Taint Analysis for Browser Fingerprinting

Jane Ahn, Will Dey, Eric Zhang
Harvard University
{jane_ahn, will_dey, ekzhang}@college.harvard.edu

December 11, 2020

Abstract

Browser fingerprinting is a major security concern on the modern web. It allows remote servers to collect information about devices and identify them across website boundaries. We discuss how dynamic taint analysis can be used to mitigate common fingerprinting methods. We also build extensible software to explore various taint tracking approaches for identifying and preventing browser fingerprinting. In particular, we implemented a Chrome extension that blocks all connections from a tab that has previously accessed sensitive data and demonstrate how such preventative measures can be embedded into the browser kernel via a simple code modification.

1 Introduction

Modern browsers have to balance complex functionality and privacy requirements to serve users. Unfortunately, much of the functionality added in recent years yields a large surface area for fingerprinting and cross-site tracking, which compromises user privacy. The Electric Frontier Foundation's *Panopticlick* provides a concrete list of common ways to fingerprint a browser through modern web technologies, a combination of which can even generate unique identifiers for most users [2]. In 2010, Peter Eckersley from the Electronic Frontier Foundation collected 470,161 fingerprints using data from HTTP headers, JavaScript, and plugins and showed 83.6% of them were unique [10]. Unique fingerprints can be used to track activity across multiple sessions without user consent or knowledge [20], which can then be used for advertising or malicious purposes.

Recently, there has been a large movement to protect users from trackers that use browser fingerprinting, but no solution is complete yet. Preventing browser fingerprinting while preserving the website functionality has proven to be a difficult challenge. The goal of this project is to develop potential countermeasures to browser fingerprinting while leaving a user's experience on the web unchanged to the greatest extent possible.

2 Background

Three common approaches used to defend against browser fingerprinting today are blocking the execution of fingerprinting scripts, breaking the stability of fingerprints, and breaking the uniqueness of fingerprints [27]. We discuss each of them briefly below.

Blocking JavaScript execution prevents servers from collecting data accessible through DOM APIs, greatly limiting trackers. However, this approach also blocks scripts that are necessary to display a website and load dynamic content, so it interferes with user experience. Although some implementations let users select "trustworthy" sites that would allow scripts to run [23], this approach is generally prone to overblocking.

Therefore, a common alternative way to block fingerprinting scripts is to selectively block network requests (`fetch`, `XMLHttpRequest`) matching certain filter lists. This prevents trackers in the filter lists from collecting sensitive data about a user's browser. One of the most commonly used lists is provided by *EasyPrivacy*, which aims to remove all bugs, tracking systems, and information collectors [11]. However, such lists are not complete, and new tracking systems are developed continuously.

Some approaches defend against fingerprinting by heuristically learning which sources to block. The EFF's *Privacy Badger* keeps track of third-party domains that embed elements into the websites a user visits and disallows content from a remote domain once it tries to track the user's browser on three different websites [12]. However, *Privacy Badger* can often be too aggressive in blocking, preventing necessary elements of a web page from functioning [20].

Another approach to defend against browser fingerprinting is breaking the stability of fingerprints.

This solution frequently modifies a device's fingerprints in a way that a device's old fingerprints cannot be linked to new fingerprints. While fingerprints can still remain unique, third parties will have more difficulty tracking a device employing this defense for an extended period of time. Initially, there was concern that such modified fingerprints were 'unnatural' and thus could be used for additional identification purposes [27], but attempts have since evolved. For example, *PriVaricator* uses randomization of the font and plugin related properties to make it difficult for a third party to link fingerprints of the same user, but guarantees the resulting fingerprints do not stand out in any way [25]. However, these approaches tend to have incomplete coverage since trackers can still use other identifying sources of information for browser tracking.

The third method for defending against browser fingerprinting is breaking the uniqueness of fingerprints. Trackers cannot use a browser's fingerprint to track devices if fingerprints are not unique. Hence, some countermeasures for browser fingerprinting aim to issue non-unique fingerprints for all devices. For example, Tor aims to issue identical fingerprints for all of its users [9]. However, Tor still does not completely block fingerprinting since the Tor fingerprint is brittle, and the fact that Tor is being used can still be seen as a piece of identifying information [20].

Therefore, our goal in this project is to present and analyze several approaches for preventing browser fingerprinting in a recent version of Chromium, focusing on both browser-specific methods like extension blocking, as well as inter-process communication (IPC) between the content-handling processes and the browser kernel. This allows us to record which sensitive information makes its way into web requests and potentially even block requests that contain such sensitive information.

