PanoptiChrome: A Modern In-browser Taint Analysis Framework

Rahul Kanyal
Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
rahulkanyal@cse.iitd.ac.in

Smruti R. Sarangi
Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

ABSTRACT

Taint tracking in web browsers is a problem of profound interest because it allows developers to accurately understand the flow of sensitive data across JavaScript (JS) functions. Modern websites load JS functions from either the web server or other third-party sites, hence this problem has acquired a much more complex and pernicious dimension. Sadly, for the latest version of the Chromium browser (used by 75% of users), there is no dynamic taint propagation engine primarily because it is incredibly complex to build one. The nearest contending work in this space was published in 2018 for version 57; at the time of writing, we are at Chromium version 117, and the current version is very different from the 2018 version. We outline the details of a multi-year effort in this paper that led to *PanoptiChrome*, which accurately tracks information flow across an arbitrary number of sources and sinks and is, to a large extent, portable across platforms.

As an example use case of the platform, we experimentally show that we can discover fingerprinting APIs that can uniquely identify the browser and sometimes the user, which are missed by state-of-the-art tools, owing to our comprehensive dynamic analysis methodology. For the top 20,000 most popular websites, we discovered a total of 362 APIs that have the potential to be used for fingerprinting – out of these, 208 APIs were previously not reported by state-of-the-art tools.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Information flow control*.

KEYWORDS

JavaScript Taint Tracking, Program Analysis, Chromium Browser, Browser Fingerprinting, Web Measurement, Privacy

ACM Reference Format:
Rahul Kanyal and Smruti R. Sarangi. 2024. PanoptiChrome: A Modern In-browser Taint Analysis Framework. In *Proceedings of the ACM Web Conference 2024 (WWW '24), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3589334.3645699

WWW '24, May 13–17, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0171-9/24/05
https://doi.org/10.1145/3589334.3645699

1 INTRODUCTION

JavaScript (JS) ranks as the most widely used programming language, with over seventy percent of developers leveraging it for development[6]. JS has become such a ubiquitous programming language because of the availability of a very rich set of APIs for creating interactive web applications and dynamic server-side scripts. In addition, through the use of the Electron [2] framework, JS can also be used to develop full desktop and mobile applications. Due to its widespread usage, there exist a plethora of third-party libraries that are used across client, server, desktop and mobile applications. Since these libraries offer a diverse range of functionalities for collecting users' access patterns, session record and replay, advertisements, and a host of other features, it is a common practice for websites to include third-party libraries from various domains [25].

Apart from software bugs and vulnerabilities, the security and privacy threats to the hosting site increases manifold when third and fourth-party libraries get included. These libraries have equal privileges and can access and alter the shared page state (DOM, JS variables) from other co-located scripts. As per a recent survey [1], 37% of third-party scripts are known to contain undisclosed vulnerabilities – this puts all kinds of personal data such as passwords, medical data and credit card details at risk. In addition to explicit information leaks, side channels can also be used to uniquely identify browsers and characterize user behaviour, such as through the use of battery level indicators [36]. These subtle sources of information are the basis for the area of *browser fingerprinting* [21]. Hence, to summarize, a comprehensive security analysis tool at the client side that can track the flow of information across code from different scripts and third-party libraries is necessary to identify APIs and websites that display such malicious behavior.

Information flow analysis or taint analysis in web browsers is an established problem. Here, the flow of information is tracked from a sensitive source, such as a password field to a sink, which can potentially exfiltrate the data to an unauthorized party. Analyses can either be static [29, 34] or dynamic [32, 39, 40]. A criticism of static analysis approaches is that they are either overly conservative or miss out on capturing vital dynamic information . A lot of information is unavailable at compile time, such as the contents of third-party APIs and the results of *eval* calls, where a JS statement is created dynamically. Dynamic analysis, on the other hand, is difficult to implement because it involves invasive changes to the code of the web browser and JS engine (V8 in the case of Chromium). To provide context, it's worth noting that the combined code size of Chromium and the V8 engine is quite large: 40 million lines [10] and 3 million lines [11], respectively. Additionally, their memory allocation and Garbage Collection (GC) mechanisms are quite complex, which can pose challenges when attempting to add metadata to objects or track information flow, particularly implicit flows.

Hence, many researchers [22, 27] have opted for simpler methods where they annotate JS APIs and then log their executions. This can either be easily detected or requires a complete reimplementation of the entire runtime in JS. Furthermore, these methods often miss many subtle interactions and control flow based dependences. As a result, the gold standard in this area is to track flows by modifying the browser and the JS engine, which is what the nearest work, Mystique [17], did for Chromium version 57. At the time of writing, we are currently at version 117, and in the last 60 versions, a lot of fundamental changes have happened in the source code. For instance, Chromium has transitioned from a stack-based to a register-based virtual machine, the execution pipeline has changed, and the memory management and GC systems have been completely overhauled.

We thus propose a bespoke dynamic taint analysis framework called *PanoptiChrome*, which adds roughly 7,000 lines of code to the existing V8 engine. Its novel features are as follows.

❶ It accurately captures all kinds of information flows (explicit and implicit) while supporting a variable number of sources and sinks that can be changed at runtime. ❷ It is mostly portable across Chromium versions (requires minor changes) and is platform agnostic. ❸ From a software engineering point of view, *PanoptiChrome* is fairly self-contained, where all the changes are limited to the V8 engine's Ignition module (interpreter part) only. ❹ Our solution works with V8's complex garbage collection and memory relocation framework. ❺ We tested the efficacy of *PanoptiChrome* for the problem of identifying fingerprinting APIs across the top 20,000 websites. We identify 164 hitherto undiscovered APIs that are potentially fingerprinting.

Section 2 describes the background, Section 3 elaborates on the design of *PanoptiChrome*, Section 4 shows the evaluation results, Section 5 describes the related work, and finally we conclude in Section 6.

