# StealthDev: Side-Channel-Resistant Forensic Framework for Investigating Websites with Anti-Debugging

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

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

## Abstract

The analysis of JavaScript within websites is a foundational step in web security analysis. One of the most powerful tools available for such an analysis is the browser's integrated developer tools (`DevTools`), which provide deep visibility into the client-side code execution. However, sophisticated malicious websites often employ anti-debugging techniques to detect the presence of such tools and alter their behavior to appear benign. Prior research into such polymorphic sites has focused mainly on isolated client-side detection vectors. In this work, we argue that collusive vectors, which involve explicit communication between the client and an attacker-controlled server, are common, highly effective, and stealthy. Furthermore, despite the growing prevalence of such tactics, the security community still lacks a comprehensive framework to counter these anti-debugging tactics effectively.

In this paper, we examine the prevalence of isolated and collusive anti-debugging vectors and then develop a solution that enables stealthy analysis of websites. First, we introduce a hitherto unstudied class of anti-debugging techniques in which adversaries exploit debugger-related resource requests, such as those for source maps or debugging symbols, initiated by the browser to detect `DevTools`. Through a large-scale analysis of the top 100,000 websites and their subpages, we find that approximately 1 in 17 websites employing severe anti-debugging measures leverage such collusive vectors to identify and obstruct `DevTools`-based analysis. Second, to defend against these threats, we present *StealthDev*, a policy-driven debugging framework built on a modified Chromium browser. *StealthDev* is specifically designed to remain stealthy and resilient against isolated and collusive anti-debugging techniques, including those relying on timing side channels. Our evaluation shows that *StealthDev* successfully bypasses anti-debugging defenses on over 97% of the most evasive websites, while incurring a modest overhead of just 5% on average page load times.

## CCS Concepts

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

## Keywords

Chromium Browser, Anti-Debugging, Browser Forensics

This work is licensed under a Creative Commons Attribution 4.0 International License.
*ASIA CCS '26, Bangalore, India*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2356-8/26/06
https://doi.org/10.1145/3779208.3805987

**ACM Reference Format:**
Rahul Kanyal and Smruti R. Sarangi. 2026. StealthDev: Side-Channel-Resistant Forensic Framework for Investigating Websites with Anti-Debugging. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 1–5, 2026, Bangalore, India.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779208.3805987

## 1 Introduction

Malicious websites often disguise themselves behind benign front-ends, offering free movies, online games, or fake shopping platforms, while performing harmful actions in the background. These actions may include browser fingerprinting[12], enabling stateless user tracking without consent; advertisement fraud[7, 45], which involves generating illegitimate clicks or interactions with ads; phishing, where users are tricked into disclosing sensitive personal information; and cryptojacking[13], wherein the attacker covertly uses the user's device to mine cryptocurrency. In addition, such websites can participate in distributed denial-of-service (DDoS) campaigns or act as automated clients for brute-force password attacks[4, 32]. Because these behaviors can lead to detection by malware scanners, safe browsing mechanisms, or result in being blacklisted by search engines, ultimately lowering their visibility and reach, attackers go to great lengths to remain undetected.

Developer tools (`DevTools`) [10], integrated in modern browsers, are among the primary tools for the manual inspection of website behavior. They reveal the actual client-side source code executed in the browser and offer capabilities to inspect script execution, track network requests, and understand data flows. However, the effectiveness of these tools is increasingly challenged by the rise of anti-debugging techniques used by websites. Malicious websites, in particular, often detect the presence of `DevTools` and respond by serving sanitized or misleading versions of their code and behavior [26]. A recent study by Musch and Johns [29] demonstrates that **such polymorphic behavior—triggered specifically by the activation of `DevTools` —is alarmingly widespread** and constitutes a major obstacle to manual analysis.

`DevTools`, while powerful, are not designed for stealth. Their presence is easily detectable, triggering evasive actions such as self-destruction, code rewriting, and redirection by websites, hindering effective analysis. This detectability arises from several factors: existing tools cannot conceal their presence, and websites can exploit subtle timing differences or resource utilization patterns to infer the presence of debugging tools. The consequences of these limitations are significant. Detectable tools lead to skewed results, which provide an incomplete or, at worst, misleading view of the website's

behavior. Additionally, detection strategies frequently exploit fundamental JavaScript APIs, making it challenging to develop patches without affecting the functionality of legitimate websites.

The real-world impact of these anti-debugging measures is significant. Some websites employ aggressive countermeasures, such as blocking IP addresses for extended periods after detecting `DevTools` usage, while others continually revert to previous history states to prevent analysis. Research conducted by Li et al. [26] has demonstrated that anti-debugging libraries rank among the most prevalent APIs in cryptocurrency scam websites, second only to common helper libraries such as jQuery and analytics tools. For native (x86) malware, extensive research [8, 23, 25, 31] has been conducted to counteract anti-debugging practices. However, for web-based malware and evasive websites, no solution effectively addresses all the measures deployed by malware authors. This highlights the need for more advanced and stealth-capable `DevTools`.

This paper explores existing anti-debugging measures, analyzes the limitations of existing tools, and develops a solution to enhance web analysis capabilities in response to these evolving challenges.

Our specific contributions are as follows:

❶ Systematic analysis and documentation of a collaborative attack vector previously unstudied, including detection of its prevalence in the wild.

❷ Development of *StealthDev*, the first browser-based debugging environment capable of counteracting prevalent anti-debugging tactics.

❸ Comprehensive evaluation of the effectiveness of *StealthDev* demonstrating the robust stealth capabilities of the framework.

