header - a mechanism adopted by all of the other browsers that we evaluated [27, 28]. While this protects the browser

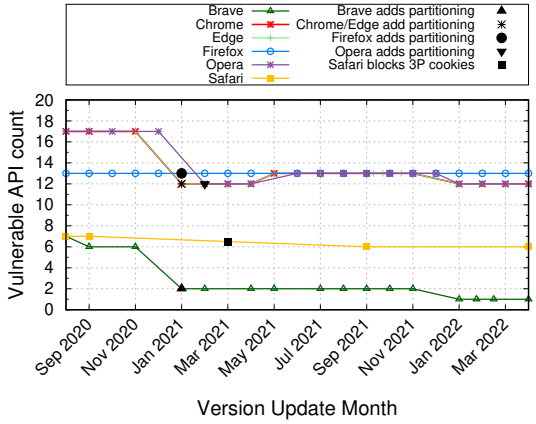

Fig. 2: *Browser mechanisms that can be used as 3P tracking vectors. A breakdown is provided in Table V.*

from tracking vectors enabled by this feature, websites have to advertise all supported protocols, including HTTP/3, by editing their DNS entries, which would result in only one of the protocols being used from the first visit itself, instead of a future upgrade to a dynamically alterable domain or port [7, 50]. Moreover, Safari mitigates HSTS-based tracking by limiting upgrades in HSTS State to the entire site (eTLD+1), therefore reducing the number of bits that can be set for every site to one. By additionally restricting such state upgrades to 1P links, they prevent trackers from abusing this mechanism [49]. Safari also identifies cross-site top-frame redirects, and classifies it as *bounce tracking*, further reducing the feasibility of creating an identifier across 32 sites in a redirection chain [6].

3P Tracking. CanITrack also assesses access to each tested browser mechanism in three different 3P contexts, as explained in §II. Browsers that allow cross-site access offer the same view of the mechanism's state to all domains visited by the same browser instance. This form of global access allows malicious or invasive actors to track users across multiple browsing contexts without needing to be the 1P top-level context when reading or writing an identifier. As can be seen in Figure 2, CanITrack reveals that in the past two years all major browsers have allowed such unrestricted, cross-site access to at least one of the evaluated browser mechanisms. Interestingly, we observe that for certain mechanisms browsers realize the possibility of misuse and subsequently isolate these mechanisms to the domain that accessed them. Moreover they do so using different approaches, shown in Table V, further highlighting the requirement for testing the additional contexts included in CanITrack's 3P tracking test suite.

Partitioning Key. Browsers add a key to each entry associated with a browsing mechanism. This key includes the URL of each resource associated with the mechanism, in addition to the context that made such an entry in the browser. The context considered for the key varies across browsers. For instance, consider the version updates observed in January 2021 for Firefox (v85) and Chrome (v87), as shown in Figure 2. Both browsers identified potential 3P tracking issues in prior versions, enabled by making the same view of the Stylesheet cache, Image cache, Font cache, and the HTTP Disk Cache available to all domains. They both chose to key entries to these mechanisms using additional context considered in each request. Chrome used the domain (eTLD+1) of the frame (if

TABLE V: *Breakdown of 3P tracking capabilities.*

Browser	Versions	3P Tracking Contexts			Total
		Cross-site	IFrame Across 3P Contexts	3P IFrames in a Site	
Brave	Pre-partition (v1.17)	6	0	0	6
	Post-Partition (v1.19)	2	0	0	2
Chrome	Pre-Partition (v87)	8	9	0	17
	Post-Partition (v88)	4	8	0	12
Edge	Pre-Partition (v87)	8	9	0	17
	Post-Partition (v88)	4	8	0	12
Firefox	Pre-Partition (v84)	7	6	0	13
	Post-Partition (v85)	0	6	7	13
Opera	Pre-Partition (v73)	8	9	0	17
	Post-Partition (v74)	4	8	0	12
Safari	Pre-Block 3P Cookies (v13)	1	2	4	7
	Post-Block 3P Cookies (v14)	1	1	4	6

the entry was added by an iframe) and the site of the top-level context, in addition to the URL of the resource. For instance, consider that a font available at (`font.com/f.tff`) was added by an iframe (`iframe.com`) while embedded in another site (`news.com`). Starting from v87, Chrome keys each entry in a way that considers the entire context. In our example the key will include `news.com, iframe.com, font.com/f.tff`.

Firefox, on the other hand, also identified that those mechanisms and two additional ones (Alt-Svc and HSTS) can be misused for tracking. Following an alternative strategy, Firefox chose to key each resource only to the top-level site (eTLD+1) under which such an entry was added to the mechanism. In the same example scenario, Firefox wll add the new font with a key that includes `news.com, font.com/f.tff`, thus ignoring the domain of the iframe under which the request was made. Table V shows that considering a partial view of the context in each key leaves Firefox vulnerable to 3P tracking, albeit in a reduced number of scenarios. We additionally observe that Safari adopts a similar approach to their partitioning of similar browser mechanisms, i.e., Fonts, Stylesheets, Images, and the HTTP Disk Cache.

We observed that Chrome's adoption of keys for 4 mechanisms (i.e., Fonts, Stylesheets, Images, HTTP Disk Cache) were also inherited by Brave. Additionally, Brave partitioned the Alt-Svc header (v1.33, 2022), and restricted the use of the favicon cache (v1.15, 2020), independently of Chrome.