2 BACKGROUND

The following section begins with an overview of WebIDL, information flow analysis and its usage in various aspects of web privacy and security. We then present the reasons for adopting a static+dynamic taint analysis algorithm over static taint analysis approaches. Furthermore, we establish the need to instrument the runtime system, specifically the V8 engine.

2.1 Browser APIs and WebIDL

Browser APIs allow a website to access certain features such as the browser type, current date and time, screen dimensions, etc. Since different browsers might offer different capabilities with differing syntax, the WebIDL [12] standard formalizes the interfaces and properties that need to be offered by a compliant browser. In Chromium, these APIs are implemented as a part of the Blink rendering engine; they are exposed to web applications using the standard Web IDL specification [12]. Figure 1 presents an overview of the Chromium architecture and the Web IDL interface between the V8 engine and the Web APIs.

Due to the variability in the nature of the devices that access a given site, along with geographical differences (detected from the IP address and time zone), browser APIs typically return different

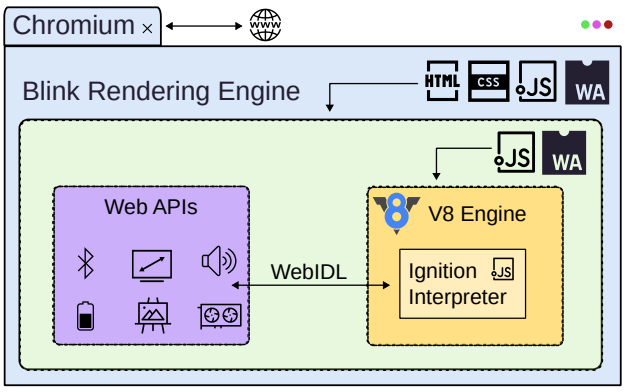

Figure 1: Overview of the Chromium browser

kinds of information to different devices. These values obtained from browser APIs, along with the meta-information about the device, allow the site to create a unique *fingerprint* for each device. The site can then use this fingerprint to track the user across different sessions and websites, even if the network used to access the site changes.

2.2 Information Flow Analysis

JS uses an asynchronous event-driven model, where functions are registered as event handlers and invoked when the corresponding event happens, such as a "mouse hover" or a "mouse click". As a result of the dynamic nature of events, static analysis approaches that solely analyze the source code of an application are ineffective and seldom generate accurate information/control flow information. Another factor that complicates static analyses is the fact that JS is a deeply object-oriented language where properties (variables/functions) can be added and removed dynamically from parent classes, and functions are treated as objects. They also may take a variable number of arguments. To complicate matters, JS has the reflection API[3] and functions like *eval*, *setTimeout* and *setInterval* [4] that allow the interpreter to execute strings provided at runtime as code; this makes static analysis nearly impossible. Hence, dynamic analysis approaches are required that can inspect the executing code at runtime. Approaches that instrument the JS code instead of the runtime cannot add hooks to all the objects' properties without a priori information about the properties themselves. Further, multiple API calls like *document.location* exist that a proxy object with hooks cannot wrap. Also, an adversary can easily detect such hooks.

Mystique [17] is the nearest work that added dynamic information flow analysis to the Chromium browser. For explicit flows (due to assignments), Mystique [17] created an edge in the Data Flow Graph (DFG) from the R-value to the L-value. Implicit flows where there is data transfer from caller to callee parameters during function calls and returns are also handled in a similar manner. To handle control dependencies, all the variables in the branch path are tainted. The limitations of Mystique [17] are that it does not handle dynamic sources and sinks, is not designed for a register-based machine, modifies the garbage collection engine and memory relocator, has extended object liveness, and requires changes in

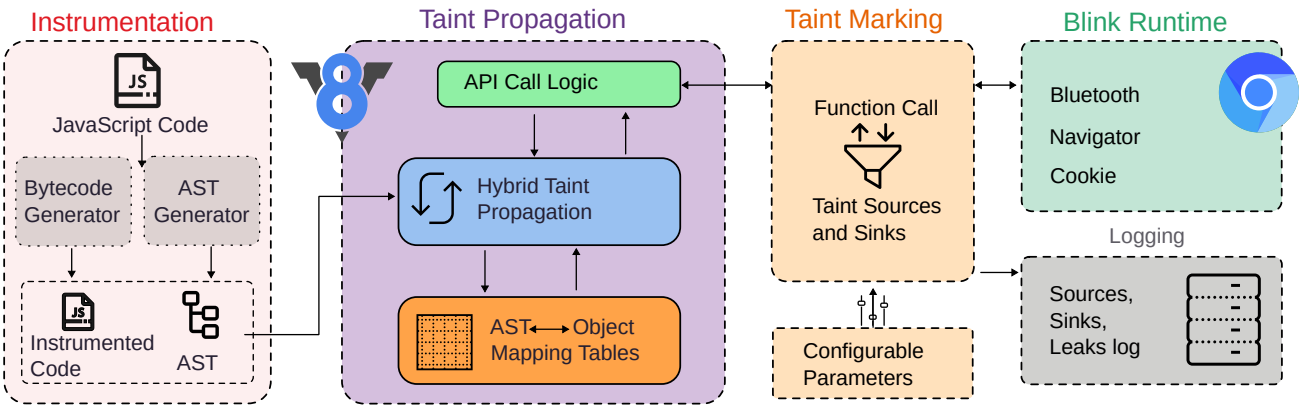

Figure 2: Overview of *PanoptiChrome*

Chromium's Blink rendering engine - all of these changes reduce the portability of the design. Furthermore, because of its limited set of sinks, it misses many information leakage paths.

3 DESIGN OF PANOPTICHROME

3.1 Design Overview

To detect data leakage from the browser, *PanoptiChrome* needs to identify and mark all the values obtained from a known subset of browser APIs as *tainted*. ❶ We instrument the code generated for API methods and property accesses and add hooks (callback functions). ❷ The custom **T**aint **M**arking **E**ngine**(TME)** handles the marking of tainted values as these APIs get accessed. ❸ Once we have identified and marked the taint sources, the **T**aint **P**ropagation **S**ystem **(TPS)** disseminates the taint tags via explicit and implicit flows to all the objects that somehow use the tainted value (directly or indirectly). ❹ On invocation of an API labelled *sink*, all the parameters passed to the API are checked for their *taint* status. If a parameter is found to contain data from tainted sources; the sources, sink and the parameters are logged to a file. An overview of the steps in *PanoptiChrome* for taint marking and propagation is presented in Figure 2. The patches to the Chromium's V8 engine developed for *PanoptiChrome* are available here.

