Same Script, Different Behavior: Characterizing Divergent JavaScript Execution Across Different Device Platforms

Ahsan Zafar
azafar2@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Junhua Su
jsu6@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Sohom Datta
sdatta4@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Alexandros Kapravelos
akaprav@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Anupam Das
anupam.das@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

JavaScript drives dynamic content across modern web platforms, yet differences in browser engines, hardware, and APIs create distinct execution environments on mobile and desktop devices. This divergence raises important concerns about platform-specific tracking, but its scope and impact remain underexplored. In this paper, we present a hybrid analysis of JavaScript execution across mobile and desktop to uncover behavioral differences in how identical code operates. By combining static analysis of script structure with dynamic tracing of runtime behavior, we identify execution path divergences tied to the user's device. Our study shows that 20.6% of scripts on the top 10K Tranco-ranked websites exhibit platform-specific execution. Our tracing algorithm pinpoints the sources of divergence for 92.8% of conditional Web API calls, with 76% involving known fingerprinting APIs and 6% relying on lesser-known but platform-revealing interfaces. We further categorize divergent paths, finding asymmetric tracking patterns: desktop flows are dominated by fingerprinting and bot detection, while mobile flows focus more on behavioral profiling.

CCS Concepts

• **Security and privacy → Web application security**.

Keywords

Web Measurement; JavaScript; Static Analysis; Dynamic Analysis

ACM Reference Format:
Ahsan Zafar, Junhua Su, Sohom Datta, Alexandros Kapravelos, and Anupam Das. 2025. Same Script, Different Behavior: Characterizing Divergent JavaScript Execution Across Different Device Platforms. In *Proceedings of the 2025 ACM SIGSAC Conf. on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765202

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765202

1 Introduction

The web has grown into a complex ecosystem, supporting everything from daily tasks like shopping and communication to advanced applications such as real-time analytics and immersive experiences. To meet rising demands for interactivity, browsers have expanded JavaScript capabilities, though execution often differs between desktop and mobile due to hardware, input methods, and API support. Developers use techniques like conditional loading to optimize performance across platforms. A major driver of this complexity is the JavaScript executed on websites, including third-party scripts from ads and trackers. While core functionality may remain consistent, execution paths can vary by platform, activating different code on mobile versus desktop. These differences affect not only usability but also how tracking and fingerprinting work, sometimes enabling more invasive tracking depending on the device. As platform-specific tracking grows, understanding execution differences is critical for addressing privacy and security risks.

Prior studies have explored platform-specific web tracking on mobile and desktop platforms. Eubank et al. [16] used a browser extension to crawl the top 500 websites across desktop and emulated Android, finding substantial overlap in third-party trackers and consistent cookie modifications, but no clear platform-specific differences. Yang et al. [58] introduced WTPatrol, built on OpenWPM [15] (a web privacy measurement framework), to study tracking on Firefox's mobile and desktop versions. Crawling 23,000 websites, they identified over 4,000 trackers, but reported no significant platform disparities. Cassel et al. [11] developed OmniCrawl using *mitmproxy* [2] to intercept network traffic and monitor JavaScript API usage across 20,000 sites on desktop, mobile, and emulated mobile. They found that mobile platforms were more exposed to tracking, while highlighting the limitations of emulated crawlers. While these studies have made valuable contributions, their instrumentation is often low-fidelity, capturing only a subset of dynamically executed APIs. As a result, they may miss nuanced behaviors triggered at runtime, limiting the visibility into the full scope of script execution across platform. In contrast, our approach leverages an instrumented Chromium-based browser with a modified V8 engine (i.e., VisibleV8 [25]) to log low-level JavaScript API and property access, providing near-complete API coverage. This design ensures stealth and accuracy, capturing fine-grained JavaScript execution while minimizing detection by tracking or anti-bot systems.

We adopt a hybrid approach that combines static and dynamic analysis to study divergent fingerprinting behavior across platforms. For static analysis, we examine the structure of JavaScript code using tools that construct Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs) [17], allowing us to understand the structural properties and dependencies within scripts. Complementing this, we use dynamic analysis with VisibleV8 [25] to monitor JavaScript execution traces at runtime to map executed APIs to the PDGs and construct execution graphs. This combined approach enables us to identify script routines that execute on one platform (e.g., mobile) but not on another (e.g., desktop), despite originating from the same codebase. More specifically, we seek to address the following research questions in this paper.

RQ1: What is the prevalence of divergence in the execution of JavaScript on mobile and desktop platforms? We quantify how frequently JavaScript (identically sourced) behavior diverges when executed on mobile versus desktop platforms. This analysis is performed purely from the client side and establishes the baseline extent of platform-dependent execution differences across websites. **RQ2: What information do JavaScript programs use to trigger divergence?** We analyze the data sources and conditions that drive platform-specific branching in JavaScript code. This includes identifying what platform-related information (e.g., user-agent, screen size) scripts access to select exclusive execution paths, and measuring how much of this information is fingerprintable. **RQ3: What do platform-specific JavaScript execution paths accomplish?** We examine the functionality implemented in the divergent code paths to understand the purpose behind platform-specific behavior. This post-divergence analysis helps uncover whether divergence is used for content optimization, tracking, fingerprinting, or other behaviors. In summary, we make the following contributions:

- We introduce a hybrid analysis framework that combines graphical representation of code with execution trace to outline the conditional branching of identically-sourced JavaScript as it runs across various platform environments. Our algorithm locates fine-grained data-sources that contribute to these branching decisions. We have open-sourced our work [1].
- We crawled the top 10,000 websites from the Tranco list [29] using VisibleV8 on both mobile and desktop platforms. Our analysis shows that 20.6% of identically-sourced scripts exhibited divergence across platforms. Notably, desktop environments accounted for 67.9% more divergent execution flows than mobile.
- We analyze the data sources feeding conditional nodes prior to divergence and find that 76% of the associated information chains rely on well-known fingerprinting APIs. An additional 6% involve lesser-known APIs that still expose critical device-specific information to these conditionals. This exploratory analysis lays the foundation for identifying emerging fingerprinting vectors.
- We categorize tracking behaviors across platform-specific divergent flows and find clear trends. On desktops, browser fingerprinting, bot detection, and session state tracking dominate, making up 77% of desktop-only executions. In contrast, mobile flows are driven largely by behavioral tracking—such as touch, click, scroll, and tap events—accounting for 59% of all cases.

2 Background and Related Work

2.1 Web Crawling Tools

The web ecosystem is rapidly evolving and highly heterogeneous, making systematic study challenging. To collect representative data, researchers often employ web crawlers and analyze the results for emerging patterns. Because JavaScript underpins much of the web's dynamic and interactive functionality, it has become a primary focus of such analyses. In turn, specialized tools have been developed to monitor JavaScript execution during crawling, with *OpenWPM* and *VisibleV8* among the most widely adopted in recent research.

OpenWPM. OpenWPM is a Firefox- and Selenium-based web measurement framework designed to collect cookies and monitor a predefined set of browser API calls. It has been widely used to study online tracking and fingerprinting. Englehardt et al. [15] introduced OpenWPM to investigate fingerprinting techniques like Canvas, Audio, and WebRTC. Subsequent studies leveraged it to examine tracking across websites and apps [5, 9], misuse of mobile sensors [13], and tracking in the TV ecosystem [35]. More recent work applied machine learning to detect fingerprinting scripts and analyze browser API abuse at scale [7, 24].

VisibleV8. VisibleV8 is an instrumented version of the V8 JavaScript engine paired with a Puppeteer-based crawler, enabling comprehensive monitoring of browser API execution. Developed by Jueckstock et al. [25], VisibleV8 has been widely used in web measurement studies, including analyses of bot detection [25], JavaScript obfuscation [47], and the impact of crawling vantage points [26]. More recent work has used VisibleV8 for fingerprinting detection [54] and uncovering evasive malicious code through forced execution [41].

Other Crawling Studies. Several studies have employed crawling techniques to investigate online tracking and privacy. Roesner et al. [44] used a Firefox extension to classify third-party trackers, while Mayer et al. [33] developed FourthParty to monitor DOM activity, HTTP requests, and cookies. Acar et al. [6] introduced FPDetective on Chromium and PhantomJS to track fingerprinting APIs, revealing rapid growth. Snyder et al. [50] injected JavaScript via a Firefox extension to study API usage and ad-blocker effects, and Lerner et al. [30] combined a Chrome extension with the Wayback Machine to analyze cookies and fingerprinting over two decades.

2.2 JavaScript Behavior Analysis

In analyzing JavaScript behavior, both static and dynamic approaches have been used, each bringing its own strengths and limitations.

Static Analysis. Static analysis inspects the structure of JavaScript code without executing it, making it useful for identifying fingerprinting routines and third-party dependencies. Techniques like Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs) help map control and data flows, revealing how scripts are constructed and interconnected. While static analysis has been extensively applied to JavaScript in general [27, 31, 51], few studies have specifically focused on its use for web fingerprinting [23]. Tools such as JStap [17] construct data flow graphs and integrate machine learning to detect malicious patterns, while JSFlow [22, 49] traces information flows to identify how tracking data reaches third-party services like Google Analytics. However, static methods fall

short in capturing runtime behavior—such as DOM state changes or dynamic property accesses—making them insufficient alone for detecting divergent fingerprinting behavior.

Dynamic Analysis. Dynamic analysis monitors the runtime behavior of JavaScript by tracking execution traces, crucial for identifying divergent code paths that cause different behaviors on mobile and desktop platforms. While static analysis can suggest potential divergences, it often suffers from high false-positive rates [42]. For this study, we use VisibleV8 [25], an instrumented Chromium browser that provides fine-grained, API-level tracing. Integrated into the JavaScript engine, VisibleV8 logs every API and function call, offering comprehensive coverage compared to other tools that track only a limited set of APIs [11], making it ideal for our analysis.