❹ Open-source release of *StealthDev*, with patches for the latest 10 versions of Chromium (versions 118 through 128) available here.

The remainder of this paper is organized as follows: Section 2 provides an overview of debugging and anti-debugging techniques in the context of JavaScript. We then introduce a collaborative anti-debugging attack vector in Section 3, followed by our threat model in Section 4. Section 5 introduces the design and architecture of *StealthDev*, while Section 6 presents our extensive evaluation results. Section 7 explores the advanced techniques discovered during our Web-scale measurement study. Section 8 presents an overview of the related work, and finally we conclude in Section 9.

## 2 Background

### 2.1 Debugging JavaScript

While JavaScript can be analyzed statically and dynamically, static analysis approaches face significant limitations when dealing with the complexities of real-world web applications. Features like `eval`, `setTimeout`, `setInterval`, `Function` and the inherently dynamic nature of JavaScript code generation pose substantial challenges for static analysis, often leading to inaccurate representations of runtime behavior. Comprehensive surveys on JavaScript analysis challenges [2, 38] consistently emphasize these limitations of static analysis approaches. Furthermore, the complicated interplay between JavaScript, the Document Object Model, and browser events, which are essential aspects of web-based JavaScript, requires a complete browser environment for accurate analysis. The level of emulation needed for the correct analysis of web-based JavaScript proves extremely difficult to achieve with purely static analysis tools. This is where browser-integrated `DevTools` become essential. These tools provide a rich suite of features for inspecting, debugging, and modifying web page states, enabling deeper insights into potential attacks. The Chromium browser, which serves as the open-source foundation for Google Chrome, offers a comprehensive toolkit with the following features: The **record and replay** functionality [19] enables analysts to reproduce attack scenarios and accurately investigate root causes. Additionally, the **network panel** [28] provides a log of network resources loaded by websites. Finally, the **console panel** [3], which supports debugging through breakpoints, variable inspection, and also offers the ability to execute custom JavaScript within the site's context [30]. By providing these dynamic analysis capabilities within the browser, `DevTools` enables developers and analysts to understand, debug, and secure complex JavaScript applications in ways that static analysis alone cannot achieve.

### 2.2 Anti-Debugging

Malicious actors employ various anti-debugging techniques to evade detection by `DevTools`. Recent research has shown that some websites go to great lengths to detect and disable the browser's `DevTools` [29]. These anti-debugging techniques can include detecting the presence of `DevTools`, disable the `DevTools` panel, or modify the browser's JavaScript APIs to deceive `DevTools` [29]. Multiple sites like [21, 33] provide commercial anti-`DevTools` and anti-debugging features. Musch and Johns [29] systematized nine techniques used in the wild for anti-debugging and analyzed the top 1 million sites to find that up to 1 out of 550 websites contain severe anti-debugging measures. Anti-debugging techniques can be categorized into three main types: **Impede**, **Detect**, and **Alter**.

**Impede** techniques are designed to obstruct the analysis process, making it more difficult for the analyst to inspect a website. For example, the **CONCLEAR** method involves continuously clearing the console using the `console.clear` API, which prevents the analyst from seeing any logs. The **SHORTCUT** method disables all the keyboard shortcuts that open `DevTools`, such as `F12`, effectively blocking quick access to debugging tools. Another technique, **TRIGBREAK**, repeatedly triggers debugger statements, causing frequent interruptions in the code flow, which can slow the analyst.

**Detect** techniques aim to identify whether `DevTools` or a debugger is being used, providing early warning signs to the adversary. The **MONBREAK** method detects `DevTools` by comparing timestamps before and after a `debugger` statement, as the execution only pauses if a debugger is present. The **CONSPAM** technique involves running `console` APIs repeatedly and noting timing differences, which can indicate the presence of `DevTools`. Similarly, the **NEWBREAK** method repeatedly executes an arbitrary function and measures the timings; if an analyst sets a breakpoint, the timings will diverge from the norm, signaling a debugger's presence. The **WIDTHDIFF** method checks for changes in window size to detect if `DevTools` are open, as resizing can be a signal. Lastly, the **LOGGET** technique inserts a marker in the `toString` function of an object, helping to identify if the object is logged in the console, thus revealing the presence of a debugger.

**Alter** techniques modify the output of logging APIs to mislead the analyst. For example, the **MODBUILT** method involves

**Table 1: Examples of anti-debugging techniques [29]**

| Category | ADT | Signature | Description |
|---|---|---|---|
| IMPEDE | CONCLEAR | `for(;;) console.clear();` | Constantly invoke console APIs to clear logged data |
| | TRIGBREAK | `for(;;) debugger;` | Place anonymous debugger statements to pause `DevTools` |
| | SHORTCUT | `if(event.key==="F12"){`<br>`event.preventDefault();`<br>`return false;}` | Disable shortcut keys like F12, Ctrl + Shift + I/J that open the `DevTools` pane |
| ALTER | MODBUILT | `console.log=function(arg){`<br>`if(arg==="shellcode"){`<br>`arg="benign code";}`<br>`console.log(arg)}` | Modify builtin APIs to log altered results or hide true values |
| DETECT | WIDTHDIFF | `let oW = outerWidth;`<br>`let iW = innerWith;`<br>`if((oW - iW)>threshold)`<br>`{/* DevTools open! */}` | Detect `DevTools` using the difference in dimensions of page |
| | CONSPAM | `let start=performance.now();`<br>`for(let i=0; i < 100; i++){`<br>`console.log(i)}`<br>`let current=performance.now();`<br>`if((current - start)>threshold)`<br>`{/* DevTools open! */}` | Detect `DevTools` using the difference in time to execute console APIs with and without the presence of console |
| | MONBREAK | `let start=performance.now();`<br>`debugger;`<br>`let current=performance.now()`<br>`if((current-start)>100)`<br>`{/* DevTools open! */}` | Measures debugger execution time (negligible if `DevTools` are closed) to detect if they're open. |
| | LOGGET | `var canary = function(){};`<br>`canary.toString = function()`<br>`{/* DevTools open! */}`<br>`console.log('%c', canary);` | Logs a canary function with custom getter function that marks if the `DevTools` are open. |
| | NEWBREAK | `let current=performance.now();`<br>`let diff=current-timeSinceLast;`<br>`if(diff > threshold)`<br>`{/* DevTools open! */}`<br>`timeSinceLast=performance.now();` | Compares the time since the last check to a threshold to detect `DevTools`, then updates the timer. |