3.2 Code Instrumentation and Data Structures

The `Bytecode Generator` in V8 walks the AST (Abstract Syntax Tree) generated by the JS parsing phase to emit intermediate code that the `Ignition` engine interprets. The `Ignition` engine in V8 is a register-based interpreter with handlers for around 230 bytecodes. 60 bytecode builders are responsible for emitting the properly formatted bytecode. There are 85 visitors that walk the AST generate handlers for these 230 bytecodes. *PanoptiChrome* needs to modify only **three** builders and **eight** AST visitors to track the flow of tainted information through the execution of the JS code.

In prior work [17], the taint marking was done when the rendering engine called a JS function. However, in our scheme we track dependences at a finer level and we can change the sources and sinks at runtime. Hence, in our case, the taint marking step must be intertwined with the taint propagation step. For every JS object (defined in the source code), there is a runtime object (internal to V8). Whenever we access a method or property in a JS object, we need to use the TME engine to find if we need to taint the status of the corresponding runtime object. The TME engine needs to check the list of current sources.

3.2.1 Data Structures Used. *PanoptiChrome* uses multiple hash tables to store the taint status of JS objects and their corresponding runtime objects. The **Object Taint Table** (OTT) stores the taint metadata of the runtime object and is indexed using the `ptr` (tagged heap pointer) data member of the runtime object. This hash table stores information about all the taint sources for the given runtime object. There is an important design decision here. Should we store a list of all the methods/properties via which the taint flowed to a given object's method or property? Given that prior work considers few sources, they indeed store this information. This is not a scalable solution because references to all the objects on the path will remain live, and the GC will not be able to remove them – this results in a large memory footprint.

We thus maintain two references in each OTT row: a reference to the runtime object and a reference to the string encoding of taint sources (the runtime objects on the path are not stored). An overview of the liveness of objects and handles in *PanoptiChrome* is depicted in Figure 3. If the OTT is reachable, then all the runtime objects that it points to will also remain alive, which is something that we do not want because many objects will not have valid references to them in the original JS code left. V8 can create weak references, a pointer where the destination object can be garbage collected. Such weak references are used here. The **crux of the idea** here is to use two weak references: one to the runtime object and one to the string encoding of the taint sources. The *Ephemeron hash table* ensures that if the runtime object is alive (because of references in the JS code), then the string encoding will also be alive (not garbage collected). This guarantees the existence of the string object (corresponding to the taint sources) once we reach the sink because the sink needs to be alive.

The **AST Taint Table** stores the taint status of the parsed nodes in an AST for the functions in the current activation stack. *PanoptiChrome* implements the **AST Taint Table** in the same fashion as

Mystique [17]. It uses a multi-level `HashTable` in which the first level is indexed by the frame pointer of the currently executing function, while the second level is indexed using the unique location of the node in the AST. The link between the AST node and the corresponding runtime object is maintained using the `SimpleNumberDictionary` (internal to V8) that maps the location and type (VariableProxy, Property or Call) of the AST node to the corresponding runtime object. Like the **AST Taint Table**, the **AST to Object Map Table** contains multiple levels wherein the current frame pointer indexes the first level while the second level stores the actual mapping.

3.2.2 Liveness of Objects. All the tables we add can be garbage collected; this must be avoided at all costs. Prior work [17] modified the GC itself, a very invasive change that harms portability and maintainability. We start with observing that all runtime objects in V8 can be referenced with the help of `Handles`. These are themselves not garbage collected. The `Handles` are stored in a `HandleScope` that is responsible for deallocating the `Handles` when the scope is destroyed. To make sure that the `Handles` responsible for taint tables are not deallocated, we store them in a custom `PersistentHandle` that is aware of the special tables and is not deallocated until the `PersistentHandle` is explicitly reset. The custom `PersistentHandle` is created at the start of the execution and is destroyed only when the execution ends. The stock V8 engine does not allow the deletion or updation of `Handles` added to the `PersistentHandle` list; hence, we introduce new interfaces that allow us to replace the unused handles with null runtime objects.

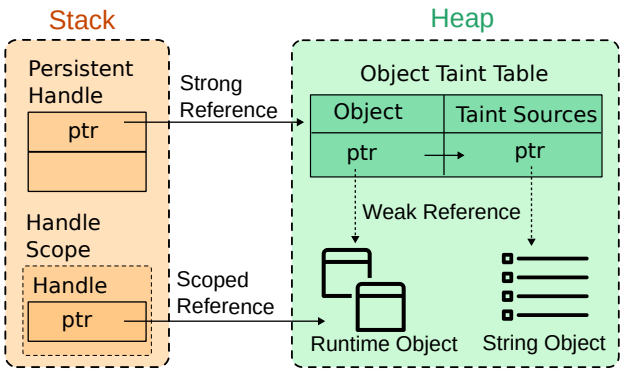

Figure 3: Liveness in the Object Taint Table

3.3 Taint Marking Engine (TME)

The role of the **T**aint **M**arking **E**ngine**(TME)** in *PanoptiChrome* is to identify the sources and sinks defined in the configuration files and taint the corresponding runtime objects. The dynamic configuration files (user-defined) contain the object's name and the corresponding methods/properties, which should be considered as sources or sinks. *PanoptiChrome* provides the same expressiveness as OpenWPM[22] for specifying the sources and sinks with the ability to selectively choose/reject specific properties or methods of an object. TME also filters out contexts where the sources should not be tainted. These contexts include native built-in functions (called during initial setup) and Chromium's intrinsic functionalities (like settings or a new tab). The TME receives details about the object, the API to access, values of all the parameters passed to the API and the return value. Based on the information received and the context derived from the object, TME sets the taint for the return value.