3 DTA Blocking

Our first approach is based on dynamic taint analysis (DTA). We use the `libdft` tool for dynamic taint tracking through a binary [19]. Taint sinks can be marked in inter-process communication or other critical paths in the JavaScript engine. We take advantage of the architecture of web browsers in isolating render processes from the kernel, which is responsible for network requests.

For a starter list of tainted user data, we will use the Electronic Frontier Foundation's *Cover Your Tracks* list, which includes identifiers like system language, screen resolution, available fonts, and various browser-specific behaviors. By tracking taint through the system, we will be able to produce a thorough list of mechanisms through which malicious web servers can glean private information, which may even include attacks that have not yet been revealed by less rigorous methods.

3.1 Dynamic Taint Analysis

Dynamic taint analysis is implemented through the `libdft` tool, which was introduced in [19]. At a high level, this tool allows you to attach hooks to certain procedures (system calls and functions) that mark taint. Then, an unmodified binary is instrumented through Intel's *Pin* tool, which allows the dynamic taint analyzer to track propagation at the binary instruction level.

The original `libdft` tool was built in 2012 and only works for 32-bit x86 binaries. However, since the vast majority of modern Intel processors are 64-bit, we instead based our work on a modified version called `libdft64` that is maintained and developed for the *Angora Fuzzer* [4]. This allows us to dynamically instrument x86-64 binaries using the latest version of Intel Pin 3.7.

As a simple example, we use `libdft64` to implement a Pintool that attaches taint value 42 to the argument of a C-like function `set_taint(void *p)` by writing the following C++ code:

```
VOID SetHandler(void *p) {
  tag_t t = tag_alloc<tag_t>(42);
  tagmap_setb((ADDRINT)p, t);
  printf("addr: %p", p);
}

/* excerpt from PIN_Init(...) handler */
RTN rtn = RTN_FindByName(img, "_set_taint");
if (RTN_Valid(rtn)) {
  RTN_Open(rtn);
  RTN_InsertCall(
    rtn, IPOINT_BEFORE, (AFUNPTR)SetHandler,
    IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
    IARG_END,
  );
  RTN_Close(rtn);
}
```

Similar methods can be used to attach taint-tracking instrumentation to Linux system calls, such as `open`, `socket`, and `read`, to mark certain inter-process communication modes as a taint source or taint sink. We can also use these hooks to read propagated taint tags. When sensitive taint reaches a sink, the tool can then log or block sensitive requests while running the binary.

The primary advantage of Pintool-based dynamic taint analysis over systems that modify the browser kernel is that Pintools do not require the user to download a modified, unsigned resesarch prototype binary. Instead, they can be dynamically attached to instrument running processes on the computer, which allows patching of a running version of Chrome without installing a separate forked binary. This also has benefits for security researchers, since cloning and compiling the monolithic Chrome kernel requires Google's vendor-specific `depot_tools` library, over 17 GB of code and dependencies, a modern C++ toolchain, and over 20 CPU-hours of build time.[1] This is inconvenient for short-term patches.

In comparison, the requirements to run a Pintool are simply to install the code of Intel Pin 3.7 from a publicly hosted `.tar.gz` file, then attach the Pintool to running instances of Chrome's render processes whenever they are available. This avoids having to change the application binary. Some Pintools may not run on different platforms (IA-32, x86-64), but the convenience of installation makes them suitable for research and security testing on a wide variety of machines, if not for wide distribution.

3.2 Design of a Selective Blocker

In this section, we discuss the theoretical design of a dynamic taint analyzer for selective blocking of requests containing sensitive information. We also briefly outline some of the practical challenges of building such a taint analyzer, due to implementation specifics of the monolithic Chrome browser.

In general, we can selectively block requests by conducting a thorough taint analysis of the binary memory space. Formally, call data at a memory address *tainted* if it is at least partially dependent on some sensitive data we would like to protect from fingerprinting, such as the browser's `User-Agent` string, or the details of its `<canvas>` rendering context implementation (which depends on the system graphics drivers). Other potential taint sources are listed in Fig. 1. The goal of a selective blocker is to limit the potential spread of tainted data across the network to third-party servers who can then use that data to perform cross-site tracking of users.

Our proposed design for a taint-based blocker combines three primary components: sources, sinks, and additional script scaffolding.