monkey-patching built-in functions such as `console.log` to alter their output, providing false or misleading information. This can make it significantly more complicated for the analyst to understand the actual behavior of the code, as the logs they rely on may not accurately reflect the program's state. Table 1 summarizes all techniques with examples.

### 2.3 V8 JavaScript Engine

The Chromium browser with over 43 million lines of code [18] has a multi-process, multi-threaded architecture. Each browser tab is managed by a separate process that uses multiple threads. Every process has well-defined roles and security permissions. This design enhances stability and security by isolating the execution environments of different tabs. The content within each tab, including HTML, CSS, JavaScript (JS) and WebAssembly (WASM), is rendered by the Blink rendering engine. Blink, in turn, uses the V8 engine to execute JavaScript and WebAssembly code. The V8 JavaScript engine has multiple execution tiers to optimize performance. The non-optimizing compiler that linearly walks the bytecode and swiftly emits the corresponding machine code; the `Turbofan` optimizing compiler, which generates highly optimized machine code from frequently executed bytecode; and the intermediate `Maglev` JIT (Just-In-Time) compiler, which provides an additional layer of optimization by compiling bytecode to machine code more quickly than `Turbofan`, albeit with fewer optimizations. Since JavaScript is interpreted and the data type of each variable used for optimizations is at best speculative, whenever a wrong optimization is performed by Turbofan, multiple de-optimizations are required to correct the generated bytecode.

## 3 SOURCETRACK

While the vectors detailed in Section 2 operate entirely within the client-side environment of the web browser, this section introduces a previously unexamined `DevTools` detection vector that uses a hidden communication channel between the browser and the server. This specific mechanism, which facilitates detection through server-side side-channels, was first identified by Gal Weizman [16, 44]. Despite its potential for high-fidelity detection, it has remained largely overlooked in the academic literature regarding its prevalence, variations and countermeasures.

JavaScript source code executed in the browser often differs from the original source code due to various transformations like

minification or transpilation (language-to-language translation) from other languages like TypeScript or CoffeeScript. To help the developer debug the transformed code, Chromium provides a feature called source maps that allows the developer to specify the mapping between the modified and original code (using the `// # sourceMappingURL`) directive at the end of the web page to specify the URL of the corresponding source map. Since this is a developer-only feature, source maps are loaded only when the `DevTools` pane is open. However, the client cannot observe this behavior since the JavaScript API or event listener currently does not notify about the loading of the source maps. Hence, the client needs a bounce-back from the server to confirm the source map request. This feature can be exploited to detect if the `DevTools` are open by making a web request for source maps; if the request succeeds, then the website can safely assume that the `DevTools` are open.

Listings 1 and 2 present an example method to detect `DevTools` using WebSockets and source mapping URLs. The process begins on the client side, where a WebSocket connection is established with a server (`ws://server_ip:port` in the listing) and the client actively listens for server messages. On the server side, incoming file requests are analyzed to determine the appropriate content type. If a request is made for a source map file (such as `/script.js.map`), the server logs the request and sends a WebSocket message to the client, indicating the detection of a request for a source map. On receiving the message, the client parses the received JSON data and dynamically updates the webpage to reflect whether `DevTools` has been detected. This hidden communication enables the client to modify its JavaScript source code and camouflage its true behavior. Moreover, the network panel in `DevTools` does not log network requests and responses for source maps; therefore, detecting sourcemap-based detection is even more challenging. We call this technique **SOURCETRACK**.

```
1  // Establishes a connection with the server
2  let ws=new WebSocket('ws://server_ip:port');
3  // Sets a listener for the incoming messages
4  ws.onmessage = (event) => {
5  // Parse the received data
6  let data = JSON.parse(event.data);
7  let messageElement = document.getElementById('message');
8  // If sourceMapRequested is true, then Devtools were
       detected
9  messageElement.innerText = data.sourceMapRequested ? '
       Devtools Detected' : 'No Devtools Detected'; };
10 //# sourceMappingURL=script.js.map
```

**Listing 1: Source map based `DevTools` detection (client side)**

```
1  // extname is the extension of the requested file
2  switch (extname) {
3  // source maps have a .map extension
4  case '.map':
5    contentType = 'application/json';
6  // the request was for a source map
7    if (req.url === '/script.js.map') {
8    console.log('Request for source map file.');
9    // if the socket connection exists
10     if (t_ws) {
11     // notify the client that the sourcemap was requested
12 t_ws.send(JSON.stringify({ sourceMapRequested: true }));
13 } }
14   break;}
```