2.3 Browser Fingerprinting

Browser fingerprinting is a stateless tracking technique that profiles a user's browser and device using standalone APIs (e.g., userAgent) or JavaScript to create persistent identifiers, such as the Canvas hash [15]. Unlike stateful tracking methods (e.g., cookies), which users can clear, fingerprinting relies on relatively permanent device attributes, raising significant privacy concerns. Although fingerprinting can serve legitimate purposes, such as adapting website content to device specifications (e.g., screen resolution), it is widely used for tracking users across the web [24]. Jonathan Mayer was among the first to demonstrate that browser APIs could be exploited to gather fragments of user information, which when combined, can uniquely identify individuals [32]. Peter Eckersley [14] further formalized this concept by showing that 18 bits of identifying information from over 286,000 users were sufficient to create a unique fingerprint. As awareness of fingerprinting grew, research identified a wide range of browser and device features that can be used for tracking, including battery status [40], browser extensions [45, 49, 52], HTML5 Canvas [37], CSS [55], execution timing and clocks [46], WebGL [10, 37], system fonts [19], mobile device configurations [28], JavaScript engine characteristics [36, 38, 48], differences in browser API implementations [39], and even a user's NoScript allowlist [36].

2.4 Cross-Platform Comparison

Eubank et al. [16] were the first to examine mobile web tracking using a modified version of FourthParty, a web measurement extension. By crawling the top 500 websites across desktop, Android, and emulated Android platforms, they found significant overlap in third-party trackers and consistent cookie modifications, though no platform-specific tracking differences emerged. Expanding the scope, Yang et al. [58] introduced WTPatrol, built on OpenWPM, to study tracking behavior across mobile and desktop versions of Firefox. Their crawl of roughly 23,000 websites uncovered over 4,000 trackers leveraging browser APIs and cookies, yet similarly found no meaningful disparity between platforms. Pushing cross-platform analysis further, Cassel et al. [11] developed OmniCrawl, a framework combining *mitmproxy* with JavaScript instrumentation to monitor browser API usage. Their study, covering 20,000 websites across desktop, mobile, and emulated mobile, revealed that emulated or WebDriver-based crawlers reduce crawl authenticity and that mobile browsers are more prone to fingerprinting.

2.5 Distinction with Prior Work

Our work introduces a hybrid analysis framework that integrates both static and dynamic techniques to uncover divergent fingerprinting behavior—cases where identical JavaScript executes differently across platforms (e.g., on mobile but not on desktop). This dual-layered approach allows us to not only identify structural fingerprinting logic via static analysis but also confirm its execution through runtime monitoring. For dynamic analysis, we leverage VisibleV8 [25] that offers fine-grained, API-level tracing by logging every JavaScript API and function call directly within the browser engine for *both* desktop and mobile platforms. We align API call offsets from VisibleV8 (dynamic) with enriched AST offsets from JStap (static) to map executed APIs to the program dependence graph and construct execution graphs. Unlike prior studies—such as Cassel et al. [11]—that rely on JavaScript instrumentation or proxies and monitor only a limited subset of APIs, VisibleV8 provides comprehensive coverage and deep visibility into JavaScript behavior. Its tight integration into the V8 engine also makes it more stealthy and less susceptible to detection by anti-bot mechanisms.

While our system is limited to Chromium-based browsers, this scope is justified by current market trends: Chrome alone holds 67.08% of the global browser market, with Edge and Opera bringing the total Chromium-based usage to over 74.39% [53]. In contrast, Firefox's 2.54% share indicates that our analysis captures the dominant share of user-facing web activity. In summary, our work differs from previous efforts by offering: (1) a *broader and deeper view* of tracking activity via high-fidelity instrumentation, and (2) a *hybrid analysis workflow* that enables robust detection of platform-specific divergences in script behavior—an area largely underexplored in existing literature.

3 Formalizing Execution Flow Divergence

When JavaScript executes on different platforms, such as mobile and desktop, variations can arise due to differences in the runtime environment, API availability, and execution order. Although the same script may be loaded, its execution sequence can diverge due to asynchronous operations, platform-specific optimizations, or conditional execution paths. We refer to a sequence of executed APIs with their character offsets (i.e., location in code) as an *execution trace*. To systematically quantify divergences, we define a metric that captures both missing and reordered API calls in the execution trace while allowing for minor execution variations.

Our approach models this problem as a sequence alignment task, similar to edit distance computations, where one sequence represents the execution flow on a mobile device, and the other represents the execution flow on a desktop. Each sequence is an ordered list of API calls, where every entry consists of an execution offset (character position within the script) and an API identifier. Given two such sequences, we construct a *dynamic programming* (DP) algorithm to measure their dissimilarity while allowing for slight reordering and gaps in execution.

Let F_m and F_d represent the API execution flows observed on mobile and desktop, respectively:

$$F_m = [(o_1^m, a_1^m), (o_2^m, a_2^m), \ldots, (o_N^m, a_N^m)] \quad (1)$$

$$F_d = [(o_1^d, a_1^d), (o_2^d, a_2^d), \ldots, (o_M^d, a_M^d)] \quad (2)$$

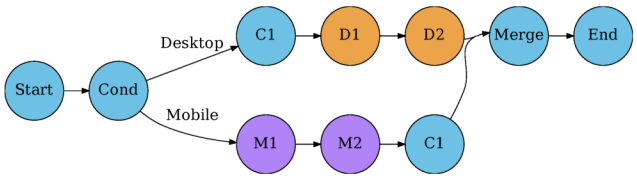

Figure 1: A graphical representation of API execution trace with a platform-dependent branch. Orange, purple, and blue nodes denote desktop-only, mobile-only, and common APIs executed, respectively.

where o_i denotes the character offset within the script, and a_i denotes the API call at that offset.

To compute divergence, we define a cost matrix D of size $(N+1) \times (M+1)$, where N and M represent the cardinality of F_m and F_d, respectively, and $D[i][j]$ represents the minimum cost to align the first i elements of F_m with the first j elements of F_d. The goal is to minimize the cost incurred by mismatches, missing APIs, and significant reordering.

Figure 1 presents an example API execution trace of a script exhibiting platform-dependent branching. The *Cond* node preceds the divergence point, and in this particular example, is the last commonly executed node. On the desktop path, nodes *C1*, *D1*, and *D2* are executed before merging. On mobile, *M1* and *M2* precede *C1*, which appears as a re-ordered node. To account for this partial alignment, our algorithm assigns a half-cost adjustment to *C1*.

Listing 1 presents a real-world example of divergence due to device-specific behavior. The variable `m` (defined in line 7) acts as the *conditional node* (line 10) in the PDG, executed on both platforms before divergence begins on the desktop. The first exclusive node executed on the desktop marks the *divergence* point.

We define "identical scripts" as source-code scripts loaded from the same domain on both platforms. Each is compared one-to-one per domain, and if it appears multiple times, divergence scores are averaged. A "divergence point" is the first exclusively executed API passed to the conditional algorithm, marking the start of a "divergent subsequence" (or "divergent trace")—a platform-specific sequence of API calls. A script may contain multiple such subsequences. While we do not track implicit flows, our information flow analysis captures APIs, functions, and variables in the PDG that explicitly influence the conditional node. This information flow chain refers to the list of PDG nodes contributing to the conditional node. If the conditional node includes only browser APIs, it alone represents the information flow.

The divergence algorithm has the following steps:

❶ **Initialization.** The first row and first column of D are initialized to represent the cost of aligning a sequence with an empty prefix:

$$D[i][0] = i \cdot g, \quad D[0][j] = j \cdot g \tag{3}$$

where g is a gap penalty applied when an API call is missing in one of the sequences.

❷ **Matching and Misalignment Handling.** For each pair (o_i^m, a_i^m) and (o_j^d, a_j^d):

- If they are an exact match (i.e., same API at the same offset), no penalty is applied:

$$D[i][j] = D[i-1][j-1] \tag{4}$$

- If the API call is found within a small offset window k, but at a different position, we apply a misalignment penalty p instead of

```
1  ft.isM = function(p, t) {
2    return !!p && p === "iPhone" || p === "iPad" || (p.substr
3    (0, 7) === "Linux a" && t > 0)
4  };
5  s.D9_120 = navigator.platform;
6  s.D9_123 = navigator.maxTouchPoints || 0;
7  var m = ft.isM(s.D9_120, s.D9_123);
8  s.D9_130 = ft.flashVersion(m);
9  ft.flashVersion = function(m) {
10   if (m) { return null }
11   try {
12     var obj = new ActiveXObject("ShockwaveFlash.
13     ShockwaveFlash.6");
14     try {
15       obj.AllowScriptAccess = "always"
16     } catch (e) { return "6.0.0" }
17   } catch (e) {
18     if (navigator.mimeTypes["application/x-shockwave-flash"].
19       enabledPlugin) {
20       return navigator.plugins["Shockwave Flash 2.0"] }}};
```

Listing 1: Alternative code execution behavior based on platform environment from *d9.flashtalking.com/d9core* from adobe.com.

treating it as a completely missing API: [1]

$$\exists j' \in [j-k, j+k] \text{ such that } (o_i^m, a_i^m) = (o_{j'}^d, a_{j'}^d) \Rightarrow D[i][j] = D[i-1][j'-1] + p \tag{5}$$

This ensures that small reordering variations due to asynchronous callbacks and network requests do not lead to excessive divergence penalties.

- If no match is found, a gap penalty g is applied to account for missing APIs:

$$D[i][j] = \min(D[i-1][j] + g, D[i][j-1] + g) \tag{6}$$

❸ **Divergence Score Computation.** The final divergence score is normalized by the maximum sequence length to ensure comparability across different scripts:

$$\delta(F_m, F_d) = \frac{D[N][M]}{\max(N, M)} \tag{7}$$

A higher value of δ indicates greater divergence between mobile and desktop execution.

This approach balances sensitivity to execution differences with robustness to minor reordering or missing API calls. Using a lookahead window and misalignment penalties, it captures meaningful divergences while dynamic programming ensures efficient comparison at scale.