A simple lookup of the object and member name (property or method) in the custom taint configuration database is insufficient since members can be references to runtime objects. Furthermore, prototypal inheritance in JS allows a child object to access all the members of superclassses. To solve these problems, TME needs the object's name (similar to runtime type information in C++). Using the constructor's details, the TME walks up the inheritance chain and finds the object in which the member is defined (the WebIDL specification is used to speed up this process). In some cases, when only a member is provided, the default object is *Window* (regular JS semantics). Once we find the object, we check whether it is a tainted source or not. This is more elegant and generic than Mystique [17], which required patching all the Blink endpoints (7000+ when writing this paper) and then tracking their accesses.

3.4 Taint Propagation System (TPS)

To ensure that the original execution is not affected while propagating taint information, *PanoptiChrome* follows a caller-saved scheme – store the original values in a set of virtual registers and allocate independent registers for storing taint metadata before starting the taint propagation routines. After the routines return, the original state is restored. To ensure proper taint propagation, we include vital information about object constructors (see Section 3.3) in the parameters that we pass to the taint routines as opposed to earlier frameworks like Mystique [17] that did not do so.

On function exit, the **AST Taint Table** and **AST to Object Map Table** are dropped (since every invocation requires a fresh AST Taint Table and map). In contrast, the Object Taint Table persists across invocations to further propagate the taint status. Once the **TME** has marked the values obtained from a select set of browser APIs as taint sources, **TPS** sends the taint tags to other objects that receive information from the labelled tainted sources. *PanoptiChrome* performs an order-independent, intra-procedural analysis on the source code received by the V8 engine for execution to create the **F**low **G**raph **(FG)** (combination of the data and control flow graphs).

Initially, an AST is generated at the level of an individual function. Analysis at the function level is sound since even the top-level scope is considered a function(unnamed). *PanoptiChrome* repurposes the parser and code generator used by the V8 JS engine to create the ASTs. Then, the generated AST is cached for future invocations. This AST is then used to construct the FG in which the vertices are nodes from the AST (56 such types in version 11.7 of V8) representing either a property access or an API call. The FG contains directed edges between the AST nodes if there is an explicit flow of information (via the assignment operator) or an implicit flow (via conditional statements).

To reduce the overhead of taint propagation, *PanoptiChrome* does not create an FG unless at least one tainted source has been visited

in the scope of the function under analysis. Once an API call has been marked tainted by the **TME**, **TPS** marks the corresponding AST node in the FG as tainted with the help of the $object \rightarrow \langle ASTnode \rangle$ mapping table similar to the **AST to Object Map Table** (see Section 3.2). Taint propagation routines are invoked only when one of the following conditions is true: ❶ an API marked as a *sink* is called, or ❷ the function returns and an object/array is created by the function. Taint propagation is then carried out by following the outgoing edges from the tainted AST node and updating the taint status of each node in the forward slice (transitive closure of nodes in the FG). Also, whenever a node is marked as tainted in the FG, the corresponding object is marked as tainted with the help of a reverse mapping table (**AST to Object Map Table**). *PanoptiChrome* can propagate taint to local storage similar to prior work [17] with the help of an additional taint marker appended to the data written to local storage. Note that the marker is removed while reading the tainted data. For DOM taint propagation, either an approach similar to that used for local storage can be followed, or the string implementation in Blink can be made aware of the taint data, the patch for which is also available here.

3.5 Logging Data

The V8 engine utilizes an `Isolate` to separate different execution contexts on the same web page. These multiple execution contexts get created due to the inclusion of numerous `iframes` (webpages loaded from different origins) in the same web page. For every `iframe`, a separate `Isolate` is instantiated with its copy of global objects and built-in functions. Distinct isolates on the same web page execute concurrently using separate threads and behave as individual sandboxed instances of the V8 runtime. Since *PanoptiChrome* attaches all the tables required for taint marking and propagating with an `Isolate`, multiple runtime instances can execute in parallel without treading on each other's data. We log all the data for each isolate separately in a different file similar to Visible V8 [27]; hence, the problem of inter-process synchronization is solved by design. For every `Isolate` *PanoptiChrome* logs all the origins (multiple sub-domains can be loaded in the same `Isolate` as long as they share the same origin) along with the tainted source APIs and sink APIs invoked during the execution. Additionally, the leaks are logged (along with the string-encoded list of taint sources) whenever data from a tainted source flows into a sink API.

4 EVALUATION

4.1 Setup

We used an `AMD EPYC 7702P` powered workstation with 64 physical cores and 128 GB RAM for crawling and post-processing tasks. Log files, averaging 150KB per website, were stored on a 1 TB SSD. The crawling process utilizes the latest Chromium browser (version 117.0) compiled with our custom V8 engine (*PanoptiChrome*). Websites are loaded concurrently in independent browser windows with transient user profiles that get erased after each website visit. Additionally, navigation and website loading are facilitated through a commercial off-the-shelf ISP used by more than 38% of the active internet users in the author's country [9], India.

4.2 Crawling Methodology

We instantiate each browser instance from the command line and pass the website URL as a parameter. We do not use any automation framework; this makes our approach as indistinguishable as possible from a regular user accessing the site. The driver script closes the browser after a preset time of 180 seconds and then starts another instance. Figure 4 summarizes the steps in the data collection pipeline. We use the top 20,000 websites from the Tranco list [7] as the seed URLs for data collection. We perform a connection test for each URL by requesting the HTTP header from the website. To request the HTTP header containing the website's status code, we initially attempt to connect to the website by appending HTTPS:// with the domain name from the Tranco list. If the connection succeeds, we log the schema and URL to the list of reachable URLs. If we fail to connect to the site within 15 seconds, we try with the HTTP protocol for the next 15 seconds. If we still fail, we log the URL with the corresponding error code.