**Listing 2: Source map based `DevTools` detection (server side)**

Source maps are enabled by default in the Chromium browser and represent the most accurate (zero false positives) and covert method for detecting the presence of `DevTools`. In our experiment to detect the prevalence of source maps, we use a real Chromium browser controlled via `Node.js` through the `DevTools` protocol to crawl the top 100,000 URLs and their three random sub-pages from the CrUX dataset[14]. A naive method of fetching the URLs of the source map could be to do a regular expression match to find `// # sourceMappingURL =` . However, as we observe in Section 7, scripts with source mapping URLs can be generated on the fly using JavaScript. These cases would be missed by a simple regular-expression-based matching. Furthermore, it might be challenging to detect source maps using regex if they are specified using the `X-SourceMap` header in the HTTP response. By registering the `Debugger.scriptParsed` event within the protocol, we can analyze the source maps embedded in websites. To detect the liveness and availability of source maps, we follow a methodology similar to Rack and Staicu [34]. Analyzing the top 100k websites, we find that approximately 51% include at least one source map. A more detailed breakdown of the presence of the source maps is provided in Figure 5 in the Appendix. This experiment demonstrates that source maps are highly prevalent on popular web pages. We further investigate the **SOURCETRACK** attack vector in Section 6.3, where we identify it on 115 out of 2,000 websites that we studied in depth.

## 4 Threat Model and Objectives

We assume a threat model similar to Musch and Johns [29], where we consider the following scenario: An analyst aims to analyze a website suspected of containing malicious JavaScript code. The analyst intends to use `DevTools` to understand the true behavior of the website as shown on the end-user's device. On the other hand, website developers, aware of potential analysis attempts, have implemented various anti-debugging measures to protect code.

**Attacker Capabilities:** We assume an attacker with extensive knowledge of web anti-debugging and evasion tactics who can deploy such mechanisms on websites to deter browser-based forensic analysis. Specifically, the attacker has the following capabilities: the attacker owns or has compromised one or more domains, such as `malicious-site.com`, which might be hiding malicious code. Moreover, the website's code may be obfuscated, encrypted, or may actively employ anti-debugging techniques to hinder analysis; the attacker has typical access to all the JavaScript browser APIs (around 7000) as would be available to any other normal website, including methods like timing and performance APIs, access to the console APIs, local storage, cookies, and self-inspection with the reflect APIs; and the attacker can remotely alter the behavior of the web application based on the detection of debugging environments.

### 4.1 Objectives

**In-Scope Objectives** Our framework, *StealthDev*, is designed to counter the attacker's anti-debugging tactics and allow reliable forensic investigation in hostile web environments with the following objectives: (1) stealthy inspection of anti-debugging techniques, enabling analysts to observe and examine site behavior as it is presented to a typical user without triggering site responses aimed

at detecting debugging environments; (2) accurate timing adjustments, whereby the framework detects and adjusts timing-based anti-debugging mechanisms that attempt to identify delays caused by the debugging environment due to `DevTools` executing additional code and incurring extra clock cycles, and then modifies the site's behavior accordingly; and (3) policy-based control over API calls, providing fine-grained control over browser API calls using a policy enforcement layer that allows certain debugging activities to be masked, altered, or bypassed entirely, as specified by the analyst.

**Out-of-Scope Objectives** *StealthDev* makes certain assumptions about the limitations of the analysis environment. First, it does not address server-side evasion techniques that rely on inherently non-deterministic remote timing measurements through network requests, as these can introduce a high rate of false positives in `DevTools` detection; therefore, network-based timing measurements are outside its scope. Second, while *StealthDev* is designed to counter various in-browser debugging detection methods, it does not handle sandbox evasion strategies, which require additional, system-level defenses beyond its focus on JavaScript-based anti-debugging tactics. Finally, *StealthDev* does not cover client-device fingerprinting techniques that depend on identifying analyst devices via IP addresses or geographical locations, since such techniques can be mitigated more straightforwardly using VPNs.

# 5 StealthDev

## 5.1 Overview

*StealthDev* modifies the V8 Bytecode Generator to introduce analyst-controlled capabilities that enable selective actions (e.g., bypassing, virtualization, or monitoring) on a per-API-invocation basis for calls originating from the webpage or the `DevTools`. By strategically applying these actions to API calls associated with timing side-channels and debugger-detection logic used by anti-debugging websites, *StealthDev* masks the presence of `DevTools`. To achieve this, *StealthDev* includes three core components: the ❶ **Policy Enforcement Layer**, which processes the policies defined by the analyst and picks the appropriate action for each API call (generated by the Bytecode Generator in the V8 engine); the ❷ **API Virtualization Layer**, which manages multiple instances of browser APIs and dynamically redirects the V8 engine to either the original or custom API implementation based on the analyst's directives; and the ❸ **Timing Control System**, which intercepts timing-related APIs specified in the policy file and modifies the reported timing data to obscure the presence of `DevTools`. Figure 1 illustrates the interaction of these three core components.

The majority of *StealthDev*'s modification involves injecting hooks in the generated bytecode to invoke custom `Runtime functions`, implemented as highly optimized C++ routines within V8; they are designed to be called directly during bytecode execution. These functions specifically track the origin of the bytecode, monitor the execution time of various APIs, and virtualize them as specified in the policy file; this enables policy-based decisions regarding the debugger's behavior. Importantly, *StealthDev*'s modifications to the `Ignition` interpreter are implemented in a way that remains fully transparent to the higher-tier optimizing compilers, namely `Sparkplug`, `Turbofan`, and `Maglev`. As a result, these changes do not interfere with the functioning of the compilers, preserving the integrity of the overall execution pipeline while seamlessly applying StealthDev's instrumentations at the interpreter level. The patches to build *StealthDev* are available here.