[1]Furthermore, if you accelerate builds with a tool like ccache, it may require hundreds of GB of free disk space.

3.2.1 Taint Sources

We attach hooks to Chrome's inter-process communication handles that inform the render process of sensitive user details. Because of the flexibility of `libdft64`, it suffices to determine the symbol names of critical paths that are responsible for obtaining data like time zones, graphics drivers, and other sensitive information. This does not have to be entirely reverse-engineered, since the Chromium source code and toolchain are both open source, which makes finding the proper symbols a matter of carefully reading through the source code. Once the taint source is found (either as a function or system call), their arguments can be accessed by using the `IARG_FUNCARG_ENTRYPOINT_VALUE` and `X64_ARG#_REG` descriptors, passed as variadic arguments to the `RTN_InsertCall(...)` API.

3.2.2 Taint Sinks

When tainted information is introduced to the render process, we request `libdft` to add a taint tag to a specific memory address `p` using the function `tagmap_setb(ADDRINT p, tag_t t)`. Then, Intel Pin is used to track the propagation of this taint, based on a binary decision diagram (BDD) data structure, while the program is executing.

Since we add this taint information, we also need a hooks on functions that might *leak* sensitive tainted data. These are known as taint sinks, and they work in a similar way using the `RTN_InsertCall(...)` and `syscall_set_pre(...)` APIs for function and syscall instrumentation, respectively. We can intercept the data passed into these taint sinks and analyze their propagated taint directly by calling the `tagmap_getn(ADDRINT p)` function, which returns a list of tag structures. If the data argument passed to one of many taint sinks such as `XMLHttpRequest::open(&method, &url)` defined in xml_http_request.cc is tainted, then the request is intercepted and logged.

3.2.3 Scaffolding

Scaffolding is a script that listens for when the browser spawns render processes, detects the type of process based on reverse-engineering hints, and attaches the correct Pintool to that process before it starts running. This could be implemented in various ways, such as with the `ptrace(2)` utility in Linux.

Taint Source	**Description**	**Entropy (bits)**
User Agent	String sent in every HTTP request with browser vendor and version.	4–10
HTTP_ACCEPT headers	String sent in every HTTP request with a list of all the content types the browser understands.	9–14
Browser Plugin list	JavaScript-requested list of all native plugins loaded in the browser. Such plugins are being phased out, so the list is often simply `undefined`, but an older browser can potentially reveal a wealth of unique information through this list.	1–14
Time Zone	JavaScript-requested string with the standard human-readable time zone.	4–5
Time Zone Offset	String sent in HTTP requests with the numerical offset from UTC. Similar to Time Zone, but looking at discrepancies between the two can reveal more information of Daylight Savings time peculiarities.	4–5
Screen Size and Color Depth	JavaScript-accessed variables that reveal the exact size of the browser window, as well as the range of colors it is able to display on the current monitor	10–13
Installed Fonts	JavaScript-accessed information about which fonts are installed on the system, which is generally very correlated with the particular operating system the user is running	5–8
Accepting Cookies	Whether or not the browser accepts cookies, which can be revealed through the behavior of HTTP requests across site visits.	1
Supercookies Enabled	JavaScript-accessed stores of data larger than cookies that does not need to be sent in every request. Examples include LocalStore or Internet Explorer's userData.	1–6
HTML5 Canvas Hash	Hash of extracted canvas data calculated by JavaScript that will change with the user's operating system, browser version, graphics card, firmware version, graphics driver version, and fonts.	14–22
WebGL Hash	Similar to the Canvas hash, but for extracting information about how the user's browser renders GPU-accelerated content.	9–22
WebGL Vendor	JavaScript-requested string with the graphics card vendor powering the browser's WebGL implementation.	7-15
CPU Class	JavaScript-requested string on the CPU manufacturer.	1-4
Memory	JavaScript-requested measure of estimated available memory on the system.	4–8

Figure 1: Tainted browser characteristics and their approximate amounts of entropy [3]. In this case, lower entropy indicates that a data source contributes less identifying information that could be used to uniquely track a specific user across the web.

3.3 Challenges

Although dynamic taint analysis is appealing because of its granularity and simplicity, there are several challenges to its implementation. First, we found that properly attaching the Pintools to Chrome's render processes was very difficult, as the browser could be very fickle about when it forks and joins child processes. Although we were able to run Pin on short proof-of-concept programs written in C++ to analyze taint, it was very difficult to practically attach a Pintool to the render process.