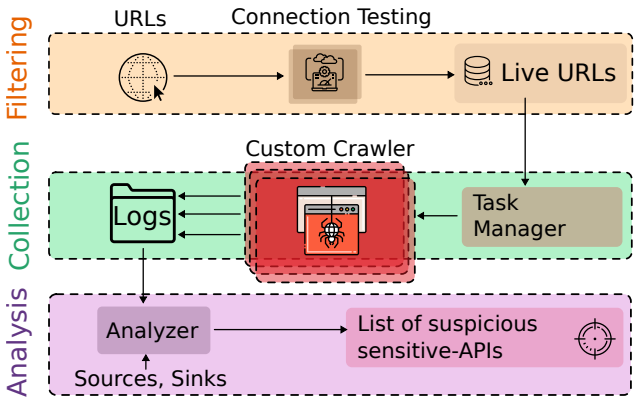

Figure 4: Overview of the data collection pipeline

Crawling results From our network vantage point, 61.91% of the Tranco top 20,000 websites were reachable (status code 200). Around 14% returned a 404 (not found) error, and around 24% timed out with both the HTTPS and HTTP protocols. Table 1 represents the status codes for the URLs in the list. Our crawler could log 12,846 unique origins and recorded 45,942,545 API calls and 24,486 leak entries. Furthermore, the recorded origins invoked 5,673 unique APIs, of which 3,426 are DOM manipulation APIs.

Table 1: Status Codes for the top 20,000 Tranco URLs

Status Code	Number of Sites	Percentage
200-299 (Success)	12382	61.91%
300-399 (Redirection)	5	0.03%
400-499 (Client Error)	2770	13.85%
500-599 (Server Error)	141	0.7%
Exception	4701	23.5%

4.3 Web API Categorization

Web API categorization is required to identify the DOM manipulation APIs that are used to get the static properties of elements in the

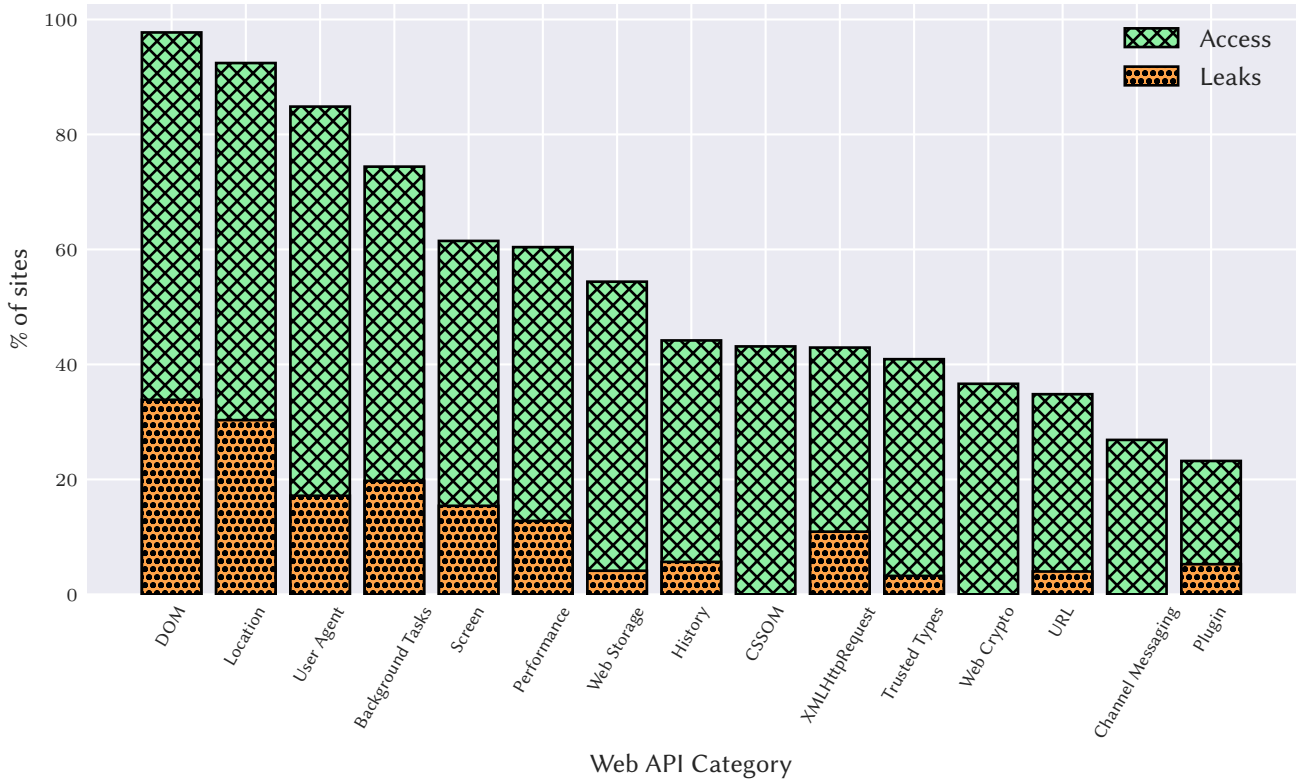

Figure 5: API access and leak in sites vs top 15 Web API categories (based on access)

web page. The values returned by these APIs are always the same for a particular element across browsers and, hence, cannot be used to fingerprint the user. In our analysis, we do not consider these APIs as sensitive. To classify the APIs, the category was decided using the developer documentation provided by Mozilla Developer Network (MDN)[5]. Out of 5,673 unique APIs in our crawl, MDN had no categorization for 1,246, which we manually classified after analyzing the documentation. The complete list of API categories can be found here. Also, for APIs categorized in multiple categories, we give the lowest priority to the `DOM` category and classify the API manually into the bin with the highest specificity. For example, `Navigator.clipboard` is categorized as both 'Clipboard' and 'User-Agent' according to MDN; in our case, the 'Clipboard' category is chosen (because of more specificity/relevance).