## 5.2 Policy Enforcement Layer

The policy enforcement layer in *StealthDev* enables analysts to precisely control the browser's behavior for each API call made by a website. For each target domain, the corresponding API actions are defined in a `policy.json` file. Each policy rule is expressed in JSON (JavaScript Object Notation) and follows the structure: `{"api": "api_name", "scope": "scope_identifier", "policy": "policy_label"}`. Each rule in the policy specification maps a specific API and its associated origin to an enforced action. This mapping governs how the browser should handle API invocations within the defined scope. We formally define these components and discuss the available action options in Appendix D.

Each API call is systematically governed by a specific action based on both its origin and execution scope. We now provide an overview of each action. The action ACCESS grants access to the API tuple by invocation. The BYPASS skips the invocation of the API call and returns `undefined` where the API was invoked. The action MONITOR monitors the execution of specified APIs and tracks the total execution time within each scope. The monitored time is then used to adjust the reported timings using timing-related APIs such as `performance` and `date`. The action ALTER applies only to timing and performance APIs such as `Date` and `Performance`. The APIs labeled with ALTER modify the reported time values within the specified scope. The action PROXY creates an isolated and unique copy of the API for the getters and setters invoked in the scope defined by the analyst. The API that is proxied in the policy specification is mapped to a different unique API for the scope defined by the analyst. For getters and setters, the proxying function is used to find the corresponding mapping for the requested API. If no mapping is found, the original browser's built-in API is invoked; otherwise, the proxied API is invoked.

An example policy to disable `console.clear` (used in the **CONCLEAR** attack) for all websites except the `DevTools` pane and a custom trusted website (http://trusted-site.com) is given in Listing 3.

```
1 [   { "api": "console.clear", "scope": "SITES",
      "policy": "BYPASS" },
2     { "api": "console.clear", "scope": "DEVTOOLS",
        "policy": "ACCESS" },
3     { "api": "console.clear", "scope":
        "http://trusted-site.com", "policy": "ACCESS" }]
```

**Listing 3: An example policy for remediating the CONCLEAR attack**

Since multiple actions may apply to a given API - provenance pair, we require a systematic way to combine them into a single consolidated action. To address this, we define an algebraic structure that forms an idempotent and commutative monoid. A monoid is an algebraic structure that comprises a set equipped with an associative binary operation and an identity element. The associativity ensures compositionality, allowing us to compute sub-actions independently and then combine them in parallel. The idempotence and commutativity of the operation ensure that repeated or reordered actions yield consistent results, which further enables efficient caching and deduplication of intermediate computations.

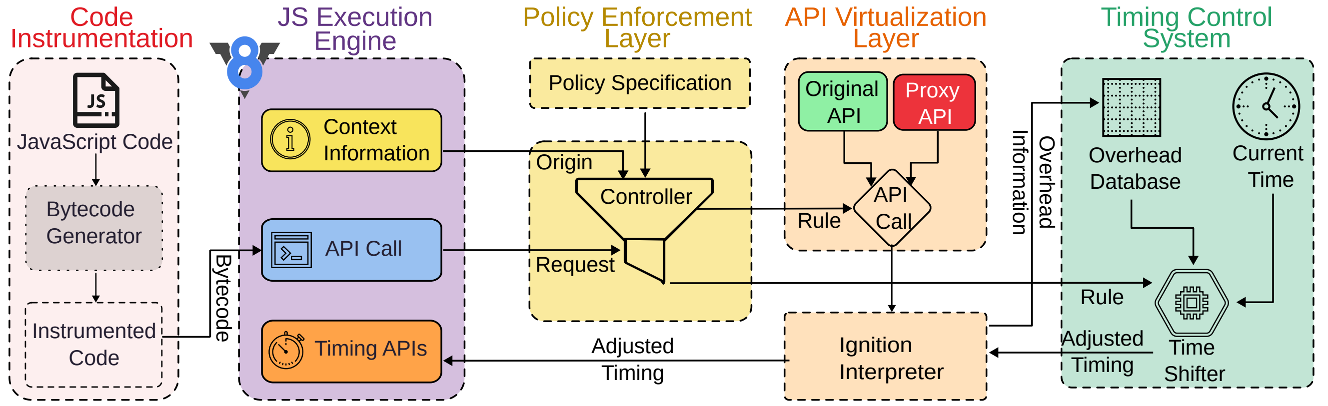


**Figure 1: Overview of the internals of StealthDev**

Together, these algebraic properties enable a range of optimizations for computing actions associated with APIs. The structure of the resulting monoid is described in detail in Appendix D.

The MONITOR action when combined with the PROXY action, tracks the execution time only in the original APIinvoked from the `DevTools`. Additionally, the ALTER action modifies the reported timings exclusively in the proxied API (the site's copy) when used in conjunction with the PROXY action. A special case arises with proxy actions, where multiple copies of the same API may be created. In such scenarios, the final consolidated action always corresponds to the copy within the same execution scope as the original API.

**Implementation** The policy database is initialized during the isolate creation phase and reloaded from the policy file whenever a new browser tab is opened. To enforce policies on named (e.g. `a.b()`) or keyed (e.g. `a[b]()`) API calls, simply checking the receiver (a) and property (b) at runtime is insufficient. This is because objects and their properties can be aliased using arbitrary variable names, such as `c = a.b; c()` or `c = a; c.b()`. Therefore, to correctly identify the origin of an API call, it is necessary to traverse the property access chain and resolve the name using the constructor. To achieve this, we track API objects when they are first loaded into memory during property access. This is done by instrumenting the `VisitPropertyLoad` function of the bytecode generator. We store the association between the object and its accessed property in a hash table, which maps object pointers to the corresponding object-property names. However, because the garbage collector may move objects around in memory, we must ensure that these mappings remain valid. For this, we use an `EphemeronHashTable`, which allows mapping a heap-allocated object pointer to another object. In our case, the hash table is used to map the heap-allocated object (for the API) to an ASCII string that represents the API's name. This data structure ensures that the mapping remains valid despite object relocation and that the associated ASCII string is deallocated once the referenced API object is garbage collected.