4 Methodology

We present a hybrid framework to analyze how identical JavaScript code exhibits divergent behavior across platforms. As illustrated in Figure 2, we combine static analysis of program structure (ASTs and PDGs) with dynamic execution traces from VisibleV8 [25]. We first build syntax graphs using JStap, a static analysis tool that builds a PDG over an AST. We extend it with updated libraries (e.g., replacing Esprima [2] with Espree [3]) and additional features to support our analysis. Next, using execution traces from VisibleV8, we mark PDG nodes with matching offsets as executed, based on API calls recorded by the JavaScript engine. We refer to the resulting combination of the PDG and execution trace as the *execution graph*.

[1] We empirically assign the following values to the parameters: $p = 0.5, g = 1, k = 5$
[2] https://esprima.org/
[3] https://github.com/eslint/js/tree/main/packages/espree

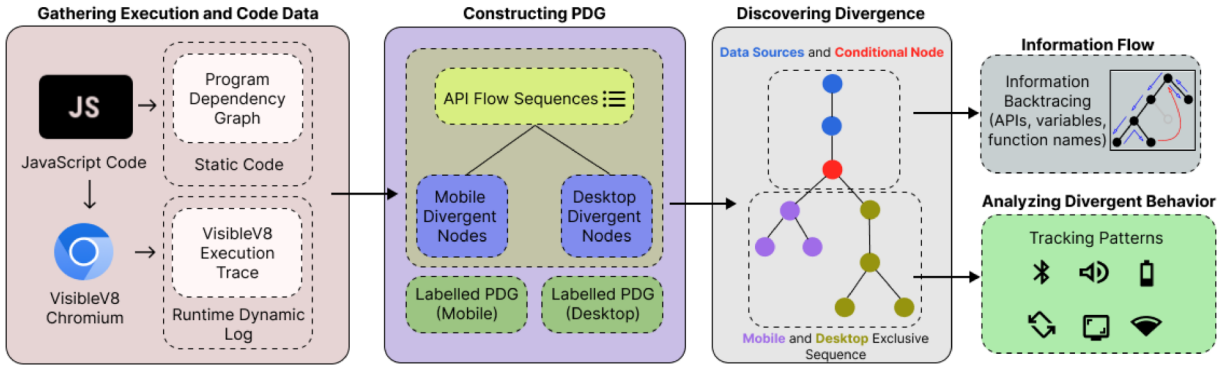

Figure 2: System overview. We combine dynamic execution traces with static PDGs to create a labeled execution graph (PDG). Conditional nodes responsible for divergence are identified, along with their data sources via the iFlow analysis. We then perform: (i) pre-divergence analysis to detect fingerprinting APIs used to infer device properties, and (ii) post-divergence analysis to categorize tracking behaviors in platform-specific execution paths.

This enriched graph allows us to identify conditional branches and trace the data sources (`iFlow` chains) that influence divergence.

Listing 1 shows a representative case from `flashtalking.com` on `adobe.com`, where the script detects mobile devices using `navigator.platform` and `navigator.maxTouchPoints`. On mobile, it skips Flash detection; otherwise, it checks Flash support via `navigator.mimeTypes`, `ActiveXObject`, and `navigator.plugins`.

Execution traces reflect this divergence. On mobile, only high-level `navigator` properties are accessed; on desktop, Flash-related APIs are additionally invoked. Our framework detects the divergent node `navigator.mimeTypes`, locates its call site (line 18), and traces the control condition `m` (line 10) back to its dependencies—revealing that `navigator.platform` and `navigator.maxTouchPoints` are used to classify the platform. This illustrates how our system links execution divergence to specific browser APIs, exposing their role in platform detection and potential fingerprinting.

4.1 Hybrid Analysis

As previously discussed, our hybrid analysis approach leverages static analysis to generate a graphical representation of JavaScript code, which is then enriched by labeling nodes corresponding to APIs executed at runtime. The PDG is built on top of AST which includes all method calls across script functions. Dynamic traces from VisibleV8—extended to capture runtime executions of all WebIDL-defined and global object APIs—are used to annotate nodes in the PDG of the corresponding method calls.

In an ideal case, a script executed on both mobile and desktop platforms would yield identical execution graphs, with matching sets of labeled nodes. However, in practice, scripts often incorporate logic to detect device or hardware characteristics and selectively execute platform-specific code paths. While such behavior can be benign—such as adapting the website layout for mobile users by resizing icons, images, or text—it may also introduce opportunities for device-specific fingerprinting. Accordingly, we analyze these divergences not only to understand functional adaptations but also to identify any potential privacy-invasive tracking mechanisms embedded within the scripts.

4.1.1 Divergence Algorithm. To identify points of divergence in script behavior, we compare execution traces from mobile and desktop platforms and use VisibleV8 to pinpoint the offset of the first differing API call. Inspired by BFAD [54], we align this dynamic information with static analysis from JStap [17], which provides an enriched AST with token offsets. By matching offsets from both tools, we map divergent API executions to nodes in the program dependence graph (PDG) and construct execution graphs. To reduce noise, we filter out traces with fewer than five API calls. This ensures our analysis focuses on meaningful behavioral differences.

4.1.2 Conditionality Analysis. The role of our *condition detection algorithm* is to first confirm whether the divergence stems from a conditional statement and, if so, to identify the specific conditional construct responsible for the variation in behavior. In program execution, such divergence often arises from conditional branching or asynchronous code execution. Our condition detection algorithm performs static analysis on the Program Dependence Graph (PDG).

The algorithm begins by taking a node in the PDG as input and then traversing upward to locate the nearest condition-related parental statement (e.g., *LogicalExpression*, *ConditionalExpression*, or *IfStatement*). If the parental statement is direct, meaning the divergent node lies within the body of an `if` statement, it simply returns that parental statement as the cause of divergence. However, if the relationship is indirect, such as when the divergent node resides in an `else` block, we instead return the parental statement preceding the `else` branch.

In a PDG, a single node may have multiple condition-related parental statements that govern its execution. Our algorithm selects the closest one because the divergent node represents the first API call that deviates from VisibleV8's execution trace. If a higher-level parental statement was responsible for the divergence, we would instead observe a different divergent node—one that executes immediately after that higher-level condition and before the current node. This approach ensures precision in identifying the root cause of behavioral differences. Note that it is common to put multiple conditions with similar purpose together in one statement through logical operator (e.g., && and ||). In this case, although there is only one condition that actually causes the divergence, the condition detection algorithm returns all conditions in that statement to allow further analysis on more divergent-related conditions.

4.1.3 Information Flow Analysis. In real-world JavaScript, it is common for conditional logic to rely on intermediate boolean variables. These variables may be the result of complex logic, and with the widespread use of JavaScript bundlers (e.g., Webpack [4]), minification, and obfuscation tools, variable names often lose their semantic clarity. As such, the information flow analysis is tasked with tracing all possible browser APIs or string literals that contribute to the value of the boolean variable used in the conditional statement.

The information flow algorithm begins by analyzing a conditional statement node in PDG given by conditionality analysis. It first identifies all relevant symbols (e.g., variables and functions) that directly contribute to the conditions through assignments, function definition, or function calls. The algorithm then recursively expands its analysis to trace all possible information flow, uncovering symbols that influence these initial contributors and contributors from previous recursion. This recursive process continues until no additional contributing symbols can be found, at which point the algorithm returns the full set of involved symbols. If a conditional statement given by previous analysis is a browser API, we directly return the browser API since this API gets information from browser. When the conditional detection algorithm returns multiple conditions from the same statement, we run the information flow algorithm on all of them individually.

In our framework, *information sinks* are variables identified within conditional statements—i.e., outputs of the condition detection phase. Our goal is to trace back to all *information sources*—APIs or constants—that influence these variables. This backward analysis is performed over the JStap-generated data flow graph [1].

5 Data Collection

This section outlines our experimental setup for data collection across both mobile and desktop platforms.

5.1 Device Setup and Crawl

Our crawling infrastructure consisted of two distinct environments: a real mobile device (Google Pixel 4a) and a desktop machine equipped with 128 GB RAM, a 32-core CPU, and running Ubuntu 24.04. Both environments used Chromium version 128 to ensure consistency in browser capability across platforms.

Automation of the crawling process was handled using Puppeteer [43] in both settings. For the desktop setting, we instrumented a headless Chromium instance with VisibleV8 to capture fine-grained JavaScript execution traces. To minimize detection by anti-bot mechanisms and ensure realistic execution, we used a community-maintained stealth plugin [8]. We discuss the implications of using headless vs. headful settings for desktop in § 9.

For each domain, we imposed a hard timeout of one minute, during which the page was fully loaded and allowed to execute. This timeout was uniformly applied across both environments and was sufficient to ensure that the majority of page resources had loaded and relevant JavaScript activity was captured before advancing to the next domain. To ensure consistency and minimize temporal discrepancies in web content, we synchronized our crawls such that each website was visited simultaneously on both platforms. This concurrent execution allowed for a controlled comparison of JavaScript behavior, enabling us to attribute any observed differences in execution solely to platform-specific factors rather than variations in content over time.

We employed *mitmproxy* [2] to capture network traffic during crawling, facilitating additional validation of resource loads and execution behaviors. While the logs produced by VisibleV8 sufficiently capture all necessary JavaScript execution data, we use *mitmproxy* logs to determine the successful loading of the webpage by looking for completed network requests in fetching the main document of the website.

Our data collection process began on 03 November 2024, lasting seven days. During this period, we crawled the top popular 10K websites from the Tranco list [29, 57]. The failed domains were revisited. Out of 10K, we successfully crawled and gathered script execution traces from 7,811 websites on both platforms. The persistently failing websites were either popular Content Delivery Networks (CDNs) or other non-public-facing endpoints.