4.4 Data Leakage from APIs

We define *data leakage* as the flow of information from any web API to any sink (storage, network). Every API invocation marks the returned data as tainted. An entry is logged whenever any tainted data reaches a sink. Out of the 5,673 unique web APIs that are accessed by 12,846 origins, our analysis reveals that data from a total of 675 unique APIs is leaked. We observe that, on average, 115 unique APIs are accessed on a website and data from 11 unique APIs is leaked. The maximum number of APIs accessed from a single origin is 531, whereas the maximum single-origin leakage was 144. DOM-related APIs (such as `NodeList.length`) are leaked on 33.84% of the sites, followed by the `Location`, the `Background Tasks` and the `User Agent` category (30.31%, 19.74% and 17.15%, respectively).

The `HTMLAnchorElement.hostname` and the `Window.location` are the most commonly leaked APIs in the Location category. These APIs are used to get the domain name of the page for constructing dynamic links or fetching web resources. In the Background Tasks category, the `Window.setTimeout` API (used to execute a JS function after a set amount of time) while `Window.navigator` and `Navigator.userAgent` from the User-Agent category (used to customize the website for different screen resolutions and sizes) are the most prominent in the set of leaks. The complete category-wise distribution of API accesses and leaks for the top 15 Web API categories (based on access) is shown in Figure 5. Of all the APIs that get leaked in a site, 55.58% belong to the DOM category. The average distribution of API categories (in the remaining 44.42%) that get leaked per site is shown in Figure 6.

4.5 Fingerprinting APIs

Sensitive APIs are those APIs that can be potentially used for fingerprinting. *PanoptiChrome* analyzes all the parameters passed to a sink and records the sources from which the information sent to a sink was generated. Out of the 675 unique APIs that get leaked, 269 APIs are DOM manipulation APIs. 121 APIs from the remaining 406 APIs have already been classified in previous work [38] as *sensitive*, and 33 have been categorized as *URL* or *sink-related* APIs that are

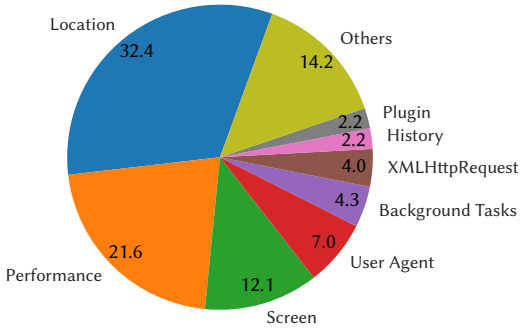

Figure 6: Average API leak distribution per site

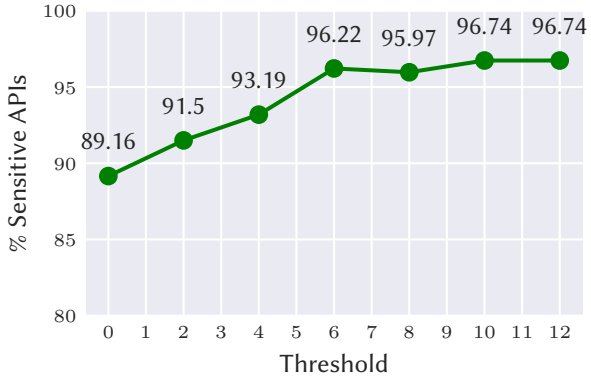

Figure 7: Variation of the #sensitive APIs with the threshold

used for fingerprinting indirectly (as a means of ferrying already fingerprinted data).

For the classification of the remaining APIs (252 APIs), we followed a method similar to prior work [38], where we investigated known fingerprinting websites for the use of the discovered APIs. In total, 78 APIs were confirmed to be fingerprinting using this method. For the remaining 186 APIs, we manually consulted the documentation for each API and the source code of the websites that use the API to establish the association with fingerprinting. 82 APIs were manually classified to have the potential to be used for fingerprinting, while 48 APIs were classified as sinks or providing URL-related data. **To summarize, we discover** a total of 362 APIs (121 + 33 + 48 + 78 + 82) that are probably being used for fingerprinting or have the potential to be misused. The complete list of APIs can be accessed here. Out of these 362 APIs, 208 APIs were previously unreported by state-of-the-art works.

4.5.1 Effect of Co-location of API Calls on Precision and Recall. Out of the 675 APIs that are leaked by the sites, 39.85% are used for DOM manipulation and 89.16% of the remaining APIs (total minus DOM) were confirmed to have the potential to be used as fingerprinting vectors (sensitive). For each sink, *PanoptiChrome* reports a list of sources that are used to compute the tainted value. For each such list, we check if it contains more APIs than a pre-defined threshold. For this experiment, we do the following: if this threshold is breached, we mark all the APIs in the list as *sensitive*. This follows from the observation in [14, 26] that browser fingerprinting often clubs data from multiple sources to form a unique identifier. After removing DOM manipulation APIs, we vary the threshold from 0 (all sources included) to 12 (maximum number of seeds found in a single leak) and plot the resulting distribution of the percentage of suspicious APIs found in Figure 7. It can be observed that the percentage of APIs detected to be sensitive increases with the threshold (precision increases at the cost of recall). With a threshold of 0, 406 APIs are marked for further analysis (with 89.16% being sensitive), while with a threshold of 10, only 92 APIs are marked for further analysis. Out of the 92 APIs marked, 96.74% are confirmed to be sensitive.

Comparison with State-of-the-art: For comparative analysis with the state-of-the-art tool for fingerprinting – BFAD[38] – we collected the API logs using VisibleV8 [27] for the Tranco top 1000 sites. The results are shown in Figure 8. Only 608 of these 1000 sites were reachable from our network vantage point. BFAD confirmed 68 APIs to be potentially fingerprinting (sensitive). *PanoptiChrome* discovers 438 unique APIs from which the data is leaked. Of these 438 APIs, 183 are used for DOM manipulation and are not considered for fingerprinting. In the remaining 255 APIs, we verify 237 APIs as potential vectors for fingerprinting manually or by using the known fingerprinting approaches.