Inconsistency in domain levels used for partitioning. While all of the browsers use their own approaches to interpreting the context included in a partitioning key, they additionally vary in their understanding of the level of domains included in such keying. Namely, even though most browsers make cookies available to all subdomains under a site (eTLD+1), they restrict such access for local storage, indexed DB, and cache storage to each subdomain. They adopt similar variations with regard to the resource-based mechanisms explained before, including only the site (eTLD+1) as part of their key. This variation enables the *Site-wide Access* scenario shown in Table III.

Restricting Access in 3P Contexts. Another approach adopted by browsers for certain mechanisms is a blanket restriction of access from 3P contexts. Safari and Brave use this approach for cookies and mechanisms under the Storage API, i.e., local storage, indexed DB, and cache storage. Any accesses made to these mechanisms in a 3P context is considered to be ephemeral. Chrome adopted a similar approach to restrict access to WebSQL in later versions (>=v97). Imposing such restrictions to access overcomes the need for a partitioning key, ensuring that sites only adopt mechanisms for 1P tracking use-cases, and greatly restricts its misuse by malicious actors.

Redirections. For browser mechanisms that are limited in terms of the number of accesses that can be made to their entries with a single page visit, we evaluate them using redirection chains. We test whether browsers impose any restrictions on these chains, and whether depending on the "origins" that comprise these chains (i.e., a list of sites (eTLD+1), subdomains, or ports) results in a different treatment from browsers. In most cases where the mechanism is keyed to the origin of a domain, different subdomains and ports under the same domain (eTLD+1) are considered to be different origins. While a resource accessed for each new subdomain would require the resolution of a new DNS request, resources accessed from different ports of the same site can do so without the DNS overhead (or the management of additional subdomains). We observed a reduction of 0.8 seconds in the average time taken to perform 16-redirections across a chain of ports to set a 32-bit identifier using the Trust Token API, in comparison to similar redirections that used a chain of subdomains instead.

Clearing Browser Data. All of the browsers we evaluate offer users a method to clear browser data, including their history, cached files, and any cookies stored in the browser. CanITrack verified prior reports of incomplete data removal with regard to the favicon cache [67]. We found that the options that were selected by default when Chrome and Brave users accessed the "Clear browsing data" menu from the browser's settings tab, failed to clear the favicon cache. Older versions of Safari (<=v14), similarly did not clear the favicon cache from either of the user actions that they provided, i.e., the "Clear History" option under the "History" menu and the "Manage Website Data" option under the "Preferences" menu.

Private mode leaks. We also verified prior findings about the favicon cache in older versions of Chrome (<=v91) and Safari being available when the user visits a site in private mode [67]. This enabled tracking vectors that re-identified users that had previously visited a service in normal browsing mode. These checks highlight the need for a completely new, sandboxed profile of all browser mechanism entries upon creating a fresh instance of a private browsing context.

Tor observations. The Tor browser, which is based on Firefox, adopts a privacy-focused approach wherein browsing sessions use the private browsing mode by default; when users quit and reopen the browser, any private information linked to the profile (cookies and browsing history) are cleared [63]. As a result of this unique design, the states of the browser mechanisms that we test are linked to the browsing profile and are cleared each time the browser is quit and re-opened. However, the state of 11 mechanisms persists within the same "identity", i.e. across different visits without the user quitting

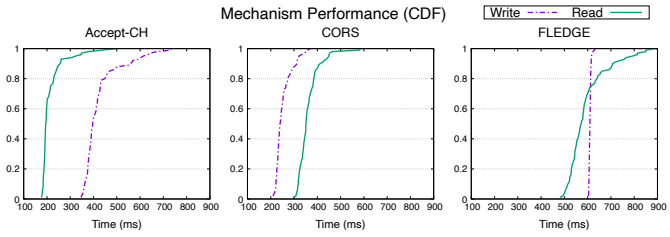

Fig. 3: *Overhead of writing and reading a 32-bit identifier using CORS, Accept-CH, and FLEDGE.*

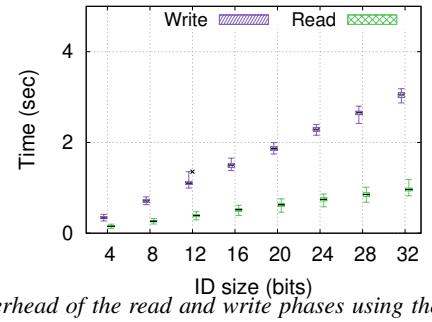

Fig. 4: *Overhead of the read and write phases using the Trust Token API across different ID sizes.*

the browser in between. Of these, 6 mechanisms (Alt-Svc, Font Cache, HTTP Auth, HTTP Disk Cache, Image Cache, and the Stylesheet Cache) are keyed in a similar manner to Firefox, and can be read by different 3P iframes under the same site. Unlike Firefox, Tor doesn't provide a menu to manage and clear browser data. Instead, it provides an equivalent "new identity" button, which clears all cookies and browsing history in addition to using new Tor Circuits for future connections [1]. While this feature works in a similar manner to quitting and re-opening the browser (i.e. it clears the states of all tested mechanisms) CanITrack found that the CORS Preflight Cache remains uncleared until the browser has been quit. Users can therefore be tracked in Tor despite adopting a "new identity", until they quit the browser.