The other practical issue was of deployment. For a non-technical audience, it is very difficult to install the software necessary to run Pintools, and running such dynamic instrumentation on binaries empirically slows down their execution by several orders of magnitude. This makes dynamic taint analysis mostly useful for theory or security research, rather than as a widespread preventative measure. Therefore, because of the complexity in working with Chrome's process architecture to develop the scaffolding necessary to attach Pintools, and diminishing returns from deployment, we leave the implementation of dynamic taint analysis to future work.

4 Chrome Extension Blocking

An alternative approach to taint analysis via Pintools is to use Chrome's extension API, which provides simple ways of extending Chrome's functionality through short JavaScript plugins. Chrome extensions are very easy to install and develop, requiring no separate toolchain or build step. They have been used in past security research as a proof-of-concept to prevent other attacks such as XSS [13]. In the browser fingerprinting literature, the *Privacy Badger* extension is also implemented solely as a browser extension, enabling millions of non-technical users to easily install it [24].

As a proof-of-concept software, we introduce the *TaintBlock* extension, which prevents web pages from making web requests that have the potential to leak sensitive information. Our security model is based on a state machine, where pages that access sensitive user data are placed in a sandboxed environment that disables further network requests. *TaintBlock* uses the `getCurrrentPosition` method from the `window.navigator` JavaScript object as its example of a sensitive function. We first present a brief outline of the anatomy of our Chrome extension in Section 4.1, then discuss our threat model and security guarantees in Section 4.2.

4.1 Extension Architecture

The current version of *TaintBlock* consists of four files: `manifest.json`, `background.js`, `content.js`, and `script.js`.

- `manifest.json`: Every Chrome extension requires a `manifest.json` file that includes important information about the extension, such as its name/version/description, permissions it requires, and the scripts it runs [8].

  TaintBlock requires Chrome permissions for `http://*/*`, `https://*/*`, `tabs`, `webRequest`, and `webRequestBlocking`. The first two allow us to run the extension on web pages with urls starting with `http://` or `https://`. The `tabs` permission gives us access to tab IDs and events regarding tab updates. Finally, `webRequest` and `webRequestBlocking` allow us to monitor and block web requests made by web pages.

- `background.js`: Background scripts allow extensions to monitor and respond to events that occur during a user's browsing experience [7].

  Our `background.js` maintains `blockedList`, a set of tab IDs that should be blocked because they have accessed sensitive data. The background script adds a tab to `blockedList` when it receives a message saying that a tab has called the `getCurrentPosition` method from the content script. It removes a tab from `blockedList` when the tab is removed, replaced, or updated with a new URL from the address bar. When `background.js` learns about a network request, it blocks the request if the associated tab is in `blockedList`.

- `content.js`: Content scripts run in the context of web pages and are able to read, modify, and send details of the web pages the browser visits. They are executed in isolation, which allows them to change the JavaScript environment without creating conflicts with the page or other content scripts [6].

  Our `content.js` is notified by `script.js` when a tab has called a sensitive method, such as `getCurrentPosition`. It then relays this message to `background.js`, implying that network requests by the tab should subsequently be blocked. The actual process of identifying which tabs access sensitive methods is executed in `script.js`. Because content scripts are isolated, `content.js` injects `script.js` as a DOM resource to expose our modified versions of sensitive functions so that the web page will execute our version of sensitive function calls. This allows us to correctly identify tabs that have accessed sensitive data and whose network requests should be blocked.

- `script.js`: Our `script.js` is an unprivileged, DOM-injected script that directly patches the `getCurrentPosition` method on the browser's `Navigator` object in JavaScript. The monkey-patched method sends a custom event message to `content.js` notifying it was called by some process, so that `background.js` can block all future network requests made by the tab accordingly.

In summary, we design our web extension around the security principles of Chrome's framework, by having a background script that blocks all connections (including links, `<iframe>`s, `<img>`s, and `XMLHttpRequests`), a content script which injects unprivileged JavaScript code at initial page load, and patched handlers for sensitive APIs. These are all tied together by standard Chrome runtime communication channels.