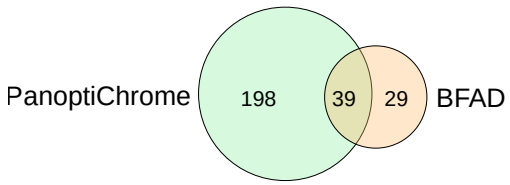

Figure 8: Comparison with the state-of-the-art, BFAD[38]

5 RELATED WORK

Dennings [19, 20] pioneered the formalization of static analysis approaches in the 1970s. Fenton [23] then studied purely dynamic monitors for managing information flows. Much of the later work has focused on adapting the work of Denning and Fenton to different languages and proposing solutions with various limitations.

Dynamic analysis techniques using virtual machines [24], source code instrumentation [37], and runtime instrumentation [15] have been employed for numerous use cases ranging from JS execution visualization [33] and record/replay [37] to privacy and security analysis of browser extensions [17] and policy enforcement[8]. Table 2 summarizes the features and limitations of existing information flow approaches for JavaScript engines. **Fine-Grained Taint Tracking** (FGTT) specifies the granularity of the taint tracking mechanism. For example, in [13], the taint is tracked at the level of scripts, whereas in the case of PanoptiChrome, the granularity of taint tracking is at the level of individual JS objects. In essence, any mechanism that instruments the binding between V8 and Blink only, without instrumenting the flows inside V8, is coarse-grained, whereas mechanisms with instrumentations of flow inside V8 are

Table 2: Summary of information flow analysis in different JS runtimes.

Work	FGTT	Implicit flows	Custom sources	Completeness	Upgradability	Platform agnostic
Vogt et al.[40]	✓	✓	✗	✗	✗	✗
DOMsday[35]	✓	✗	✗	✗	✗	✗
WebPol[8]	✓	✓	✓	✓	✗	✓
Runtime monitoring[15]	✗	✗	✗	✗	✓	✓
Crowdflow[30]	✓	✗	✗	✓	✗	✗
FPDetective[13]	✗	✗	✗	✗	✓	✓
JSgraph[33]	✗	✗	✗	✗	✓	✓
Visible V8[27]	✗	✗	✗	✗	✓	✓
Mystique[17]	✓	✓	✗	✓	✗	✗
25 million flows[32]	✓	✗	✗	✗	✗	✗
Foxhound[31]	✓	✓	✗	✗	✗	✓
PanoptiChrome	✓	✓	✓	✓	✓	✓

fine-grained. **Upgradability** describes the feature of a taint tracking mechanism to be conducive to constant code updates in the original engine. Specifically, we classify a taint tracking mechanism as non-upgradable if the instrumentation requires patching all the possible flow paths individually and sensitizing all the data structures involved in the taint flow. A taint system is classified as **platform agnostic** if all the changes are independent of the underlying system architecture and the system can be easily ported (with modifications in the build system only). PanoptiChrome is platform agnostic, has been tested with Android and Linux, and can be built for other platforms. **Custom sources** represents a taint engine's ability to accommodate a different set of taint sources and sinks that can be revised without requiring any modification in the engine's source code. **Completeness** of a taint engine describes the ability to track taint flow across all eight data types (for example, string, number, object) in JavaScript.

5.1 Augmenting Browsers with Taint Tracking

Vogt et al.[40] supplement dynamic taint tracking with static analysis to detect DOM-XSS vulnerabilities. They use static analysis to propagate taint information along implicit flows. On the same lines, in reference [32], the authors instrument the Chromium browser to track tainted strings (limited use case). Their goal was to detect and validate DOM-XSS vulnerabilities. Based on the source and context of the tainted data, they automatically generate the breakout sequence to validate the vulnerability. Like [32], Domsday [35] also instruments the Chromium browser to detect DOM-XSS vulnerabilities. The authors add one byte to each string object to keep track of the encoding and decoding functions and the data's provenance. This is also a limited use case. Another such work is FP-Detective [13], which only looks at font-related APIs.

Bauer et al. [15] treat the V8 JS engine as a black box and track information flow only across the Blink-V8 boundary; this can be used to sandbox scripts based on their respective origins. Their coarse-grained information flow approach cannot handle implicit flows and cannot reason if a source API is exploited for illegitimate use. In Crowdflow [30], the authors aim to minimize the limitations of information flow tracking by probabilistically switching between partial taint tracking and information flow monitoring in a distributed setting. The clients report a violation to an aggregator that takes appropriate action. Similar to Domsday [35], CrowdFlow employs heavyweight instrumentation and uses a fixed set of sources and sinks tailored to detect XSS-based vulnerabilities.

PanoptiChrome is much more generic than all the prior work and is not meant to target taint information for specific data types (or object types). Its taint tracking is also much more fine-grained. Unlike prior work, it does not rely on any custom JS engine that only handles a subset of the language; it can handle any site that the Chromium browser can handle.

5.2 Study of Third-Party Data Exfiltration

In this space, the closest approaches that are similar to *PanoptiChrome* are Mystique[17], Jest [18], Ichnaea [28], and JSFlow [24]. JSFlow [33] uses a bespoke JS interpreter for a subset of JS. Jest [18] is a source-code instrumentation-based approach that converts every statement and expression to a function call, and these instrumented functions are responsible for implementing the dynamic analysis methods. Ichnaea [28] is built on top of Jalangi [37], which is also a source-code instrumentation-based approach. The instrumentations proposed by both Jest and Ichnaea can be detected easily by an adversary [16].

6 CONCLUSION

We showed in this paper that it is indeed possible to build a comprehensive dynamic taint tracking engine that is completely generic and is portable across platforms and browser versions to a large extent. This was achieved by limiting the changes to a small part (interpreter) of the V8 engine and suggesting smart solutions for the dynamic addition of sources/sinks, creating persistent handles to circumvent the issues caused by the GC and memory relocation engines and optimizing the process of taint propagation by using an on-demand algorithm. We used *PanoptiChrome* to perform a detailed characterization of the information leakage in the top 20,000 websites. We further use the locality information inherent in the logs generated by *PanoptiChrome* to significantly reduce the set of APIs to be considered for manual analysis. The need for having *PanoptiChrome* is attested by the fact that we discovered 208 APIs that were not known to have a fingerprinting character.