Performance measurements. Apart from the feasibility experiments, we use CanITrack to evaluate the practicality of these tracking vectors in terms of performance. We deployed a lightweight *Express.js* [23] web server on a Quad Core machine with 16GB of RAM. We placed our VM in the same city as the devices used during our evaluation. We leveraged a Puppeteer [30] script to orchestrate visits to our web server, and recorded the time it took to read and write a 32-bit identifier. We limit this experiment to the four browser mechanisms that we are the first to demonstrate as tracking vectors, and average the timing information from 100 separate tests in each scenario. Figure 3 shows that three of the mechanisms are extremely efficient as writing or reading a 32-bit identifier requires only 200-900 milliseconds. As can be seen in Figure 4, Trust Tokens introduce additional overhead due to their reliance on redirections, along with its specific implementation [14] of the cryptographic primitives included in the underlying Privacy Pass protocol [43]. Nonetheless, while the one-time cost of writing a 32-bit identifier requires three seconds, reading the identifier only takes about one second. Importantly, this can be further optimized by leveraging immutable fingerprints as a source of identifier entropy [67].

Additional notable findings. During our evaluation, CanITrack unearthed new tracking vulnerabilities and capabilities in the latest versions of evaluated browsers. These vulnerabilities are additional to the four novel tracking vectors (see §III).

Unpartitioned Alt-Svc in Chromium Browsers. Using the Alt-Svc header to track users across websites was previously reported by Tiwari et al. [71] in 2019. Following this work, Chrome imposed restrictions on using Alt-Svc headers for upgrading requests to use HTTP/3, requiring that such servers exist in parallel with an HTTP/1.1 or HTTP/2 server. They imposed additional restrictions on the ports that can be used by these parallel servers, requiring that they both be served either on ports < 1024 or on ports $>= 1024$. They also require that servers have TLS certificates signed by a Certificate Authority already in Chrome's list of trusted CAs [60].

CanITrack's testing pipeline revealed that despite these restrictions the latest version of Chrome (v103) keeps the Alt-Svc cache unpartitioned. With support for HTTP/3 being enabled on Chrome by default since v87, the browser reads any HTTP response that returns a valid Alt-Svc Header and upgrades future requests to use the HTTP/3 protocol. This behavior allows malicious actors to write an identifier in any context within a regular browsing session, and any other malicious actor to read the same identifier in a different context during future visits, obviating the need to rely on redirections or insecure contexts. Finally, we observed that while Brave used a partitioning key for its Alt-Svc entries, no such partitioning existed for other Chromium-based browsers, including Edge and Opera.

CORS Preflight in Private Browsing. Prior to including appropriate tests in CanITrack, we found that cached CORS Preflight responses for cross-origin resources were leaked between subsequent private browsing sessions in Firefox. If the mechanism was used to store an identifier during a visit in private browsing mode, this identifier would persist even if the private window was closed and another one was opened at a later time. We reported our findings to Firefox, which led to this vulnerability being patched. We then designed a test to evaluate the behavior of mechanisms within private browsing sessions, and included it in the test suite offered by CanITrack.

CORS Preflight Cache following Clearing Browser Data. During our evaluation, CanITrack reported that tracking identifiers persisted despite user-initiated clearing of browser data on the latest versions of Chrome, Safari, and Tor. Upon further inspection, we found that user-initiated data clearing, in the context of preflight responses, does not take effect until the browser is completely closed. We observed similar behavior in Brave, Edge, and Opera as well. Browsers failing to clear the CORS-Preflight cache will result in the vector persisting until its expiration, despite the user requesting their removal.

Favicon as a Global Cache. While evaluating the latest versions of Chrome and Safari, CanITrack revealed that cross-site favicon links could be used to write and read identifiers. This flaw was also inherited by Edge and Opera, whereas Brave correctly partitions this mechanism. While previous work [67] demonstrated how favicons can be misused for tracking, that work did not identify or report the feasibility of cross-origin requests, and focused on same-origin tracking. We note that despite the disclosure of that attack, the ability to misuse favicons for tracking remains. More importantly, our system revealed that browsers allow cross-site favicon links

and serve them to all sites from the same cache. The lack of any partitioning key results in favicons becoming a cross-origin vector, which we leverage for developing a novel history sniffing attack that we describe below.

Favicon Leaking into Private Browsing Mode. CanITrack reported that while Safari cleared favicons on UI-triggered actions in Safari v15, the browser continued to serve favicons from the cache when a user visited domains while using the private browsing mode. This indicates an incomplete fix of previously reported bugs.

History Sniffing using Favicons. Here, we describe a novel history sniffing attack that we designed following our experimental findings from CanITrack's testing of the latest versions of major browsers. Browsers request favicons for a website based on the `href` attribute of the `link` element included in the returned page. The attribute can point to any 3P URL or path indicated in the element. If the browser finds an existing entry for the favicon in its cache, it does not trigger a network request, instead fetching a cached copy of the previously requested favicon. The Performance API [29], which is available in most major browsers, provides information about network requests triggered to fetch resources. The entries returned by calls to this API include information about a request for the favicon only if a network request was made. If the favicon was accessed from the cache, no such entry will be found. Once CanITrack revealed that entries to the favicon cache were shared across websites, we gathered links to favicons of popular websites which we then visited. We observed that adding these 3P links in the `href` attribute of our test page caused the browser to fetch these favicons from the cache, and no corresponding entry was found in the list returned by the Performance API. We then developed two versions of a history sniffer.