4.2 Navigator-Based Security

TaintBlock is implemented using a *state machine* model. When a tab is first opened, it starts in the initial state without any restrictions. The tab remains in the initial state unless it accesses any sensitive data, in this case the `getCurrerntPosition` method, at which point it transitions into the tainted state. A tab in the tainted state is blocked from making any more network requests. A tab returns to the initial state when it gets updated with a new url, e.g. via a link click or the user entering a new url in the tab directly. Otherwise, a tab remains in the tainted state until it is closed or replaced by another tab. Note that with the current implementation of *TaintBlock*, a url change on a tainted tab initially leads to a page that informs the user the access was blocked by *TaintBlock*, but the user can choose to load the new page simply by clicking reload. While this may be slightly annoying to the user, it also provides an extra layer of security in the case malicious web pages try to simulate link clicks on a tainted tab.

Using the state machine model allows *TaintBlock* to block web requests issued by web pages that have accessed sensitive information. This prevents identifiable data from being sent to tracking sites, shielding the device from browser fingerprinting. However, note that we still allow web pages to obtain the sensitive information. By blocking network requests but still allowing web pages to acquire such information, we are able to maintain the functionality of benign web pages that require such information, like online maps in our current version of *TaintBlock*.

The example we used for the sensitive method analyzed by *TaintBlock* is the `getCurrentPosition` method of `navigator.geolocation`. More generally, the `Navigator` interface holds important identifiable information about the state and identity of the user agent, allowing scripts to query this information [5]. For example, `navigator.clipboard` can be used to access the system clipboard and read its contents. Users often need to give special permissions for websites to access this data. Hence, it is imperative that browsers are protected against malicious servers that collect identifiable data for tracking reasons.

Aside from the security guarantees, a benefit of *TaintBlock* is that it is portable and easy to use. According to Statcounter, Chrome is the most commonly used browser with Chrome holding 63.5% of the browser market share worldwide [26]. As long as a user has access to a Chrome browser, the only setup required on the user side is loading our code onto their Chrome extensions. *TaintBlock* will work automatically once it is installed without any additional user intervention.

5 Chromium Fork

The Chrome extension could in practice be bypassed by a dedicated attacker who designs page JavaScript to undo *TaintBlock*'s changes. Hence, we explore options to modify browser architectures to be taint-aware, so that sensitive information is contained at a more core level than an extension. We take advantage of the fact that all major modern browsers use a multi-process architecture [21], where pages with different origins are isolated in different render processes, all managed by the privileged browser kernel. Fig. 2 describes how the browser kernel itself could be modified in a way to track and contain taint to within render processes.

An example of a "benign data read" mentioned in Fig. 2 a file upload that the user has consented to via a file dialog, since the user has control over what the page can access and can avoid uploading any sensitive information. A "sensitive data read" is any renderer request to the kernel that reveals a row in Fig. 1, which can be used to surreptitiously track a user across the web.

We mimic the behavior of *TaintBlock* by adding in the new `tainted_renderers` table, located in the `resource_dispatcher.cc` file, to track which render processes have been tainted by accessing sensitive information. It is important to note that our ar-

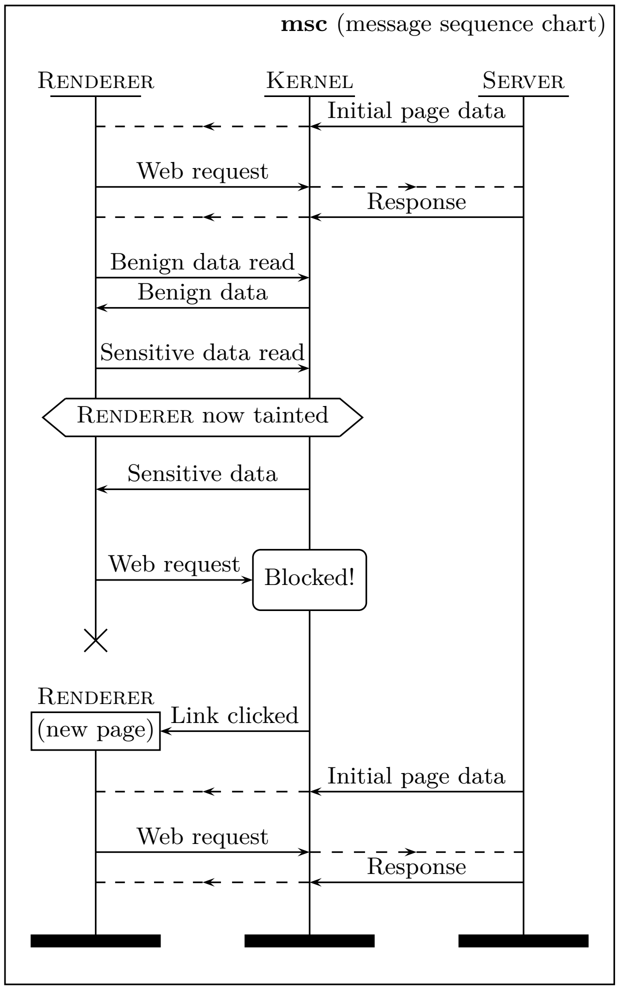


Figure 2: Browser kernel-enforced taint containment.