5.2 VisibleV8 for Android

To enable our analysis on mobile platforms, we ported the VisibleV8 patch set, originally developed for desktop Chromium, to run on Android. A key challenge was overcoming Chromium's default system call restrictions, which, enforced by Android's sandboxing, prevent applications like VisibleV8 from writing log files directly to the file system. We successfully modified the patch set to bypass these restrictions, enabling comprehensive logging of JavaScript execution without compromising the integrity of the Android security model. We have contributed our mobile-specific patch set upstream to the open-source VisibleV8 project [3].

6 Prevalence of Divergence

In this section, we address **RQ1: What is the prevalence of divergence in the execution of JavaScript on mobile and desktop platforms?** We begin by offering a high-level overview of the prevalence and distribution of divergently executed JavaScript (identically sourced) across the Tranco top 10K websites. Our analysis focuses on identifying and characterizing differences in script execution between desktop and mobile environments. To support this investigation, we develop a system that detects divergent scripts—those uniquely executed on one platform but not the other — using an edit-distance metric introduced in Section 3. This metric allows us to quantify divergence by comparing the dynamically recorded API traces of scripts across platforms.

6.1 Divergent JavaScripts

We examine the divergence in JavaScript execution between mobile and desktop environments by analyzing dynamic execution traces collected using VisibleV8. Divergence was computed using the algorithm introduced in Section 3, and throughout the paper, we refer to the resulting metric (i.e., *divergence score*) as the percentage of divergent activity. By aligning subsequences of API calls across platforms, we identify points of divergence occurring at the boundaries of these aligned sequences. We then trace these divergent endpoints back to their corresponding conditional nodes, which are further analyzed through information flow tracing to determine whether device-specific data influenced the execution path.

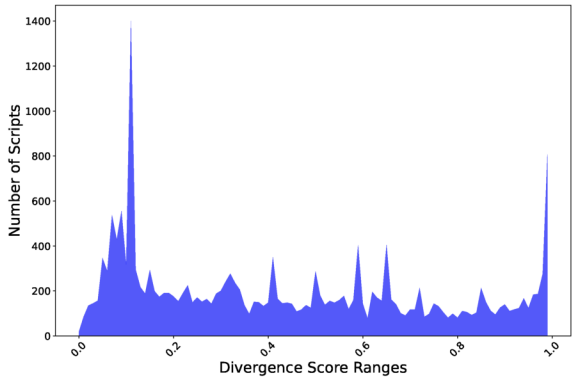

Figure 3: Distribution of divergence scores for uniquely executed JavaScript files across mobile and desktop platforms in the Tranco top 10K websites. Out of the 109,071 total scripts, 22,495 had a non-zero divergent activity.

From our crawl of the Tranco top-10K websites, we identified 109,071 unique JavaScript resources fetched in both desktop and mobile contexts. For each resource S_i, we compared its desktop ($S_{i,d}$) and mobile ($S_{i,m}$) instances to compute a divergence score. Since many scripts originate from shared libraries or third-party services—whose behavior may vary across websites—we averaged divergence scores across all domains where a script appears, producing a single representative value per script. Figure 3 shows the distribution of these scores, with values binned at 0.01 intervals and scripts with zero divergence excluded. After filtering, 22,495 scripts exhibited some level of divergent behavior, providing a consistent measure of script-level variation across platforms.

The script hosted at *https://www.google-analytics.com/analytics.js* was the most frequently encountered in our dataset, appearing on 1,309 websites. On average, it showed a divergence score of 15.6%, reflecting moderate platform-specific behavior. We also observed scripts with divergence scores nearing 1, indicating highly distinct execution paths between platforms. These high-divergence scripts were primarily Webpack-bundled modules or jQuery libraries, often customized to deliver platform-adaptive functionality.

Findings 1. Of 109,071 identically sourced JavaScript files executed during our crawl of the Tranco top 10K websites, 20.6% (22,495) exhibited divergent behavior between mobile and desktop platforms.

6.2 Conditioned Divergence

It is important to note that not all divergent scripts necessarily contain a *conditional node* that explicitly alters code execution. Divergent behavior may also arise from factors such as server-side logic, asynchronous callbacks, or network-dependent responses—cases where no clear client-side conditional node will exist. In our analysis, however, we narrow our focus to client-side scripts that explicitly alter the execution path through conditional logic. Our goal is to uncover instances where scripts make device-specific decisions that lead to divergent behavior. To achieve this, we leverage both dynamic execution traces from VisibleV8 and static representations of the code in the form of a Program Dependency Graph (PDG), enabling us to identify nodes directly responsible for divergence. The presence of such conditional nodes indicates that the script has branched into a device-specific execution path.

As described in Section 4, for each conditional node identified, we apply the information flow algorithm to find the function names, variables, and APIs passed along the execution trace leading up to the conditional node. This information flow chain is significant as it helps us understand whether device property information has propagated through the execution path to reach the conditional node and enables the script to execute a platform-dependent code path. In the example shown in Listing 1, the analysis of dynamic API traces from VisibleV8 reveals that `Navigator.mimeTypes` is the first divergent node to execute. Here, the conditional node is the variable 'm', which likely holds critical device information (such as `navigator.platform` and `navigator.maxTouchPoints`) allowing the script to detect a touch device and make a branching decision.

Table 1 presents a summary of the unique nodes, conditional nodes (responsible for exclusive execution), and information flows (associated with each conditional node). It should be noted that while conditional nodes are commonly executed in both mobile and desktop, in the context of divergence, a satisfied conditional node is the node that is responsible for the execution of a divergent subsequence in a platfom-specific style. To ensure that we are truly examining unique subsequences, we ensure that none of the APIs present in the unique subsequence in one platform device have executed in the other device's execution throughout the script's execution lifetime. In our analysis, only 849 of 75,148 divergent conditions used platform-identifying APIs (e.g., `userAgentData.platform`). Most divergence came from performance (5,201), plugin (1,153), and screen (1,793) APIs, indicating browser- rather than OS-level branching.

A key insight from our analysis is that desktop environments exhibit a substantially greater number of unique execution nodes compared to mobile platforms. Specifically, we identified 47,101 divergent subsequences on desktop, in contrast to 28,047 on mobile. When applying our conditional node detection algorithm, we found that 19,400 of the desktop subsequences and 11,581 of the mobile subsequences could be linked to specific conditional nodes.

Likewise, the number of information flows (i.e., *iFlow*) — representing data propagated to these conditional points—was also higher on desktop (17,908) than on mobile (10,850). Our information tracing algorithm successfully determined information flow chains on almost 92.8% of the conditional nodes associated with the divergent subsequences. We note that our algorithm failed to account for all of the conditional nodes. Our algorithm relies on data flow edges added by JStap to the AST. It fails when JStap misses edges, hindering backtracking. Upon closer inspection of certain failed iFlow executions, we observed that the JStap-generated representations did not consistently capture accurate data and control dependencies within the code. An example case where JStap misses edges is provided in Appendix 10. We adapted our algorithm to work directly on the AST to address test case issues and real-world scripts from experiments, but fully resolving them would require extensive manual effort to identify missing edges in complex real-world JavaScript. Additionally, some divergence arises from timer APIs (e.g., setTimeout), which our algorithm cannot capture. Since our iFlow backtracking algorithm relies on these dependencies to uncover data sources that feed the conditional node, inaccuracies in the static representation hinder its overall effectiveness. This highlights a key limitation of our approach and reflects a well-known

Table 1: Summary of divergent subsequences, conditional nodes, and information flows identified from the analysis of 109,071 identically sourced scripts executed across mobile and desktop platforms. Note: While conditional nodes are commonly executed in both mobile and desktop, in the context of divergence, a satisfied conditional node is the node that is resonsible for the execution of a divergent subsequence in a platfom-specific style.

Metric	Mobile	Desktop	Overall
Divergent Subsequences	28,047	47,101	75,148
Satisfied Conditional Nodes	11,581	19,400	30,981
iFlows	10,850	17,908	28,758

challenge in performing precise static analysis on JavaScript, given its dynamic and loosely typed nature.

These findings suggest that approximately 41% of the divergent subsequences can be attributed to explicit conditional logic responsible for platform-specific branching. It is important to note, however, that these figures represent lower-bound estimates, constrained by the precision and coverage of our conditional node detection algorithm. Additionally, due to JavaScript's inherently dynamic nature, divergence may also arise from other sources—such as asynchronous operations or network-dependent callbacks—that may not manifest as directly traceable subsequences in our analysis.

Overall, our findings highlight that while both mobile and desktop platforms share a significant portion of JavaScript execution, the desktop environment exhibits a higher degree of unique execution paths. These differences have implications for security analysis, fingerprinting techniques, and script detection methodologies, as platform-specific execution behaviors must be accounted for when assessing the behavior of JavaScript in the wild.

Findings 2. JavaScript execution divergence is 67.9% more pronounced on desktop than mobile, indicating greater platform-specific behavior. Approximately 41% (30,981/75,148) of divergent traces stem from explicit client-side conditional logic.

7 Data-Driven Divergence

In this section, we look to answer **RQ2: What information do JavaScript programs use to trigger divergence?** To understand what information JavaScript programs use to trigger platform-specific divergence, we analyze the conditional nodes responsible for branching behavior. For each such node, we examine its information flow chain—a sequence of functions, variables, and API property accesses that contribute to the branching decision. This approach allows us to identify the specific platform-related inputs (e.g., user-agent, screen size) that scripts use to differentiate execution paths.

7.1 Device Information Propagation

Information flow tracking allows us to assess whether specific API calls contribute to execution divergence across platforms. The presence of fingerprinting APIs within these flows is particularly notable, as it indicates that scripts may be leveraging device-specific characteristics to dynamically alter their behavior. To explore this, we examine the information flows leading to conditional nodes for the presence of known fingerprinting APIs [54].

Table 2: Summary of information flows to conditional nodes and top 10 contributing APIs on mobile and desktop.