Chromium Version. Chrome allows websites to dynamically change the favicons associated with a page by modifying the link element included in the DOM's head. We leveraged this feature to traverse a list of favicons gathered from popular websites, and added it to an attack page under our control. When a user visits our page after having previously visited any of the websites on the list, the attack page dynamically changes the link element associated with its favicon, as it traverses the list of targeted websites (i.e., the list of websites that we want to cross reference with the user's browsing history). The page includes a small ($\sim$100ms) wait between each update to ensure that a network request or a cache fetch is triggered. The page then calls the Performance API and traverses the list of resource requests returned by the API. Any favicon link that is not included in the returned list indicates a domain that the user has visited in the past. Moreover, the attack page can then associate a new favicon, under its own control, with its page. This helps "purge" tested 3P favicons from being associated with it in the browser's cache, thus ensuring that the attack can be re-run in future visits. Since the described attack makes use of dynamically changing favicons within the same page, this attack, unlike prior favicon-based attacks [67], does not incur the additional performance overhead added by redirections. A demo of the attack can be found here [72].

Safari Version. Unlike Chrome, Safari does not allow dynamic changes to the favicon associated with a page. As such, we develop a redirection chain, with each page in the chain requesting a single 3P favicon before querying the Performance API, and moving on to the next page. This history sniffing attack then reconstructs the user's history based on the values observed across multiple page visits. The attack works on the first visit to a page in regular browsing mode, after which Safari adds sniffed favicons to the cache. While the attack in Safari is not as stealthy, the privacy threat is exacerbated by Safari using the same cache from regular browsing sessions in the private browsing mode. Moreover, since favicon entries are not added to the cache when in the private browsing mode (i.e., the site can read but not write), this attack can be repeated each time a user visits the attack page in a new private browsing session. A demo of the attack can be found here [73].

Vulnerabilities across Chromium browsers. A large number of browser vendors rely on the underlying Chromium engine [74] for their functionality, including the implementation of the mechanisms that we evaluated. Vulnerabilities resulting from these implementations can be inherited by these browsers, exacerbating the effect of any privacy-sensitive flaw. For each vulnerability found during our evaluation of Google Chrome, we further evaluated their viability in Microsoft Edge and Opera, two popular Chromium-based browsers. All of the vulnerabilities described in this section, including the newly-evaluated browser mechanisms, affected those browsers as well. While Brave blocked access to Google's Privacy Sandbox, its latest version was vulnerable to an oversight in the clearing of the CORS-Prelight Cache. Additionally, Brave does not block inherited implementations of non-standard APIs like WebSQL and the legacy version of the File System API, both of which can be used as 1P tracking vectors.

Summary. We used CanITrack to evaluate a wide range of emerging and existing browser mechanisms and implementations across numerous versions over a two-year period. Our system unearthed novel vulnerabilities in the latest versions of all major browsers and guided the design of two versions of a new history sniffing attack. Moreover, CanITrack allowed us to confirm prior findings and also quantify the impact storage isolations and anti-tracking countermeasures deployed by browser vendors.

V. Discussion

Ethics and disclosure. We note that no users were affected by our experiments, all of which were conducted using our own devices or cloud-based virtual machines. Furthermore, we disclosed the individual tracking vectors uncovered by our system to all of the affected browsers. Importantly, due to the extensive public discourse around Google's Privacy Sandbox initiative and the long term ramifications for the web ecosystem that would result from a wider adoption, we preemptively notified major browsers (i.e., Safari, Firefox, Brave) that do *not* currently support the mechanisms we evaluated (i.e., TrustTokens and FLEDGE) about our findings. This will allow them to make a more informed decision moving forward about supporting these mechanisms. In total, we have submitted 20 bug reports to seven browser vendors.

CanITrack release and use cases. We developed our framework to be modular and extensible so as to allow other researchers to incorporate additional features and capabilities for exercising browsers and analyzing their respective

functionality. To that end, we will open source CanITrack upon publication — an anonymized repository can be found here [16]. Our system can facilitate and streamline the internal testing procedures of browser vendors during the development phase of new browser mechanisms, as well as allow comprehensive and systematic testing of existing features by the research community. Moreover, our framework can be used by researchers for evaluating the effectiveness of anti-tracking defenses they develop against specific types of online tracking.

VI. Related Work

CanITrack is the first automated system for comprehensively and systematically uncovering tracking vectors. Here, we list relevant studies that advanced our community's understanding of the tracking ecosystem and motivated our proposed framework's design.

Cookie-based Tracking. Cookies have long been used to track users across sites in both 1P and 3P contexts. The privacy-invasive nature of 3P entities that gather user data through a combination of cookies and other fingerprinting vectors has been measured and reported in prior work by Englehardt and Narayanan [48]. In a similar vein, Acar et al. [36] reported the use of *cookie-syncing*, where unique IDs were respawned by colluding trackers across different site visits, which helped them merge records of individual users. To mitigate misuse, browsers like Firefox [33] and Safari [75] added protections to limit their access in 3P contexts. Recently, Google released their Privacy Sandbox proposals [26], and announced plans to eventually phase out 3P cookies. Additionally, Dimova et al. [44] showed ways to bypass cookie-oriented restrictions using approaches like CNAME cloaking that help websites embed 3P tracking resources in 1P contexts.

Cookie-less Tracking. Besides cookies, multiple browser mechanisms have been shown to aid user tracking over the years [62]. In 2009, Soltani et al. [68] demonstrated the misuse of Flash cookies, while in 2010, Kamkar [54] demonstrated similar misuse of localStorage, sessionStorage, and ETags. More recently, mechanisms like HSTS for websites not included in the preload list [49] and the Favicon Cache [67] have been shown to enable similar tracking. While browsers have partitioned mechanisms when they realize their potential for misuse [12, 19], new vectors, like those presented in our work, go unnoticed until evaluated and reported. The lack of a structured approach to identifying tracking vectors further amplifies the possibility of privacy-invasive behaviors going unnoticed for a long time. CanITrack aims to reduce this gap, and to offer a streamlined and comprehensive system for evaluating mechanisms.