To identify the provenance of API calls to decide the action to apply, we modify the `Bytecode Generator`, which emits bytecodes after traversing the Abstract Syntax Tree (AST) in the V8 engine. Of the 60 bytecode builders and 85 AST visitors, `StealthDev` requires modifications to only one bytecode builder, namely `BuildSetNamedProperty`, and one visitor, `VisitPropertyLoad`. We allocate three additional registers: one for the name (constructor) of the object, one for the object itself, and one as an accumulator (contains the actual return value for the API call). These registers are then passed to a runtime API call that processes them further. For cleaning up these temporaries after the runtime call ends, we make use of the `HandleScope` in the V8 engine, which ensures automatic deallocation of memory when the scope of a variable ends.

The `origin` property of the global object is used to distinguish between calls that originate from `Chrome` and `DevTools`. These origins represent the setup code executed by the browser to initialize the `DevTools` window, open a new tab, or other internal Chrome pages such as `settings` or `profiles`. Furthermore, the `scriptName` field for the code loaded from `DevTools` is empty, whereas for the code from the site, it contains the actual name of the `.js` file downloaded over the network. Once the origin of the console API call has been determined, along with the name of the API (from the constructor), *StealthDev* computes the appropriate action based on the origin and name of the API. If the BYPASS action applies to the API call, then the runtime function returns a boolean `true` value in the accumulator. Additionally, the `BytecodeGenerator` injects conditional code that redirects execution to a placeholder function, which produces an `undefined` value instead of invoking the genuine API call based on the return value in the accumulator. Section 5.3 describes the implementation of the action PROXY. Section 5.4 explains the details of the action labels ALTER and MONITOR.

**Optimizations** Trivially computing the action based on the policy specification, the origin of the browser API, and the name of the API incurs substantial overhead (5.4× on the speedometer benchmark[6]). Furthermore, when multiple rules apply to the same browser API, computing the final action using the action-combination operator (as detailed in the Appendix D) adds additional overhead. Additionally, since scripts can embed sub-scripts, resolving the origin across an entire chain of script embeddings for every API call becomes computationally expensive. To improve efficiency, the origin tracking system maintains a execution context that triggers an update only upon script transitions. As a result, the origin needs to be computed once per context, rather than every instruction. To further reduce the overhead of repeated action

computation by the action-combination operator, we introduce a dynamic multilevel cache indexed by the script identifier (assigned by V8 for each unique script), followed by the API name. This approach allows us to reuse previously computed actions across invocations within the same script's context.

Additionally, we need to keep track of active scripts and skip cache lookups to stale entries for inactive scripts. We manage script liveness explicitly: once a script finishes execution, we mark all its cache entries as inactive using hooks in the `Bytecode Generator`. If the script is later reloaded, we simply reactivate its cache entries, allowing the reuse of previously computed actions and scopes. These optimizations collectively reduce the performance overhead from 5.4× to 2.9× on the speedometer benchmark.

## 5.3 API Virtualization/Proxy Layer

This section describes the implementation of the PROXY action label that is used to bypass the anti-debugging vector **MODBUILT**. JavaScript allows developers to override built-in functions and APIs which presents both significant benefits and drawbacks. On the one hand, it enhances security by allowing developers to intercept and modify attempts to log information into the console, thus protecting sensitive data. However, this can obscure important information necessary for legitimate debugging and security checks. Hence, to ensure the integrity of the browser APIs for use by the analyst, *StealthDev* creates multiple copies of the APIs for which the action label is computed as PROXY. *StealthDev* hooks the `BuildSetNamedProperty` function within the `BytecodeGenerator` class of the V8 engine. The `BuildSetNamed Property` function generates bytecode for function or object assignments to named properties. This allows us to pass the object and the name of the property being reassigned to a runtime API call for further processing.

*StealthDev* introduces an additional conditional statement in the `BytecodeGenerator` to decide whether to emit the original code or a modified version. This decision is based on a runtime API call that computes the action for each API invocation. When the runtime API call returns `true`, indicating that the API call should be proxied, the original assignment is renamed with a unique identifier by appending a random marker. For example, `console.log = function()` is transformed into `console.log_stealth = function()`. For subsequent API calls within the same scope, the unique marker is appended to the API call request, ensuring that the custom proxied API is invoked instead of the original. However, for API calls made in different scopes, the original API is invoked, thereby preserving the site's intended functionality. Additionally, in cases where no prior assignment has been made to the API, the original API is directly invoked without any proxying or modification. This selective proxying mechanism allows *StealthDev* to maintain fine-grained control over API behavior within specific scopes, while ensuring that the overall functionality of the website remains unaffected in unmodified scopes. Appendix B shows a sample instrumented bytecode generated by *StealthDev* for API renaming.

## 5.4 Timing Control System

This section explains the implementation details of the ALTER and MONITOR action labels used to counter the **CONSPAM**, **MONBREAK**, and **NEWBREAK** anti-debugging vectors. These vectors exploit inherent timing differences that arise due to the additional clock cycles required to format and print data to the console, which is bypassed when the console is not visible or when the debugger is paused for manual analysis.