Metric / API	Mobile	Desktop	Total
Unique Subsequences	28,047	47,101	75,148
iFlow Chains	10,850	17,908	28,758
Fingerprinting APIs in iFlows	8,323	13,132	21,455
Window.performance	1340	1970	3310
Window.navigator	749	2254	3003
Window.getComputedStyle	1384	799	2183
Window.location	581	1367	1948
MutationRecord.target	394	896	1290
Navigator.userAgent	349	852	1201
Window.window	185	1035	1220
Navigator.plugins	29	1030	1059
Performance.now	359	547	906
PluginArray.length	0	842	842

Table 2 provides a detailed breakdown of our analysis, including the number of uniquely executed subsequences, total information flow (iFlow) chains extracted, and the frequency of fingerprinting APIs within these flows. In addition to these summary metrics, the table lists the top 10 APIs that most frequently appeared in the iFlow chains feeding into conditional nodes—points at which execution branches based on platform-specific logic.

Specifically, 21,455 information flows contained at least one fingerprinting API, indicating that approximately 74.6% of conditional nodes with a derived iFlow chain involved device-specific information at the point of divergence. This is important because when a fingerprinting API appears in the execution chain of a unique node, it implies that the script has acquired critical device information, which may influence its decision to execute a different code path based on the detected platform. Such behavior is common in scripts designed for targeted content delivery, feature adaptation, or even evasion techniques in web tracking.

The most prevalent contributors include `Window.performance`, `Window.navigator`, and `Window.getComputedStyle`, reflecting their widespread use in runtime introspection and layout adaptation. Interestingly, several APIs such as `Navigator.plugins` and `PluginArray.length` are overwhelmingly desktop-specific, likely due to their limited support on mobile devices and their stronger association with fingerprinting. Conversely, `Window.getComputedStyle` appears more in mobile flows, indicating its utility in adjusting UI logic based on rendering and layout differences across devices.

Other commonly observed APIs—such as `Window.location`, `MutationRecord.target`, and `Navigator.userAgent`—appear across both platforms and serve as foundational signals for gathering contextual device or DOM state prior to divergence. The inclusion of `Performance.now` further points to timing-based heuristics as part of the decision-making process.

While examining iFlow chains, we observed that certain variable and function names—such as those containing "android" or "isMobile"—were suggestive of platform-related context being stored or processed. This highlights a strength of our information flow algorithm: its ability to perform fine-grained data tracing that surfaces implicit information sources beyond explicit API calls.

Overall, our findings suggest that a significant portion of execution divergence can be attributed to fingerprinting mechanisms embedded within JavaScript execution flows. The heavy use of environment-exposing APIs prior to divergence supports the view that scripts tailor their logic dynamically in response to runtime conditions. These insights are crucial in understanding how scripts differentiate between devices and can be leveraged to detect covert tracking mechanisms or platform-specific optimization strategies.

Findings 3. Approximately 76% (21,455/28,758) of conditional nodes with derived information flows were found to reference fingerprinting APIs, indicating that the majority of divergence in JavaScript execution is influenced by device-specific characteristics.

7.2 Detection of Novel Information Sources

While established fingerprinting techniques often rely on a well-known set of APIs, our analysis of conditional branching reveals that platform-specific divergence is not always driven by these known API interfaces alone. Our analysis from the previous subsection revealed that 7,303 (25.4%) of conditional nodes (associated with unique subsequences) had not acquired device information through known fingerprinting APIs. This presents an opportunity to uncover lesser-known APIs that scripts may use to detect device type and enforce platform-specific behavior. While not commonly linked to fingerprinting, these APIs can still reveal details about device configurations.

In this subsection, we focus on API accesses within the information flow chains of conditional nodes that do not intersect with the canonical fingerprinting API set. We identify and aggregate top 100 most frequently accessed APIs in these chains, specifically those preceding branches where exclusive platform-specific logic is executed. To better understand their utility, we categorize these APIs by general feature usage and analyze how each feature may enable scripts to infer device characteristics.

Table 3 presents a summary of the distribution of feature categories identified in information flows that did not contain fingerprinting APIs, along with representative API examples. In total, we identified 1,748 information flows utilizing these lesser-known APIs, which may provide insights into how scripts deduce the underlying platform. Below, we outline how each of these feature categories can be used to uncover device-specific properties.

DOM Manipulation. Scripts use DOM manipulation APIs to inspect or dynamically alter page structure. Variations in the rendered DOM across platforms, such as element availability or layout behavior, can reveal device characteristics and inform platform-specific logic. *This feature was the most frequently observed, with 3,201 instances on desktop and 2,445 on mobile, highlighting its central role in cross-platform differentiation.*

Event Handling. Event-related APIs help scripts determine which input modalities are supported. For instance, the presence of touch versus mouse events can indicate whether the script is running on a mobile or desktop device, guiding adaptive interaction logic. Listing 2 demonstrates how `touchstart` event can trigger a signal for the script to determine that the device likely supports touch. We empirically evaluate the touch-based events from the dynamic logs to ascertain such events. Listing 9 in Appendix A shows a code

```
window.addEventListener("touchstart", () => {}, { once: true });
const isMobile = "ontouchstart" in window;
```

Listing 2: `addEventListener` being used to detect device type

snippet from a script that combines touch-event signals with a CSS media query for a 'coarse' pointer—typical of touch devices—to detect touch support. *Event handling was significantly more frequent on desktop (1,997) than on mobile (444), suggesting that event-modality checks can be initiated from desktop scripts attempting to detect mobile contexts.*

Custom API Interfaces. Scripts often access analytics-related global objects such as `gaData` or `gaplugins`—custom interfaces exposed by services like Google Analytics. These interfaces may encode telemetry or plugin metadata that varies by platform, indirectly revealing the device type or environment. For example, certain plugins may be selectively loaded on mobile, or identifiers within the analytics context may reflect platform usage. *These interfaces were observed in 756 desktop and 354 mobile flows, underscoring their utility as indirect signals for third-party scripts to infer device characteristics.*

Timing APIs. High-resolution timing APIs expose performance characteristics such as input latency or page load timing. These metrics can be used to detect slower execution environments typical of mobile devices and conditionally trigger lightweight or deferred logic. *Timing APIs were accessed in 495 desktop flows and 257 mobile flows, indicating their use in indirectly inferring platform performance characteristics.*

Viewport Observation. Observation APIs, such as `ResizeObserver`, enable scripts to monitor changes in layout or screen dimensions. Variations in screen size or orientation can be leveraged to infer device type and load platform-specific components. *Although these APIs are used less frequently overall, the distribution (217 desktop flows vs. 35 mobile flows) suggests they are primarily employed to detect larger screen environments, such as desktops, which offer greater flexibility for window resizing.*

Performance Monitoring. Scripts may use navigation and rendering performance APIs to assess the speed and responsiveness of the environment. Significant discrepancies in these metrics across platforms allow detection of mobile contexts. *With 139 desktop and 92 mobile observations, these APIs likely serve as complementary indicators of device capabilities.*

Device Fingerprinting. APIs that expose detailed device or browser properties are directly used to fingerprint the environment. Detection of touch capabilities, pointer types, or platform-specific identifiers provides clear signals for device classification. *Despite lower frequency (85 desktop and 74 mobile), their targeted use suggests scripts invoke them when a confident platform distinction is needed.*

Storage Access. Differences in the availability or behavior of storage mechanisms like `localStorage` can signal platform constraints, such as limited quota or cross-context restrictions on mobile, guiding adaptive script paths. *This feature appeared in 54 desktop and 36 mobile flows, indicating niche but meaningful use in detecting mobile platform limitations.*

Table 3: Summary of novel API feature usage in desktop and mobile iFlow chains with representative examples

API Feature	Desktop	Mobile	Examples
DOM Manipulation	3,201	2,445	HTMLDocument.createElement, HTMLDocument.querySelector
Event Handling	1,997	444	Window.addEventListener
Custom API Interfaces	756	354	Window.gaData, Window.gaplugins
Timing APIs	495	257	PerformanceNavigationTiming.activationStart, Window.google_measure_js_timing
Viewport Observation	217	35	Window.ResizeObserver
Performance Monitoring	139	92	PerformanceNavigationTiming.startTime,
Device Fingerprinting	85	74	Window.isMobile, Navigator.msPointerEnabled
Storage Access	54	36	Window.localStorage
Media Queries	142	97	HTMLVideoElement.canPlayType
Information Flows	**1,003**	**745**	

```
const isAndroid = (
  video.canPlayType('video/webm; codecs="vp9"') === "probably" &&
  video.canPlayType('video/mp4; codecs="hvc1"') !== "probably");
```

Listing 3: Example of `HTMLVideoElement.canPlayType` being used to infer availability of Android video codec.

Media Queries. APIs that test media capabilities enable scripts to infer support for specific content formats. As media codec support and rendering constraints differ between devices, these tests inform decisions on content delivery and layout. Listing 3 shows how video codec availability may signal Android environments. *This feature appeared in 142 desktop and 97 mobile flows, reinforcing its role in content-specific adaptation.* A representative example of media format enumeration used as a source of divergence is shown in Listing 8 in Appendix A.

By isolating the information flows that did not contain known fingerprinting APIs, we uncover the lesser-known or novel APIs that may be contributing to fingerprinting or device inference. This exploratory analysis provides a foundation for identifying emerging fingerprinting vectors that scripts can use to determine device properties before executing exclusive code paths.

Findings 4. Our analysis reveals that 6% of divergent execution paths (that are enabled by conditional nodes) are influenced by lesser-known APIs not traditionally associated with fingerprinting. These APIs—spanning DOM manipulation, event handling, and performance monitoring—appear in over 1,700 information flows, suggesting they play a significant role in enabling scripts to infer platform characteristics and trigger device-specific logic.