Longitudinal Studies. Online tracking has been studied at scale [37, 48] and retrospectively [57]. These studies also presented frameworks for detecting the use of known tracking vectors by sites, and measured the extent of tracking in the wild. They further showed the importance of detecting non cookie-based vectors, given the extent of use by trackers.

Frameworks. Next to frameworks that analyzed tracking across websites, recent work has also suggested systems for evaluating mechanisms that enable cross-site communication [55, 66]. More relevant to our work is the recent PrivacyTests project [47]. The service tests and provides a snapshot of the state of known supercookies, blocking of tracking content, and fingerprint resistance measures within the latest versions of browsers. While their evaluation of supercookies is similar to those proposed by CanITrack, they only cover a single aspect of the 1P and 3P tracking tests evaluated by our system (see §II). Our system includes a suite of additional tests that capture tracking vectors across multiple 1P and 3P contexts, and further evaluate the composition of a partitioning key associated with a mechanism. Our system also supports the evaluation of vectors that can benefit from redirection chains, and verifies the possibility of optimizing tracking vectors by replacing the use of subdomains with ports. Additionally, our system uncovers leaks into, from, and within private browsing modes, and assesses the effects of clearing browser data. These tests help determine the extent to which each mechanism provides tracking capabilities, and the limits of tracking use that each browser permits. The systematic approach used by our framework makes it easy to test a plethora of browser mechanisms with various configuration requirements, including the flexibility to handle network requests, host resources, customize HTTP headers, and set up parallel servers on different ports (see Table I). Finally, CanITrack can be used in the evaluation of new, unreleased, experimental features hidden behind command-line flags.

Overall, the motivation behind our work was recognizing the need for a structured and comprehensive methodology and system for assisting developers and researchers in uncovering the tracking risk introduced by browser mechanisms. To address that gap we developed CanITrack, and demonstrated its capabilities by analyzing a multitude of heterogeneous mechanisms. In fact, our system was able to identify the privacy threat introduced by four mechanisms that have not been previously analyzed (including two high profile proposals from Google's Privacy Sandbox) and unearth previously undiscovered bugs in existing mechanisms.

VII. Conclusion

With the web playing a pivotal role in some of our most private and sensitive moments, ensuring the privacy of our online activities has become a matter of paramount importance. This complex ecosystem is driven by the ever-evolving set of browsers that mediate our online actions and communications. As more features get incorporated, systematically testing the privacy risk introduced by new mechanisms has become a daunting task. To facilitate and streamline research around online tracking we have developed CanITrack, a mechanism-agnostic framework that comprehensively assesses whether a browser mechanism can be misused for tracking purposes under different scenarios. To demonstrate the utility of our system, we presented an extensive evaluation of 21 browser mechanisms, including four that to the best of our knowledge have never been analyzed before. Our experiments uncovered a wide range of flaws, with the latest version of *every* browser we tested being vulnerable to *at least one* tracking technique. Overall our findings highlight the importance of employing principled and comprehensive browser-auditing strategies for detecting and tackling the severe privacy threats that users face, and we believe that our system addresses a significant gap that currently exists.

REFERENCES

[1] "New Identity | Tor Project | Support." [Online]. Available: https://support.torproject.org/glossary/new-identity/

[2] "Redirect tracking protection - Privacy, permissions, and information security | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Privacy/Redirect_tracking_protection

[3] "What is Cookie Syncing and How Does it Work? - Clearcode Blog," Dec. 2015. [Online]. Available: https://clearcode.cc/blog/cookie-syncing/

[4] "Intelligent Tracking Prevention 2.0," Jun. 2018. [Online]. Available: https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/

[5] "View Cache data," 2019. [Online]. Available: https://developer.chrome.com/docs/devtools/storage/cache/

[6] "Tracking Prevention in WebKit," Jun. 2020. [Online]. Available: https://webkit.org/tracking-prevention/

[7] "Accelerate networking with HTTP/3 and QUIC - WWDC21 - Videos," 2021. [Online]. Available: https://developer.apple.com/videos/play/wwdc2021/10094/

[8] "Chrome Design Doc: Trust Token API," 2021. [Online]. Available: https://docs.google.com/document/d/1TNnya6B8pyomDK2F1R9CL3dY10OAmqWlnCxsWyOBDVQ/edit?usp=sharing&usp=embed_facebook

[9] "CISCO - Consumer Privacy Survey," https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/cisco-cybersecurity-series-2021-cps.pdf, 2021.

[10] "Desktop Browser Market Share Worldwide," Oct. 2021. [Online]. Available: https://gs.statcounter.com/browser-market-share/desktop/worldwide/

[11] "NordVPN - How Am I Being Tracked," https://nordvpn.com/research-lab/tracked-down/, 2021.

[12] "State Partitioning - Privacy, permissions, and information security | MDN," 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Privacy/State_Partitioning

[13] "W3c - permissions policy explainer," 2021. [Online]. Available: https://github.com/w3c/webappsec-permissions-policy/blob/main/permissions-policy-explainer.md

[14] "Issue 1176287: Reconsider the choice of crypto for signing trust tokens, or document why we chose what we did," 2021-02-09. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=1176287

[15] "Access-Control-Max-Age - HTTP | MDN," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Max-Age

[16] "Automated Browser-Feature Testing for Uncovering Novel Tracking Vectors," 2022. [Online]. Available: https://anonymous.4open.science/r/ccs2022artifact-743C/

[17] "browsingData.remove() - Mozilla | MDN," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/browsingData/remove

[18] "Chrome - An updated timeline for Privacy Sandbox milestones," https://blog.google/products/chrome/updated-timeline-privacy-sandbox-milestones/amp/, 2022.