To implement the MONITOR action, *StealthDev* utilizes the `TimeTicks` class in the V8 engine. The `TimeTicks` class provides an abstract and monotonically incrementing time representation, internally represented in microseconds. Additionally, `TimeTicks` offers a memory-relaxed operation, meaning that it does not impose synchronization or ordering constraints on other reads or writes: only the atomicity of the timing instruction is guaranteed. This property allows *StealthDev* to capture exact timing information without requiring explicit synchronization.

To accurately measure the time elapsed during a monitored API call, *StealthDev* inserts hooks into the `VisitCall` function of the `BytecodeGenerator`, executing runtime API calls immediately before and after each browser API invocation. If the MONITOR action is active, the current timestamp from `TimeTicks` is recorded before the call. After invocation of the API, another timestamp is captured, and the difference is stored in a timing overhead database, maintained per isolated context, including separate tracking for web workers. This overhead is later used to adjust the timing data reported by the `Performance` and `Date` APIs. For the ALTER action, which intercepts and modifies calls to the `Performance` and `Date` APIs, *StealthDev* instruments the `VisitCall` function in the `BytecodeGenerator`. This function generates bytecode for property accesses. During execution, it passes the original timing, API name, and object to a runtime call that applies the ALTER action. To conceal `DevTools`, *StealthDev* recomputes and returns modulated timings using the overhead database from the MONITOR action.

API calls originating from unmarked scopes, such as the `DevTools` pane, remain unaffected. Additionally, *StealthDev* accounts for the timing overhead introduced during paused debugger states to counter **NEWBREAK** and **MONBREAK** anti-debugging techniques that measure time under pause to detect `DevTools`. This is handled within the `V8Debugger` class's `BreakProgramRequested` function, where a timer is started when program execution is paused. Upon resumption, the total paused time is updated using the timer, ensuring that the measured time under pause is accurately integrated into the timing adjustments.

Alternatively, adversaries may use counter-based timing via loop iterations. However, these counters scale with page load fluctuations and system variability, remaining consistent with expected baselines. Even if real elapsed time leaks, hardware diversity, system load, and browser extensions introduce substantial noise, making detection via raw load differences impractical. Our experiments show that browser extensions can increase load times by up to 30%, creating enough variability that minor overhead leaks fail to provide a reliable detection signal.

## 5.5 Miscellenaous Patches

To handle detection techniques based on variations in the height or width of the browser window, *StealthDev* always opens the `DevTools` pane undocked in a separate window. This ensures that the dimensions of the original browser window remain unchanged. Since shortcuts to `DevTools` can be easily re-assigned, as illustrated

in the listing 1, it is necessary to ensure that the tool remains functional even in the presence of such reassignments. To address this, *StealthDev* automatically spawns an undocked `DevTools` window for every new website, without relying on user-initiated shortcuts. This guarantees that `DevTools` are always available, regardless of shortcut remapping. This mechanism effectively mitigates the **SHORTCUT** and **WIDTHDIFF** anti-debugging vectors. Furthermore, to counter **TRIGBREAK** in *StealthDev*, the `Output Debugger()` function in the `BytecodeArrayBuilder` class of the V8 engine is modified to prevent the output of the `debugger` statement whenever the debugger is disabled in the policy specification file. Since the `debugger` statement is never executed in the JavaScript code of a website during normal execution, its deactivation does not affect site functionality. Attempting to remove the `debugger` statement at the network layer using a browser extension, which replaces it with a null statement, may unintentionally trigger anti-tampering techniques deployed by the website that check the integrity of the JavaScript code. Such an approach is non-stealthy. Instead, *StealthDev* modifies the Abstract Syntax Tree (AST) during the bytecode generation phase, ensuring that the source code remains untouched and that no anti-tampering mechanisms are triggered.

# 6 Evaluation

**Overview** We begin by crawling the top 100,000 URLs and three random sub-pages for each URL from the CrUX dataset [14] using a custom crawler controlled through the `DevTools` protocol. This process generates a dataset of sites that exhibit Basic Anti-Debugging Tactics (BADTs). Subsequently, we apply a severity filter to identify the top 2,000 URLs with the highest levels of anti-debugging activity. These most severe BADTs serve as a basis for detecting the prevalence of Sophisticated Anti-Debugging Tactics (SADTs). For the **SOURCETRACK** vector, we use the top 2,000 URLs that have at least one source map. Furthermore, to assess the presence of SADTs and validate the stealth capabilities of *StealthDev*, we record these severe sites and perform a deterministic coverage-guided replay.

**Ethics** In conducting our web crawling activities, we implemented several precautions to minimize the risk of overburdening target websites. Specifically, we ensured that no simultaneous requests were made to the same site. In addition, we introduced deliberate delays between consecutive requests to further reduce the potential server load. For cases where multiple visits to a single website were required, we employed a record-and-replay framework to limit the number of live interactions with the site while still allowing for comprehensive analysis.

## 6.1 Prevalence of Basic Anti-Debugging Tactics

Basic Anti-Debugging Tactics (BADTs) refer to the following **CONCLEAR**, **TRIGBREAK**, **SHORTCUT**, **MODBUILT**, **WIDTHDIFF** techniques. In this section, we describe our evaluation setup to measure the prevalence of BADTs along with the observations.