8 Post Divergence

In this section, we analyze the divergent subsequences of JavaScript executions on mobile and desktop platforms to uncover platform-specific behaviors exhibited by these scripts. We place particular emphasis on examining exclusive API accesses to assess whether scripts engage in more tracking-related activities on one platform compared to the other. Broadly, this analysis addresses **RQ3: What do platform-specific JavaScript execution paths accomplish?**

Various tracking techniques can be commonly used for purposes for benign purposes such as delivering seamless web experience adjusted for varying platform environments. These may also be used for fraud detection and cross-site identification. However, from a privacy standpoint, it is important to understand how the same scripts behave differently when executed on desktop versus mobile devices. Our goal is to uncover disparities in tracking activity when users switch between these platforms.

To analyze the imbalance in script behavior across platforms, we investigate execution traces of JavaScript segments that are common to both mobile and desktop environments but contain subsequences that execute exclusively on one. In Section 6, we identify these divergent scripts, pinpoint their divergence points, and trace the flow of platform-dependent information. In Section 7, we further examine the underlying data sources that may contribute to this divergence. Building on this foundation, we now focus on the platform-specific API calls that occur in the post-divergence execution paths of the scripts identified from Section 6. Our goal is to identify tracking-related patterns that emerge uniquely on each platform. Specifically, we examine the exclusively executed code from the divergence point to where the flows merge back. We begin by categorizing fingerprinting and tracking operations observed in these exclusive subsequences. Then, we conduct in-depth case studies of divergent scripts to better understand the nature and intent of their platform-specific behavior.

8.1 Behavior in Divergent Subsequences

To categorize the tracking operations present in the platform-specific subsequences, we leverage the set of fingerprinting APIs—combining previously known techniques from prior literature [13, 15, 23, 24, 44, 54] and industry-grade fingerprinting solutions such as FingerprintJs [12] with the newly identified APIs uncovered in Section 7.2. Using this expanded set, we construct tracking heuristics, following the methodology of prior work [11, 15], to analyze the distribution of tracking behaviors across platforms. This allows us to create 8 different categories for understanding the behavior of exclusive activities. These categories are highlighted in Figure 4 that illustrates the tracking activities that occur exclusively on each device, with the numbers atop each bar indicating the number of domains where these activities were observed. Overall, we find that desktop-exclusive executions lean on high-entropy, persistent identifiers like fingerprinting, storage and network APIs. In contrast, mobile-exclusive executions lean on event-driven behavrioal signals like touch and swipe events. Below, we provide a summary of our observations for each of these categories.

Browser Fingerprinting. Scripts can attempt to derive a unique device signature using low-level APIs. This includes the use of canvas rendering for pixel-based identifiers, font rendering to detect

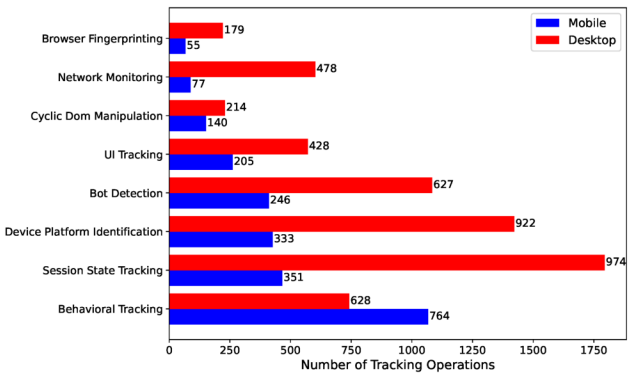

Figure 4: Tracking activities in the exclusively run code subsequences for each device. Numbers that proceed each bar represent the number of domains on which these tracking operations were applied in a platform-specific fashion.

available fonts, WebGL parameter enumeration for graphics hardware profiling, audio APIs for subtle signal differences, and image decoding or drawing routines. Divergent execution of fingerprinting was *more pronounced on desktop*, with 221 exclusive operations versus 67 on mobile. This disparity likely stems from the richer and more consistent API surface on desktop platforms, making them more suitable for high-entropy fingerprint generation. Mobile devices, by contrast, often restrict access to these APIs or produce less varied results across devices. For instance, there were 18 instances of exclusive accesses of *WebGLRenderingContext.getExtension* in desktop and none in mobile. One plausible reason for this bias is that bot traffics are dominantly hosted on desktop environments and WebGL extensions are a high-quality signal for bot detection. [4]

Device and Platform Identification. Scripts can access APIs that expose device-specific or environment-level properties. These include querying user agent strings, hardware concurrency, battery status, peer connection metadata, and time zone information—all of which are commonly used to infer platform characteristics. This form of tracking was observed *more on desktop* (1,424 exclusive operations vs. 422 on mobile). This higher usage on the desktop is a strong indicator that scripts may be leveraging detailed platform and device information to optimize rendering performance, tailor functionality, or adjust resource loading strategies—especially in more complex desktop environments where device variability and multi-threading capabilities are higher (e.g., more entropy in screen sizes and CPU cores).

Behavioral Tracking. Scripts monitor user interactions, such as mouse movement, key presses, touch gestures, scrolling behavior, and event timestamps as a form of behavioral tracking. These interactions help construct detailed behavioral profiles of users. Behavioral tracking was *disproportionately higher on mobile* (1,068 exclusive operations vs. 741 on desktop), likely because mobile usage inherently involves richer and more continuous user interaction—such as swipes, taps, and multi-touch gestures—that can reveal behavioral traits. Our crawl data support these assumptions: for example, *TouchEvent.clientX* appeared 203 times in mobile, whereas *ClickEvent.clientX* occured just 56 times on desktop in the same set of scripts. Scripts may preferentially enable such tracking on mobile to leverage these nuanced interaction signals. A representative example of such behavioral tracking appears in Listing 7 in Appendix A, where touch events were triggered on mobile, but the corresponding click events were absent in the desktop execution.

Session State Tracking. We examine the platform-specific attempts by scripts to store or retrieve session-specific information using web storage, IndexedDB, cookies, or database APIs. These methods enable the persistence of identifiers across page reloads or visits, facilitating continuous user recognition. This tracking method *diverged heavily toward desktop*, with 1,796 operations compared to only 466 on mobile. The likely incentive is that desktop browsers offer more reliable, persistent storage mechanisms less constrained by OS-level policies or ephemeral browsing modes, making them a preferred environment for stateful tracking [21].

Network Connection Monitoring. This type of tracking uses a user's network environment to create a profile. It accesses information like connection speed, network type, and background communication from beacon APIs. Network monitoring appeared *more frequently on desktop* (601 exclusive operations vs. 87 on mobile). This may be because desktop devices tend to have stable and more varied network environments, such as wired connections or corporate networks, making them more informative for profiling and traffic shaping. Mobile networks are often transient (e.g., 4G and 5G), reducing their utility for fine-grained network inference.

UI Tracking. Element observers and event listeners monitor changes in the DOM or user interface. The use of mutation observers, intersection observers, and event registration routines indicates close tracking of how users engage with the interface. DOM tracking was *more prevalent on desktop* (571 exclusive operations vs. 260 on mobile), likely due to the complexity and interactivity of desktop interfaces. Desktop websites often feature more dynamic components and responsive layouts suited for instrumentation, while mobile interfaces are simpler and more optimized for touch.

Cyclic DOM Manipulation. We identify scripts that cyclically (more than 5 iterations) add and remove DOM elements—such as creating an element, appending it to the document body, and then removing it—which may indicate attempts to detect whether certain elements (e.g., ads) are being successfully rendered or blocked. This behavior is often used to infer the presence of content blockers or to adjust ad delivery strategies accordingly. We observed 230 divergent executions of this pattern on desktop and 152 on mobile. The *slightly higher usage* in desktop is likely due to the more complicated nature of the web on the desktop that renders more content at default. Mobile's often use lazy loading before completely rendering resources.

Automated Bot Detection. Scripts may determine whether the user is a real human or an automated agent. These include checks for WebDriver flags, precise timing via performance APIs, and animation frame analysis to detect scripted interactions. Divergent execution of bot detection was *far greater on desktop* (1,084 exclusive operations vs. 411 on mobile), likely reflecting the common use of headless or automated browsers in desktop environments for

[4] Although WebGL extension properties can aid bot detection, we categorize them under browser fingerprinting, following prior work [24].

```
O = { blockId: I, limit: E, designId: g, size: y, bannerId v,
backpackData: u,  elHeight: f, diff: h, blockName: d,
    url: C, isTouchScreenDevice: a.isTouchScreenDevice, userAgent: T };
a.isTouchScreenDevice &&
o.addEventListener(o.trackElement, "touchmove", (function() {
    return o.onTouchmove()
}
```

Listing 4: Code snippet from https://yastatic.net/partner-code-bundles/1146736/3fbe7c4448a81e40b6cc.js that runs behavioral tracking on touch devices (mostly mobiles).

scraping and testing, prompting scripts to conditionally execute bot detection only when they detect a desktop context.

The platform-specific execution of tracking behaviors underscores how scripts adapt to the capabilities and limitations of their operating environments. While not inherently malicious, these behaviors disproportionately impact desktop users—subjecting them to more persistent and high-entropy tracking methods such as browser fingerprinting, device profiling, and long-term storage. In contrast, mobile users are more often targeted through interaction-based tracking that leverages rich, gesture-driven input. These findings reveal that tracking is selectively deployed rather than uniformly applied, raising concerns about transparency, user consent, and the uneven distribution of privacy risks across platforms.

Findings 5. Browser fingerprinting, bot detection, and session state tracking are significantly more prevalent in desktop-exclusive executions, accounting for 77% of all such tracking in this category. In contrast, behavioral tracking—such as touch, click, scroll, and tap interactions—is more dominant on mobile platforms, comprising 59% of all occurrences.

8.2 Case Studies

Our hybrid analysis provides deep insights into the branching logic of scripts, revealing how device-specific properties are used to trigger specialized behaviors. In this subsection, we manually examine key data sources, the conditional branches they influence, and the resulting platform-specific API executions to better contextualize the actions scripts take within their exclusive code paths.