[19] "Chrome Web Storage and Quota Concepts," 2022. [Online]. Available: https://docs.google.com/document/d/19QemRTdIxYaJ4gkHYf2WWBNPbpuZQDNMpUVf8dQxj4U/edit#heading=h.uc5wcu4n4rnw

[20] "chrome.browsingData," 2022. [Online]. Available: https://developer.chrome.com/docs/extensions/reference/browsingData/

[21] "chromium/src - Commit r960512," Jan. 2022, publisher: Google. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/6241ea2c4875d1343594f3db53be489649335351

[22] "Clear-Site-Data - HTTP | MDN," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Clear-Site-Data

[23] "Express - Node.js web application framework," 2022, publisher: Express. [Online]. Available: https://expressjs.com/

[24] "Fenced Frames Ad Reporting," 2022, publisher: turtledove. [Online]. Available: https://github.com/WICG/turtledove/blob/main/Fenced_Frames_Ads_Reporting.md

[25] "FLEDGE API developer guide," 2022, publisher: turtledove. [Online]. Available: https://github.com/WICG/turtledove/blob/main/FLEDGE.md

[26] "Google - The Privacy Sandbox," https://privacysandbox.com/, 2022.

[27] "HTTP/3 protocol | Can I use... Support tables for HTML5, CSS3, etc," 2022. [Online]. Available: https://caniuse.com/http3

[28] "HTTP/3 protocol | Can I use... Support tables for HTML5, CSS3, etc," 2022. [Online]. Available: https://caniuse.com/?search=alt-svc

[29] "Performance - Web APIs | MDN," 2022, publisher: MDN. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance

[30] "Puppeteer | Tools for Web Developers," 2022. [Online]. Available: https://developers.google.com/web/tools/puppeteer

[31] "reCAPTCHA," 2022, publisher: Google. [Online]. Available: https://www.google.com/recaptcha/about/

[32] "Reporting API - Web APIs | MDN," 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Reporting_API

[33] "Third-party cookies and Firefox tracking protection | Firefox Help," 2022. [Online]. Available: https://support.mozilla.org/en-US/kb/third-party-cookies-firefox-tracking-protection

[34] "Welcome to PyAutoGUI's documentation! — PyAutoGUI documentation," 2022. [Online]. Available: https://pyautogui.readthedocs.io/en/latest/

[35] W. I. C. , "Trust Token API Explainer," 2020, publisher: WICG. [Online]. Available: https://github.com/WICG/trust-token-api

[36] 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*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 674–689. [Online]. Available: https://doi.org/10.1145/2660267.2660347

[37] ——, "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.

[38] M. D. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoofnagle, "Flash cookies and privacy ii: Now with html5 and etag respawning," *Available at SSRN*, 2011. [Online]. Available: http://dx.doi.org/10.2139/ssrn.1898390

[39] J. Bell, "File and Directory Entries API," Aug. 2021, publisher: W3C. [Online]. Available: https://wicg.github.io/entries-api/

[40] Brave Software, "Understanding Redirection-Based Tracking," Aug. 2018. [Online]. Available: https://brave.com/redirection-based-tracking/

[41] D. Creager and I. Clelland, "Network Error Logging," Jul. 2021, publisher: W3C. [Online]. Available: https://w3c.github.io/network-error-logging/

[42] D. Creager, I. Clelland, and M. West, "Reporting API," Apr. 2022, publisher: W3C. [Online]. Available: https://www.w3.org/TR/reporting-1/

[43] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda, "Privacy pass: Bypassing internet challenges anonymously." *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 3, pp. 164–180, 2018.

[44] Y. Dimova, G. Acar, L. Olejnik, W. Joosen, and T. Van Goethem, "The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion," in *Proceedings on Privacy Enhancing Technologies*. Proceedings on Privacy Enhancing Technologies, Mar. 2021, pp. 394–412, arXiv: 2102.09301. [Online]. Available: https://petsymposium.org/2021/files/papers/issue3/popets-2021-0053.pdf

[45] S. Dutton, "Getting started with Trust Tokens," 2020, publisher: web.dev. [Online]. Available: https://web.dev/trust-tokens/

[46] ——, "FLEDGE API," 2022, publisher: web.dev. [Online]. Available: https://developer.chrome.com/docs/privacy-sandbox/fledge/

[47] A. Edelstein, "Which browsers are best for privacy?" 2021. [Online]. Available: https://privacytests.org/

[48] 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.

[49] B. Fulgham, "Protecting Against HSTS Abuse," Mar. 2018. [Online]. Available: https://webkit.org/blog/8146/protecting-against-hsts-abuse/

[50] A. Ghedini, "Speeding up HTTPS and HTTP/3 negotiation with... DNS," 2020, publisher: Cloudflare. [Online]. Available: https://blog.cloudflare.com/speeding-up-https-and-http-3-negotiation-with-dns/

[51] E. Hammer-Lahav and M. Nottingham, "Defining Well-Known Uniform Resource Identifiers (URIs)," Internet Engineering Task Force, Request for Comments RFC 5785, Apr. 2010. [Online]. Available: https://datatracker.ietf.org/doc/rfc5785/

[52] I. Hickson, "Web SQL Database," Nov. 2010. [Online]. Available: https://www.w3.org/TR/webdatabase/

[53] C. Hothersall-Thomas, S. Maffeis, and C. Novakovic, "Browseraudit: automated testing of browser security features," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 37–47.