REFERENCES

[1] 2022. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web - NDSS Symposium. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/thou-shalt-not-depend-me-analysing-use-outdated-javascript-libraries-web [Online; accessed 24. Feb. 2023].

[2] 2023. Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron. https://www.electronjs.org [Online; accessed 1. Mar. 2023].

[3] 2023. ECMAScript 2015 Language Specification – ECMA-262 6th Edition. https://262.ecma-international.org/6.0 [Online; accessed 11. Oct. 2023].

[4] 2023. ECMAScript® 2024 Language Specification. https://tc39.es/ecma262 [Online; accessed 11. Oct. 2023].

[5] 2023. HTMLAnchorElement: hostname property - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API [Online; accessed 11. Oct. 2023].

[6] 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023 [Online; accessed 11. Oct. 2023].

[7] 2023. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation - NDSS Symposium. https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation [Online; accessed 11. Oct. 2023].

[8] 2023. WebPol: Fine-grained Information Flow Policies for Web Browsers (JSTools 2017) - ECOOP 2017. https://2017.ecoop.org/details/JSTools-2017-papers/6/WebPol-Fine-grained-Information-Flow-Policies-for-Web-Browsers [Online; accessed 24. Feb. 2023].

[9] 2024. India: wireless subscriber market share by provider 2022 | Statista. https://www.statista.com/statistics/258797/market-share-of-the-mobile-telecom-industry-in-india-by-company [Online; accessed 2. Feb. 2024].

[10] 2024. The Chromium (Google Chrome) Open Source Project on Open Hub: Languages Page. https://openhub.net/p/chrome/analyses/latest/languages_summary [Online; accessed 13. Feb. 2024].

[11] 2024. The Google V8 JavaScript Engine Open Source Project on Open Hub: Languages Page. https://openhub.net/p/v8-js/analyses/latest/languages_summary [Online; accessed 13. Feb. 2024].

[12] 2024. Web IDL Standard. https://webidl.spec.whatwg.org [Online; accessed 3. Feb. 2024].

[13] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. Association for Computing Machinery, New York, NY, USA, 1129–1140. https://doi.org/10.1145/2508859.2516674

[14] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2021. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. *Proceedings on Privacy Enhancing Technologies* (2021). https://www.semanticscholar.org/paper/FP-Radar%3A-Longitudinal-Measurement-and-Early-of-Bahrami-Iqbal/72bb8e71702fef660b44133d34b9a8a5456e99c3

[15] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Network and Distributed System Security Symposium*.

[16] Darion Cassel, Su-Chin Lin, Alessio Buraggina, William Wang, Andrew Zhang, Lujo Bauer, Hsu-Chun Hsiao, Limin Jia, and Timothy Libert. 2022. OmniCrawl: Comprehensive Measurement of Web Tracking With Real Desktop and Mobile Browsers. *Proceedings on Privacy Enhancing Technologies* (2022). https://petsymposium.org/popets/2022/popets-2022-0012.php

[17] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1687–1700. https://doi.org/10.1145/3243734.3243823

[18] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 629–643. https://doi.org/10.1145/2810103.2813684

[19] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

[20] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

[21] Peter Eckersley. 2010. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies*. Springer, Berlin, Germany, 1–18. https://doi.org/10.1007/978-3-642-14527-8_1

[22] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 1388–1401. https://doi.org/10.1145/2976749.2978313

[23] J. S. Fenton. 1974. Memoryless subsystems. *Comput. J.* 17, 2 (Jan. 1974), 143–147. https://doi.org/10.1093/comjnl/17.2.143

[24] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC '14: Proceedings of the 29th Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, New York, NY, USA, 1663–1671. https://doi.org/10.1145/2554850.2554909

[25] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. 2019. The Chain of Implicit Trust: An Analysis of the Web Third-party Resources Loading. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2851–2857. https://doi.org/10.1145/3308558.3313521

[26] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1143–1161. https://doi.org/10.1109/SP40001.2021.00017

[27] Jordan Jueckstock and Alexandros Kapravelos. 2019. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *IMC '19: Proceedings of the Internet Measurement Conference*. Association for Computing Machinery, New York, NY, USA, 393–405. https://doi.org/10.1145/3355369.3355599

[28] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. 2018. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Trans. Software Eng.* 46, 12 (Oct. 2018), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[29] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

[30] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. CrowdFlow: Efficient Information Flow Security. In *ISC 2013: Proceedings of the 16th International Conference on Information Security - Volume 7807*. Springer-Verlag, Berlin, Germany, 321–337. https://doi.org/10.1007/978-3-319-27659-5_23

[31] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns. 2022. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 236–250. https://doi.org/10.1109/EuroSP53844.2022.00023

[32] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. Association for Computing Machinery, New York, NY, USA, 1193–1204. https://doi.org/10.1145/2508859.2516703

[33] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2018. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *Network and Distributed System Security Symposium*.

[34] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 499–509. https://doi.org/10.1145/2491411.2491417

[35] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Network and Distributed System Security Symposium*.

[36] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2016. The Leaking Battery. In *Data Privacy Management, and Security Assurance*. Springer, Cham, Switzerland, 254–263. https://doi.org/10.1007/978-3-319-29883-2_18

[37] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[38] Junhua Su and Alexandros Kapravelos. 2023. Automatic Discovery of Emerging Browser Fingerprinting Techniques. In *WWW '23: Proceedings of the ACM Web Conference 2023*. Association for Computing Machinery, New York, NY, USA, 2178–2188. https://doi.org/10.1145/3543507.3583333

[39] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 49–59. https://doi.org/10.1145/2610384.2610385

[40] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium*.