To support interpretation, we focus on case studies featuring unobfuscated, human-readable code snippets, allowing us to present them in their original form that preserves their authenticity. Listing 4 provides an excerpt from a script loaded on `interfax.ru`, which leverages custom APIs and the `userAgent` string to infer device characteristics and register event listeners for touch-based interactions. This script was loaded exclusively on mobile devices, explicitly checking for touch support before execution. Within the mobile-specific dynamic trace, we observed message events used to transmit viewport-related data back to the server. Overall, we identified three distinct API subsequences triggered at different stages of the script's lifecycle, all tied to mobile execution and focused on behavioral UI tracking.

Listing 5 presents a bot detection script discovered on `europas-press.com`, which performs canvas fingerprinting to infer low-level graphics capabilities. The browser profiling logic— particularly the sequence of WebGL and canvas-based APIs—was triggered exclusively on the desktop platform. The script detects the missing support for canvas for the headless browser at line 9 and then, as a fallback, starts populating *browser_info* (line 11 and onwards). The

```
async function _() {
    performance.mark("stop"),
    performance.measure("elapsed_time", "start", "stop");
    let e = function() {
        let e = document.createElement("canvas");
        return !!(e.getContext("webgl") ||
        e.getContext("experimental-webgl"))
    }();
    e ? (webgl_vendor = k().vendor, webgl_renderer = k().renderer)
    : (webgl_vendor = "Not Available",
    webgl_renderer = "Not Available"), window.browser_info = {
        Color_Depth: screen.colorDepth, Browser: navigator.appName,
        ↪  Browser_Version: navigator.appVersion, Browser_Engine:
        ↪  navigator.product,
        Webdriver: navigator.webdriver
        // Profiling some more fingerprinting APIs
    }, report_info = {
        browser_info: window.browser_info
    };
}
```

Listing 5: Code snippet from https://madrid.report.botm.transparentedge.io/static/js/bm.js that profiles various device configurations and properties.

```
function d(b) {
    return a.includes("Chrome") && a.includes("Android") }
e.exports = { checkIsAndroidChrome: d }
var l = d.checkIsAndroidChrome
e.exports = new b(function(b, c) {
    b = i(a.navigator.userAgentData, j);
    if (b == null) {
        a.navigator.userAgentData != null && g(new Error("[ClientHint
        ↪  Error] UserAgentData coerce error")); return
    } else if (!l(a.navigator.userAgent)) return;
    b = a.navigator.userAgentData.getHighEntropyValues(["model",
    ↪  "platformVersion", "fullVersionList"]).then(function(a) {
        var b = c.asyncParamFetchers.get(q);
        b != null && b.result == null && (b.result = a,
        ↪  c.asyncParamFetchers.set(q, b));
        return a
    })["catch"](function(a) {
        a.message = "[ClientHint Error] Fetch error" + a.message, g(a) });
    c.asyncParamFetchers.set(q, { request: b, callback: v });
    c.asyncParamPromisesAllSettled = !1 })
```

Listing 6: Code snippet from https://connect.facebook.net/signals/config/823166884443641 that checks if `navigator.userAgentData` reports an Android Chrome environment, then asynchronously fetches high-entropy client hints

mobile browser provides the canvas renderer and vendor information and therefore avoids the fallback.

Listing 6 shows another example of divergent code from Meta's Pixel tracking library [34]. The function at line 1 sets up a detection mechanism for Android Chrome. At line 9, non-Android Chrome devices return a null value, while on Android Chrome, the script reports high-entropy client hints—such as model, platformVersion, and fullVersionList—at line 10. These examples highlight two key takeaways. First, fingerprinting scripts can leverage fine-grained environmental cues to selectively trigger profiling logic based on perceived platform traits. Second, our tool effectively surfaces such divergences, making it valuable not only for privacy analysis but also for bot detection system developers aiming to evaluate platform environments accurately for user experience. While our headless setup may introduce minor variations, our analysis shows that its behavior closely mirrors headful execution (see Section 9 for more details). In contrast, the differences between headless/desktop and mobile environments are far more pronounced.

9 Discussion

Significance of Platform-Specific JavaScript Divergence. Understanding how JavaScript adapts to different execution environments is crucial for addressing security, privacy, and transparency risks on the web. Although platform-specific divergence often serves legitimate purposes—like optimizing rendering, adapting to input methods, or managing resources—it can also lead to asymmetric privacy impacts depending on the user's device. While this work does not definitively demonstrate that such divergences cause asymmetric tracking, it reveals the scale, mechanisms, and consequences of such divergence, showing how seemingly benign adaptations can potentially be exploited for targeted tracking.

Our findings show that platform-specific divergence is both widespread and systematically skewed. Over 20.6% of identically sourced scripts exhibit divergent execution between desktop and mobile, with desktop accounting for 67.9% more of these cases. This indicates that users on different devices are subject to different JavaScript behaviors. Our behavioral analysis of platform-specific execution flows further reveals potential asymmetric tracking behaviors. Desktop users are disproportionately targeted by browser fingerprinting, session replay, and bot detection scripts, with 77% of such operations appearing in desktop-exclusive paths. In contrast, mobile-exclusive flows more frequently employ behavioral profiling—leveraging gesture events and input traces such as scroll, tap, and touch to construct user interaction models.

A key contribution of our work is a hybrid analysis framework that combines static code analysis with dynamic execution traces to detect platform-specific branching in script behavior. Unlike prior approaches that treat fingerprinting as a binary presence of known APIs, our method traces data dependencies to reveal the conditions and information sources driving divergence. We find that 76% of such branches involve known fingerprinting APIs, while 6% rely on lesser-known but platform-revealing APIs—exposing a broader, underexplored attack surface. Our framework also uncovers subtle or emerging behaviors that may bypass traditional detection. By enabling high-fidelity, cross-platform execution analysis and moving beyond static API lists, our approach provides a scalable path forward. It opens new opportunities for transparency and accountability in web privacy and security, shifting the focus from merely detecting trackers to understanding how and when they adapt across users.

Selection of Tools for Hybrid Analysis. We selected VisibleV8 [25] for dynamic analysis over OpenWPM [15] due to its broader browser API coverage, enabling more complete execution graphs while minimizing detection risk. For static analysis, we chose JStap [17], which shares the PDG generation process with DoubleX [18], because its AST-level manipulation allows seamless integration with VisibleV8 logs via offsets. Although CodeQL [20] is a powerful static analyzer, its query language lacks the flexibility needed for our tasks, complicating modifications, and it does not integrate well with VisibleV8 data. Additionally, AST-based analysis with JStap is faster than CodeQL queries for our purposes.

Headless vs. Headful Crawling. To ensure our desktop headless browser emulates organic browsing, we show that results also generalize to headful mode. We re-crawled the top 10K Tranco domains using three modes (mobile, headless, headful) and analyzed the 6,957 domains successfully rendered by all modes. Over a 10-day period, the mobile crawl retrieved the most unique scripts (303K), while headful and headless fetched slightly fewer (281K and 275K, respectively), likely due to mobile-specific resources.

Table 4: Jaccard similarity of script *sets* across platforms.

	mobile	headless	headful
mobile	1.000	0.621	0.602
headless	0.621	1.000	0.871
headful	0.602	0.871	1.000

To understand the similarity of script distribution across platforms, we compare the distinct scripts across platforms by computing the Jaccard similarity scores across different modes, with results highlighted in Table 4. Headless and headful crawls load largely the same script set—(Jaccard $\sim$ 0.87)—indicating that headless Chrome is a good stand-in for real desktop browsing. By contrast, the mobile crawl overlaps with either desktop mode by only about 60-62%, underscoring a clear mobile-versus-desktop split in script usage.

Using the divergence method from Section 3, we analyzed identically-sourced scripts across desktop (headless, headful) and mobile environments based on their dynamic execution. The headless–headful pair exhibits low divergence (mean = 0.097, median = 0.068; std. dev. = 0.09), suggesting consistent control-flow behavior across desktop automation and full-browser modes. In contrast, the headful–mobile pair exhibits markedly higher divergence (mean = 0.46, median = 0.42; standard deviation = 0.31), with the headless–mobile comparison displaying a similarly elevated divergence (mean = 0.49, median = 0.47; standard deviation = 0.29). These results indicate that desktop-based headless automation closely replicates headful execution, while mobile introduces significantly more variation.

Temporal Stability. To evaluate temporal stability, we compared headless–mobile divergence from two crawls (Nov 2024 and July 2025). The first crawl yielded a mean deviation of 0.52 (median = 0.49; std. dev. = 0.29), and the second a mean deviation of 0.49 (median = 0.47; std. dev. = 0.29) A two-sample t-test revealed no significant difference ($p = 0.37$), supporting consistency over time.

Limitations. Our analysis, while comprehensive, has several limitations. First, our information flow tracing (i.e., iFlow) relies on JStap [17], which may miss data flow edges due to its static nature and incomplete modeling of dynamic JavaScript features such as `eval`, higher-order functions, and runtime code generation. As a result, iFlow may not capture all information chains contributing to divergence. Second, VisibleV8 [25] logs all WebIDL-defined APIs but does not log JavaScript built-in functions (e.g., Math.round()), and user-defined functions if these functions are not attached to global objects (e.g., Window). While built-in functions are excluded, they are not central to our divergence analysis, as their behavior is typically consistent across OSes and execution contexts. Moreover, user-defined functions without access to WebIDL APIs generally lack sufficient information to distinguish between platforms. These limitations may result in under-reporting of divergence and incomplete attribution of information sources. Third, when multiple conditions are grouped, our algorithm flags all related nodes, which

can overestimate the responsible data sources. While this introduces noise, all flagged sources may still contribute to divergence. Fourth, we recognize that incomplete support for certain Canvas features may have introduced spurious divergence signals. Nevertheless, the headless–mobile and headful–mobile mean divergence scores remain nearly similar despite this limitation. Finally, 25.4% of iFlows lacked known fingerprinting APIs, but manual inspection of the top 100 APIs in other flows revealed that 6% used lesser-known device indicators. Other unanalyzed APIs may also contain platform-revealing heuristics. Future work can extend our framework to uncover additional latent profiling behaviors.