chitecture does not block the sensitive read request, since many tracking methods abuse widely deployed JavaScript features, and outright blocking such data accesses would break much of the modern web. Instead, once an access has been made, the renderer becomes marked as tainted, and can no longer make requests to external servers. From the point of view of the web page, the user will appear to have lost internet connection after it makes a sensitive data request, which is a situation that is far more common and more gracefully handled than the unavailability of all sensitive JavaScript methods. From the point of view of the server who is potentially watching the requests a particular user is making in an effort to extract identifying information, the user will again appear to go offline sometime in the middle of exchanging requests, so no sensitive data is leaked and it is not possible to launch an information extraction attack on the modified architecture itself.

We have implemented tainting at the browser kernel level for sensitive data reads which always make a kernel request, such as GPU details and geolocation, in a fork of the Chromium browser hosted at `https://github.com/wi11dey/chromium`. There are some fingerprinting channels like screen resolution that are cached in each renderer before JavaScript accesses it, but one can envision further modifications to the JavaScript engine in future work that marks a renderer as tainted if any of the cached sensitive data has ever been accessed.

6 Results

We empirically and quantitatively evaluate the performance of each method we study.

6.1 Dynamic Taint Analysis

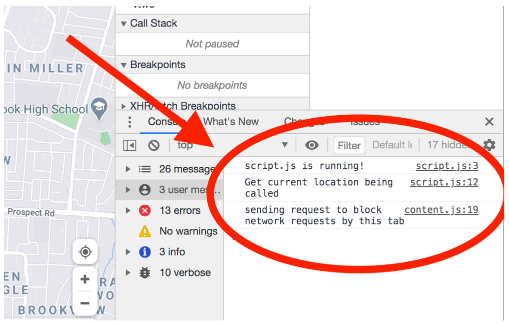
As mentioned in Section 3, we were not able to implement our design for a dynamic taint analyzer in Chromium. However, we can still qualitatively evaluate the performance of our proposed design. In accuracy and granularity, it is likely without question to be the most failure-proof method, as it enforces taint propagation at the binary level. It can also maintain the least loss of user functionality after a page requests identifiable data.

However, dynamic taint analysis has issues with performance, reliability, and portability, so it is not practical to deploy on a large scale to general users. This is the primary downside of our proposed design, but it can still be useful for security researchers who are looking to detect, with proof, websites that use fingerprints in cross-site tracking scripts.

6.2 Chrome Extension

We began by testing *TaintBlock* on websites known to use Navigator Geolocation. Fig. 3 shows that *TaintBlock* immediately blocks network requests coming from the tab with Google Maps, as expected. As Google Maps use the device's location for better accuracy and user experience, we see that most of the functionality of the map is maintained (including search and navigation of the map), but network requests become blocked.

We also noticed *TaintBlock* blocks network requests made by all Google search result pages. Hence, a user must open a search result in a new tab, or open it in the same tab, but then refresh the page to view the contents. While *TaintBlock* protects the



```
script.js is running!                  script.js:3
Get current location being             script.js:12
called
sending request to block               content.js:19
network requests by this tab
```

Figure 3: Loading Google Maps with *TaintBlock*.

user from potentially being fingerprinted by geolocation data, we acknowledge that this particular feature of the extension may be unwanted by many users.

Finally, we tested *TaintBlock* against five websites[2] known to call the `getCurrentPosition` method, as supplied by [22]. We verified that each of these sites called the target navigator API, and *TaintBlock* successfully blocked network requests coming from tabs after those web pages called `getCurrentPosition`.

These promising results indicate that *TaintBlock* is simple, modular, customizable, and easily extensible in the future to create more sophisticated and complete analyses of user information flow.