[54] S. Kamkar, "Evercookie- virtually irrevocable persistent cookies," Septemer 2010. [Online]. Available: http://samy.pl/evercookie/

[55] L. Knittel, C. Mainka, M. Niemietz, D. T. Noß, and J. Schwenk, "Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1771–1788. [Online]. Available: https://doi.org/10.1145/3460120.3484739

[56] 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. [Online]. Available: https://petsymposium.org/2020/files/papers/issue4/popets-2020-0077.pdf

[57] 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.

[58] M. Luo, P. Laperdrix, N. Honarmand, and N. Nikiforakis, "Time does not heal all wounds: A longitudinal analysis of security-mechanism support in mobile browsers," in *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*, 2019.

[59] M. Luo, O. Starov, N. Honarmand, and N. Nikiforakis, "Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 149–162.

[60] R. Marx, "HTTP/3: Practical Deployment Options (Part 3)," Sep. 2021. [Online]. Available: https://www.smashingmagazine.com/2021/09/http3-practical-deployment-options-part3/

[61] M. Nalpas, "Monitor your web application with the Reporting API," Oct. 2021. [Online]. Available: https://web.dev/reporting-api/

[62] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 541–555.

[63] M. Perry, E. Clark, S. Murdoch, and G. Koppen, "The Design and Implementation of the Tor Browser [DRAFT]," 2019. [Online]. Available: https://2019.www.torproject.org/projects/torbrowser/design/

[64] J. Schwenk, M. Niemietz, and C. Mainka, "Same-Origin Policy: Evaluation in Modern Browsers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 713–727.

[65] E. Skeggs, "Using CORS policies to implement CSRF protection - Mixmax Engineering Blog," 2017. [Online]. Available: https://www.mixmax.com/engineering/modern-csrf

[66] P. Snyder, S. Karami, B. Livshits, and H. Haddadi, "Pool-party: Exploiting browser resource pools as side-channels for web tracking," *CoRR*, vol. abs/2112.06324, 2021. [Online]. Available: https://arxiv.org/abs/2112.06324

[67] K. Solomos, J. Kristoff, C. Kanich, and J. Polakis, "Tales of Favicons and Caches: Persistent Tracking in Modern Browsers – NDSS Symposium," Virtual, Feb. 2021, pp. 1–19. [Online]. Available: https://dx.doi.org/10.14722/ndss.2021.24202

[68] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle, "Flash cookies and privacy," in *2010 AAAI Spring Symposium Series*, 2010. [Online]. Available: http://dx.doi.org/10.2139/ssrn.1446862

[69] M. Squarcina, M. Tempesta, L. Veronese, S. Calzavara, and M. Maffei, "Can i take your subdomain? exploring Same-Site attacks in the modern web," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2917–2934. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/squarcina

[70] P. Syverson and M. Traudt, "HSTS supports targeted surveillance," in *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/foci18/presentation/syverson

[71] T. Tiwari and A. Trachtenberg, "Alternative (ab)uses for HTTP alternative services," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/tiwari

[72] Vimeo, "Favicon History Sniffer: Chrome," 2022. [Online]. Available: https://vimeo.com/705259642

[73] ——, "Favicon History Sniffer: Safari," 2022. [Online]. Available: https://vimeo.com/705259659

[74] Wikipedia contributors, "Chromium (web browser) — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Chromium_(web_browser)&oldid=1084930496, 2022, [Online; accessed 28-April-2022].

[75] J. Wilander, "Full Third-Party Cookie Blocking and More," Mar. 2020. [Online]. Available: https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/

APPENDIX

In Table VI, we provide descriptions of the `write()` and `read()` actions for each evaluated mechanism. The descriptions offer further insight into the unique quirks of each approach, reframing accesses to these mechanisms as *reading* and *writing* methods.

Example implementation of a browser mechanism in CanITrack. We use the Image Cache as an example browser mechanism to illustrate the testing process using CanITrack. This example also provides an estimate of the workload required to add a new browser mechanism for testing with CanITrack.

When a webpage makes a request for an image resource, the browser caches the returned image based on the `Cache-Control` header included in the response object. This mechanism requires three components in order to be tested using CanITrack.

1) *File Resource.* The user provides an image file as a resource, whose path is accessible to the *Web Server*.
2) *Network Requests.* The user handles two paths relevant to the browser mechanism on the *Web Server*:
   - *Image Request.* To be responded with the image file. If the request is received from the `write()` method, the response should include a `Cache-Control` header to ensure that the browser stores the image. If the request is received from the `read()` method, a global object should be updated to record the number of requests that were seen for the requesting domain.
   - *Number of Accesses.* To be responded with the number of times a request for the image was received from each requested domain.
3) *`write()` and `read()` methods.* These JavaScript methods will be called by various test scripts on the client-side.
   - `write()`: This method receives a unique, 32-bit identifier, and a list of domains. For every bit of the identifier that is equal to '1', it requests an image from the corresponding domain.

```
    // Image Request
    if (request.url.includes("/image")) {
      if (testPhase == 'write') {
        response.set('Cache-Control', 'max-age
            =31536000');}
      if (testPhase == 'read') {
        imageAccesses[req.headers.host]+=1;}
      response.sendFile('/path/to/image');}
    // Number of Accesses
    if (request.url.includes("/accesses")) {
      response.send(imageAccesses[request.headers.
          host]);}
```