10 Conclusion

In this paper, we examine platform-dependent divergences in JavaScript execution across mobile and desktop environments. Leveraging large-scale dynamic instrumentation combined with static information flow (iFlow) analysis, we uncover conditional branching in identically sourced scripts and trace the data inputs that drive such behavior. Our analysis reveals that over 20.6% of scripts exhibit platform-specific divergence, with desktop users disproportionately subjected to aggressive tracking—most notably through fingerprinting and bot detection. By pinpointing the data sources behind these conditional branches, we expose both established and emerging fingerprinting vectors, shedding light on privacy asymmetries that often elude conventional detection.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and the shepherd for their guidance throughout the process. This material is based upon work supported in parts by the National Science Foundation (NSF) under grant number CNS-2138138 and CNS-2047260. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] 2025. JSPrism. https://github.com/ahsan238/JSPrism.
[2] 2025. Mitmproxy. https://mitmproxy.org/.
[3] 2025. VisibleV8 for Android. https://github.com/wspr-ncsu/visiblev8/pull/35.
[4] 2025. Webpack. https://webpack.js.org/.
[5] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 674–689.
[6] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1129–1140.
[7] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. *Proceedings on Privacy Enhancing Technologies (PETS)* 2022, 2 (2022), 557–577.
[8] berstend. 2023. *puppeteer-extra-plugin-stealth*. https://www.npmjs.com/package/puppeteer-extra-plugin-stealth
[9] Reuben Binns, Jun Zhao, Max Van Kleek, and Nigel Shadbolt. 2018. Measuring third-party tracker power across web and mobile. *ACM Transactions on Internet Technology (TOIT)* 18, 4 (2018), 1–22.
[10] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-) browser fingerprinting via OS and hardware level features. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society.
[11] 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).
[12] FingerprintJS Contributors. 2025. FingerprintJS. https://github.com/fingerprintjs/fingerprintjs. Accessed: 2025-07-22.
[13] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1515–1532.
[14] Peter Eckersley. 2010. How unique is your web browser?. In *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10*. Springer, 1–18.
[15] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1388–1401.
[16] Christian Eubank, Marcela Melara, Diego Perez-Botero, and Arvind Narayanan. 2013. Shining the floodlights on mobile web tracking-a privacy survey. In *Proceedings of the IEEE Workshop on Web*.
[17] Aurore Fass, Michael Backes, and Ben Stock. 2019. Jstap: a static pre-filter for malicious javascript detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 257–269.
[18] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1804.
[19] David Fifield and Serge Egelman. 2015. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19*. Springer, 107–124.
[20] GitHub, Inc. 2021. CodeQL. https://codeql.github.com/. Accessed: 2025-08-03.
[21] Google Chrome Developers. 2025. *chrome.storage API*. https://developer.chrome.com/docs/extensions/reference/storage/
[22] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1663–1671.
[23] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Anirban Mahanti, and Balachander Krishnamurthy. 2017. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. *Proceedings on Privacy Enhancing Technologies* 2017, 1 (2017), 79–99.
[24] 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)*. IEEE, 1143–1161.
[25] Jordan Jueckstock and Alexandros Kapravelos. 2019. Visiblev8: In-browser monitoring of javascript in the wild. In *Proceedings of the Internet Measurement Conference*. 393–405.
[26] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Ben Livshits, and Alexandros Kapravelos. 2021. Towards Realistic and Reproducible Web Crawl Measurements. In *Proceedings of The Web Conference (WWW)*. 80–91.
[27] 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 *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.
[28] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix Freiling. 2016. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies* (2016).
[29] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*.
[30] Ada Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. 2016. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*.
[31] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.
[32] Jonathan R Mayer. 2009. Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0. *Undergraduate Senior Thesis, Princeton University* 85 (2009).
[33] Jonathan R. Mayer and John C. Mitchell. 2012. Third-Party Web Tracking: Policy and Technology. In *2012 IEEE Symposium on Security and Privacy*. 413–427.
[34] Meta Platforms, Inc. 2025. *About Signals Gateway Pixel | Meta Business Help Center*. Meta Business Help Center. https://www.facebook.com/business/help/514664901027990?id=921478266803729
[35] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching you watch: The tracking ecosystem of over-the-top tv streaming devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 131–147.
[36] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. 2011. Fingerprinting information in JavaScript implementations. *Proceedings of W2SP* 2,

11 (2011).
[37] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* 2012 (2012).
[38] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, Edgar Weippl, and FC Wien. 2013. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, Vol. 5. Citeseer, 4.
[39] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 541–555.
[40] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2016. The leaking battery: A privacy analysis of the HTML5 Battery Status API. In *10th International Workshop on Data Privacy Management, and Security Assurance*. Springer, 254–263.
[41] Nikolaos Pantelaios and Alexandros Kapravelos. 2024. FV8: A Forced Execution JavaScript Engine for Detecting Evasive Techniques. In *Proceedings of the USENIX Security Symposium*.
[42] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 61–70.
[43] Puppeteer Contributors. 2025. *Puppeteer*. https://pptr.dev
[44] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and Defending Against Third-Party Tracking on the Web. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 155–168.
[45] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium (USENIX Security 17)*. 679–694.
[46] Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2018. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1502–1514.
[47] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the ACM Internet Measurement Conference (IMC)*. 648–661.
[48] Michael Schwarz, Florian Lackner, and Daniel Gruss. 2019. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits.. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*.
[49] Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld. 2017. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 329–336.
[50] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*. 97–110.
[51] Thodoris Sotiropoulos and Benjamin Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, Vol. 134. 8:1–8:29.
[52] Oleksii Starov and Nick Nikiforakis. 2017. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 941–956.
[53] StatCounter. 2025. Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share.
[54] Junhua Su and Alexandros Kapravelos. 2023. Automatic Discovery of Emerging Browser Fingerprinting Techniques. In *Proceedings of The Web Conference (WWW)*. 2178–2188.
[55] Naoki Takei, Takamichi Saito, Ko Takasu, and Tomotaka Yamada. 2015. Web browser fingerprinting using only cascading style sheets. In *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. IEEE, 57–63.
[56] tft. 2012. PlayReady DRM. Google Groups, "android-platform". https://groups.google.com/g/android-platform/c/OX2a3Z2jtf8 Accessed: 2025-08-02.
[57] Tranco. 2025. https://tranco-list.eu/list/PN5QJ. Accessed: 2025-07-22.
[58] Zhiju Yang and Chuan Yue. 2020. A Comparative Measurement Study of Web Tracking on Mobile and Desktop Environments. *Proceedings on Privacy Enhancing Technologies (PETS)* 2020 (2020).

A Extended Case Studies

Listing 7 shows code that embeds a wrapper that turns an element into a touch-draggable item by capturing touch events. The particular code snippet shared is setting event listeners that report touch movements and update offsets of the gestures routinely.

Listing 8 shows an example of a script that checks for various media formats that are supported. While the divergence occurs at `Window.navigator` (offset: 132,950), the enumeration of these

```
function Draggable(e) {
return (0, le.A)(Draggable, e), (0, ce.A)(Draggable, [{
    key: "initialize",
    value: function initialize(e) {
        this.ref = e, this.ref.addEventListener("touchstart",
        ↪ this.handleStart, {
            passive: !0 }), this.ref.addEventListener("touchmove",
            ↪ this.handleMove, {
            passive: !0 })
        // more touch event handlers
        } }]
)}
```

Listing 7: Code snippet from res.365scores.com/static/js/1452.6aaeee32.chunk.js that defines eventListeners for touch-based devices.

```
var vt = {
    "com.widevine.alpha": "Widevine", "com.microsoft.playready":
    ↪ "PlayReady", "com.apple.fps": "FairPlay" };
var yt = {
    eme: Object.keys(vt).reduce(function(e, t) {
        e[t] = { name: vt[t], persistentState: false, support: false };
        return e }, {
        support: Boolean(e.navigator.requestMediaKeySystemAccess ||
        ↪ e.MSMediaKeys && e.MSMediaKeys.isTypeSupported ||
        ↪ e.WebKitMediaKeys && e.WebKitMediaKeys.isTypeSupported) })
};
```

Listing 8: Code snippet from players.brightcove.net/5348771529001/938M1Zecs_default/index.min.js that checks for playable media types.

```
hasTouch: function() {
    return !!navigator.maxTouchPoints || !!navigator.msMaxTouchPoints ||
    ↪ (l.matchMedia ? l.matchMedia("(any-pointer: coarse)").matches :
    ↪ "ontouchstart" in l)
}, isMobile: function() {
    var e = navigator.userAgent;
    return /Android|webOS|iPhone|iPad|iPod|Windows Phone|IEMobile|Opera
    ↪ Mini|Mobile|mobile|Tablet [^PC]|CriOS/i.test(e)
}
```

Listing 9: Code snippet from afcs.dellcdn.js that creates helper functions that detect whether the device is a touch-capable device using touch-based events.

```
1 var a = 1;
2 var b = 0;
3 b = b + a + 2;
```

Listing 10: An example JavaScript code snippet that JStap failed to establish data flow edges

media types is part of the information flow and contributes to the information available to the conditional node (offset: 132,928). It should be noted that not all of these formats are supported by Android devices. For instance, Microsoft's PlayReady is not part of the standard Android OS distribution [56].

Listing 9 is a code snippet from a script that uses these touch-event signals along with CSS media queries for any 'coarse' pointer (typical of touch devices) to determine if a device supports touch.

Listing 10 is an example showing missing data flow edges of JStap [17]. In this example, the value of the variable b on the left-hand side of the assignment on line 3 depends on the variables *a*, *b*, and a number 2. The PDG built with JStap shows no data-flow edges between *b* on lines 2 and 3.