6.3 Patching Chromium

We have demonstrated how to block two classes of leaks in Fig. 1: leaks about details of the GPU vendor and geolocation, with strong guarantees on the security of the system as page JavaScript cannot alter browser kernel behavior. Overall, we summarize the privacy and usability requirements we were able to achieve with each method and compare their advantages in Fig. 4.

7 Limitations

When working with monolithic applications such as the Chrome web browser, it is difficult to directly modify code and distribute patched binaries. However, built-in extension mechanisms often do not offer the freedom to perform complex analysis due to security reasons. At the same time, tools like Intel Pin can be unportable to different system architectures and tricky for end users to install. Therefore, privacy on the web is an unsolved problem, and each approach has its advantages and trade-offs.

We firmly believe that users should care deeply about their privacy on the web. However, browser fingerprinting is a serious issue that threatens to make obsolete same-origin policies for protecting users' sensitive information and preventing it from leaving a site. Therefore, any approach that motions toward solving this problem or offering more tools to analyze privacy threats is extremely important for users.

One major limitation of all of our proposed privacy measures is the issue of persistent storage. By using persistent browser storage APIs such as Web Storage (also widely known by its identifier `window.localStorage`) [15, 16], IndexedDB [1], and WebSQL [14], an attacker can circumvent taint-based blocking approaches with the following attack:

1. Obtain sensitive user data by using web APIs or other methods, such as canvas fingerprinting. For example, a malicious website could call `geolocation.getCurrentPosition()` to obtain the user's current location.

2. Save the results of the geolocation query to a serializable JavaScript object, which is tainted and cannot be sent in a network request. However, they can place it in persistent storage with `Storage.setItem(key, value)`.

3. On the next visit of the user to the site, all taint tracking has been refreshed. The website can then call `Storage.getItem(key)` to obtain the sensitive location data, which **no longer has taint**, then freely send that data in a network `POST` request to a fingerprinting or analytics server.

Therefore, a knowledgeable attacker could implement this approach to work around our defense and obtain sensitive data. Another issue with the browser extension approach is that page links cannot be trusted after a single sensitive info request. For example, a user looking at Google search results with location being used has their links on the page blocked, despite this being benign functionality. Unfortunately, this is a necessary tradeoff for security, as otherwise an attacker could circumvent the taint protections with JavaScript code that uses anchor elements to circumvent taint protections, such as:

```
let payload = encodeURIComponent(data);
let url = `https://evil.com/?q=${payload}`;
let anchor = document.createElement("a");
anchor.setAttribute("href", url);
anchor.click();
```

The problem with the above code is that it is indistinguishable from a user clicking on an actually benign

[2]These were sportclips.com, skechers.com, viamichelin.com, storelocator.samsonite.com, and salvage-parts.com.

	Prior work		Introduced in this paper		
Metric	**Filter List** [17]	**Learning** [18]	`libdft` **DTA**	***TaintBlock***	**Browser fork**
Overhead	Minimal	3+ requests	Large	None	None
Maintenance	Daily	Continuous usage	None	None	None
Granularity	Domain	Domain	Binary-level	Renderer	Renderer
Geolocation	Disabled	Disabled	N/A	Contained	Contained
GPU hash	Per domain	Learned domains	N/A	Contained	Contained
Coverage	Known in list	Repeat offenders	Entire class	Entire class	Entire class
Site Impact	Moderate	Moderate	Minimal	Unclear	Unclear
Bypassable	Depends on list	Yes [18]	No	Carefully	No

Figure 4: Evaluation of various anti-tracking mechanisms in browsers alongside our methods.

link, but it allows an attacker to leak an arbitrary string `data` in a GET request to the server behind the `evil.com` domain, even if other network requests are blocked. Unfortunately, with such a coarse-grained blocking strategy as available to Chrome extensions, this loss of functionality is unavoidable.

Low specificity is an unfortunate but necessary tradeoff for the convenience of implementing taint tracking in a browser extension such as *TaintBlock*. As mentioned before, the dynamic taint tracking abilities of a library like `libdft64` come with performance and portability penalties, so there is currently no single solution to this problem.

8 Conclusion

We have introduced several approaches for defending against browser fingerprinting, including taint-based blocking, extension-based blocking, and Chromium-based blocking. Our theoretical design for dynamic taint analysis-based tracking is implemented through `libdft64`, preventing functions from leaking sensitive tainted data. *TaintBlock* is a simple extension to block web pages from leaking the browser's geolocation data, and it can be extended to protect other sensitive data. Finally, we are able to directly modify Chromium's source code to add a table tracking tainted state across various processes.