Listing 5: *Sample Network Request Handling for Image Cache.*

- `read()`: This method receives a list of domains as an input. It requests an image from each of the 32 domains. It then requests the server to respond with the number of requests that had been sent over the network for each domain. If the image for a domain was served from the cache, the server wouldn't have observed any request for the corresponding domain (i.e., bit 1). All other domains correspond to bit 0 values.

```
write (uniqueID, domainList) {
  for (let i = 0; i < domainList.length; i++) {
    if(uniqueID[i] == '1') {
        let image = document.createElement("img");
        image.src = `https://${domainList[i]}/
            image`;
        document.body.appendChild(image);}}}

read (domainList) {
  let uniqueID = '';
  for (let i = 0; i < domainList.length; i++) {
    // Request the image
    let image = document.createElement("img");
    image.src = `https://${domainList[i]}/image`;
    document.body.appendChild(image);
    // Check if image was fetched from the server
    let response = await fetch(`https://${
        domainList[i]}/accesses`);
    uniqueID += (await response.text()).trim();}

  return uniqueID;
}
```

Listing 6: *Sample Read and Write Methods for Image Cache.*

CanITrack's framework then places and invokes the `write()` and `read()` mechanisms in different first- and third-party contexts. Since those two methods interact with the mechanism-specific server-side requests and file resource, the additional workload for the user includes handling those requests and providing the resource.

TABLE VI: Overview of the caching mechanisms evaluated by CanITrack. Novel tracking vectors are indicated by ✚.

API	Write Mechanism	Read Mechanism	Bits/Page	Notes
Cookies	Write the identifier using the `document.cookie` API	Look at entries in the `document.cookie` API	32	
Storage API:				
local Storage	Call `localStorage.setItem()`	Call `localStorage.getItem()`	32	
indexed DB	Create a new Database and Object Store. Write the identifier to the Object Store.	Read the identifier from the same Object Store.	32	
Cache Storage	Add identifier to URL path and cache the request using the client-side API.	Access entries from the cache API.	32	
File-based Mechanisms:				
CSS Cache	Create a new link element for a stylesheet. The server responds with a 'Cache-Control' header.	Request stylesheets from the server. Requests not observed at the server were served from the cache.	32	
Font Cache	Add a new webfont and apply it to an HTML element. The server responds with a 'Cache-Control' header.	Apply the same webfont to an HTML element. The server observes requests for fonts that were not served from the cache.	32	
Image Cache	Create a new img element, set the `src`, and add it to the DOM. The server responds with a 'Cache-Control' header.	Create a new img element, set the same `src`, and add it to the DOM. The server observes requests for images that were not served from the cache.	32	
HTTP Disk Cache	Create a new Fetch or XHR request for any resource. The server responds with a 'Cache-Control' header.	Send the same Fetch or XHR request. The server observes requests that were not served from the cache.	32	
Favicon Cache	The browser requests a favicon based on the link element. The server responds with a 'Cache-Control' header.	Revisit the same page. The server observes a request for favicons that were not served from the cache.	1	[67]
Service Workers:				
Variable Scope	Register a new service worker with is scope set to '/', set a value in a variable.	Access the service worker and read back the value stored in the previously defined variable.	32	
Cache	Register a new service worker with is scope set to '/'. Add a cache entry that requests the server for a 32-bit identifier.	Create a fetch request that is intercepted by the service worker, and the identifier stored in the cache entry is returned.	32	
HTTP Headers and Network Config-based Mechanisms:				
Alt-Svc	Send a network request for a domain. The server includes an Alt-Svc header in each response, indicating the availability of HTTP/2 and HTTP/3 services.	Create the same network request. HTTP/2 or HTTP/3 requests observed at the server indicate entries in the Alt-Svc cache.	32	[71]
HSTS	Send a new HTTP request. The server ugrades the requests to HTTPS.	Resend the same HTTP requests. The browser upgrades the request to HTTPS.	32	[70]
HTTP Auth	Send a network request including credentials in the 'Authorization' header.	Send the same requests without providing credentials. The browser adds the 'Authorization' header from cache.	32	
CORS Preflight	Send a cross-origin request. The browser sends a preflight request before the actual request.	Send the same cross-origin request. The browser observes an existing preflight in the cache and does not send an OPTIONS request.	32	✚
Accept-CH	The server includes client hint values in the 'Accept-CH' response header.	The server observes client hints added by the browser on future requests.	5	✚
NEL	The server includes a reporting URL in the 'NEL' and 'Report-To' Headers.	The server receives reports from the browser at the reporting URL.	32	
Chromium-specific Mechanisms:				
File System	Call the 'window.webkitRequestFileSystem'. Create a new Directory and store the identifier in a new file.	Call the 'window.webkitRequestFileSystem'. Access the previously created directory and file. Return the identifier.	32	
WebSQL	Access the DB using 'window.openDatabase'. Use SQL commands to create a new table and insert the identifier into a row.	Access the DB and run a 'SELECT' query to read the identifier.	32	
FLEDGE API	Add the browser to an interest group, whose owner is a specific domain.	Run an auction, with a single domain as the buyer. If the auction is successful, the browser has an interest group belonging to the domain.	8	✚
Trust Tokens	Create a new fetch request, and issue a token from a domain.	Call the 'document.hasTrustToken()' API to check if a token exists from the domain.	2	✚