As emphasized in previous sections, our solutions are not yet complete. Below, we outline several directions we can expand our work in the future:

- Implement taint analysis using libdft64: As mentioned in Section 3, we leave the tricky implementation details of our proposed dynamic taint analysis-based blocker for future work.

- Defend against variable access in *TaintBlock*: Many aspects of the DOM used for fingerprinting are defined as properties rather than methods, such as cookies and sizes of screens and windows. Identifying web pages that access these sensitive data will help build our defense against browser fingerprinting. We can do this by transparently shimming JavaScript property getters.

- Support more granular blocking in *TaintBlock*: The current structure of *TaintBlock* blocks all network requests made by a tab once the page on the tab accesses sensitive information. We can instead block specific frames on a tab to improve user experience.

- Extend the Chromium fork to cover more classes of leaks from Fig. 1, as well as cover cases that *TaintBlock* is unable to, like sensitive variable access.

Taint analysis is a natural approach for defending against the problem of cross-site browser tracking. Our study helps understand the challenges and limitations of defenses against browser fingerprinting, which is a privacy concern that will continue to be salient in the near future.

References

[1] Ali Alabbas and Joshua Bell. Indexed database API 2.0. W3C recommendation, W3C, January 2018. `https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/`.

[2] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. User tracking on the web via cross-browser fingerprinting. In *Nordic conference on secure it systems*, pages 31–46. Springer, 2011.

[3] Bill Budington. Panopticlick: Fingerprinting your web presence. In *Enigma*, San Francisco, CA, January 2016. USENIX Association.

[4] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[5] MDN contributors. Navigator. https://developer.mozilla.org/en-US/docs/Web/API/Navigator.

[6] Chrome Developers. Content scripts. https://developer.chrome.com/docs/extensions/mv2/content_scripts/.

[7] Chrome Developers. Manage events with background scripts. https://developer.chrome.com/docs/extensions/mv2/background_pages/.

[8] Chrome Developers. Manifest file format. https://developer.chrome.com/docs/extensions/mv2/manifest/.

[9] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 1–21. USENIX Association, 2004.

[10] Peter Eckersley. How unique is your web browser? In *Proceedings of the 2010 Privacy Enhancing Technologies Symposium*, pages 1–18, 2010.

[11] Fanboy, MonztA, Famlam, and Khrin. Easyprivacy. https://easylist.to/index.html.

[12] Electronic Frontier Foundation. Privacy badger. https://privacybadger.org/.

[13] Shashank Gupta and Brij Bhooshan Gupta. XSS-immune: a Google chrome extension-based XSS defensive framework for contemporary platforms of web applications. *Security and Communication Networks*, 9(17):3966–3986, 2016.

[14] Ian Hickson. Web SQL database. WD not longer in development, W3C, November 2010. https://www.w3.org/TR/2010/NOTE-webdatabase-20101118/.

[15] Ian Hickson. Web storage. W3C recommendation, W3C, July 2013. https://www.w3.org/TR/2013/REC-webstorage-20130730/.

[16] Ian Hickson. Web storage (second edition). W3C recommendation, W3C, April 2016. https://www.w3.org/TR/2016/REC-webstorage-20160419/.

[17] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, pages 171–183, 2017.

[18] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. Information leaks via safari's intelligent tracking prevention. *arXiv:2001.07421*, 2020.

[19] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[20] Pierre Laperdrix, Nataliia Bielova, B. Baudry, and G. Avoine. Browser fingerprinting. *ACM Transactions on the Web (TWEB)*, 14:1–33, 2020.

[21] Lei Liu, Xinwen Zhang, Guanhua Yan, Songqing Chen, et al. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[22] BuiltWith Pty Ltd. Websites using navigator geolocation. https://trends.builtwith.com/websitelist/Navigator-GeoLocation.

[23] Giorgio Maone. Noscript. https://noscript.net/.

[24] Johan Mazel, Richard Garnier, and Kensuke Fukuda. A comparison of web privacy protection techniques. *Computer Communications*, 144:162–174, 2019.

[25] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 820–830, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[26] Statcounter. Browser market share worldwide — bar chart and csv dataset. https://gs.statcounter.com/browser-market-share#monthly-202011-202011-bar.

[27] Antoine Vastel. *Tracking Versus Security: Investigating the Two Facets of Browser Fingerprinting.* PhD thesis, Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL), 2019.