**Setup** We visited the 100K most popular websites according to the Chrome User Experience Report (CrUX) [14]. We chose CrUX rankings because of the findings of [35] which state that CrUX captures the set of most popular sites with the greatest precision. For each visit, we use the same user agent (`Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36 Edg/126.0.0.0`) and desktop resolution (1366x768), allow all third-party cookies, do not set the "Do Not Track" HTTP header or other privacy-preserving techniques (e.g., anti-tracking extensions). We started 30 parallel instances of Chromium 125.0.0 to detect anti-debugging websites. We follow a methodology similar to Musch and Johns [29] where on each page, our crawler waits up to 30 seconds for the load event to trigger. Otherwise, we flag the site as failed and move to the following site. After the load event, we wait up to 3 more seconds for pending network requests to resolve to better handle pages that dynamically load additional content. Finally, we stay for 5 seconds on each loaded page for the spamming techniques to trigger. **We repeat this process for 3 random sub-pages.** Table 2 summarizes the status codes and exceptions that we observed while navigating to the websites from the CrUX dataset.

**Table 2: Status Codes for the top 100,000 CruX URLs**

| Status Code | Number of Sites | Percentage |
|---|---|---|
| 200-299 (Success) | 88529 | 88.52% |
| 300-399 (Redirection) | 6 | 0.006% |
| 400-499 (Client Error) | 5744 | 5.74% |
| 500-599 (Server Error) | 477 | 0.477% |
| Exception | 5244 | 5.24% |
| Total sites : | 100000 | |

**Table 3: Sites with BADTs**

| Technique | 1-2 | 3-4 | 5-10 | 10-20 | >20 | TOTAL |
|---|---|---|---|---|---|---|
| SHORTCUT | 1647 | 755 | 106 | 2508 | 0 | 2508 |
| TRIGBREAK | 322 | 22 | 30 | 45 | 220 | 639 |
| CONCLEAR | 107 | 30 | 72 | 284 | 74 | 567 |
| MODBUILT | 4297 | 565 | 3145 | 522 | 417 | 8946 |
| WIDTHDIFF | 0 | 520 | 835 | 1126 | 6386 | 8867 |

**Results** We identified anti-debugging markers in 19,373 websites. Consistent with Musch and Johns [29], **MODBUILT** and **WIDTHDIFF** were the most prevalent techniques. Sub-page navigation increased detection by 11% on average. Among sites using **MODBUILT**, `console.log` and `console.debug` were the most frequently modified APIs. **SHORTCUT** is the third most common technique found in the wild and is a very strong indicator of anti-debugging. **TRIGBREAK** and **CONCLEAR** are the strongest anti-debugging indicators found on 1,100 websites.`Console.clear` might have a legitimate use to hide the errors and warnings thrown at runtime due to various policy violations or network errors, from the end-user, but continuously invoking `Console.clear` might indicate nefarious actions. In terms of combinations of anti-debugging techniques, we observe that most websites deploy a single anti-debugging technique, with two techniques being the second most common combination. This is different from the trend observed in the 2019 study by Musch and Johns[29], where sites with two anti-debugging measures were more common. The full distribution is shown in Table 3.

## 6.2 Prevalence of Sophisticated Anti-Debugging Tactics (SADTs)

Sophisticated Anti-Debugging Tactics (SADTs) refer to the **CONSPAM**, **MONBREAK**, and **NEWBREAK** techniques.

**Setup** We record websites using the WPR tool[11] and then replay to measure the code divergence. The tool is designed to record the loading of a website and create an archive file with all requests and responses, including the headers. This archive file can then be used to create a deterministic replay of the previously recorded page. We use the severity metrics defined by Musch and Johns [29] to quantify the strength of the anti-debugging mechanisms deployed on a website. These metrics rank how frequently an anti-debugging vector is observed in a website compared to the websites where the same vector was found at least once. More details about the severity metrics can be found in the Appendix C. Table 4 reports the distribution of severity scores for websites that exhibit at least one anti-debugging marker.

**Table 4: Distribution of severity score**

| Severity Score | # Websites | % Total |
|---|---|---|
| 0-1 | 18502 | 95.50 |
| 1-2 | 468 | 2.42 |
| 2-5 | 355 | 1.83 |
| 5-10 | 48 | 0.25 |

We then select the top 2,000 sites with the highest severity scores and perform an in-depth evaluation of side-channel-based detection `DevTools`, following the Musch and Johns technique [29]. The sites with the highest severity scores belong primarily to the "suspicious" category, followed by sites in the "shopping" and "business" categories (as classified by Symantec site review[39]).

**Results** A total of 661 sites had at least one of the sophisticated anti-debugging techniques. We observe that **NEWBREAK** is the most common side-channel-based detection technique present in more than 67% of sites with active anti-debugging tactics (661). This is closely followed by the **CONSPAM** and **MONBREAK** techniques present in around 54% AND 56% sites, respectively. Furthermore, we observe that the highest number of sophisticated anti-debugging techniques is from the shopping category, followed by the suspicious and gambling categories. In addition, we observe that **CONSPAM** and **MONBREAK** frequently occur together, as is evident from the most popular combination of SADTs containing two sophisticated anti-debugging techniques in Table 5. Only 114 sites out of 2000 most severe BADTs employed all three techniques.

**Table 5: Combination of BADTs and SADTs**

| #Techniques | BADTs | | SADTs | |
|---|---|---|---|---|
| | # Sites | %Total | # Sites | %Total |
| 1 | 17371 | 89.67 | 257 | 38.88 |
| 2 | 1862 | 9.61 | 290 | 43.87 |
| 3 | 128 | 0.66 | 114 | 17.25 |
| 4 | 12 | 0.06 | 0 | 0 |

## 6.3 Prevalance of SOURCETRACK

**Setup** Since the record-and-replay method is ineffective for analyzing client-server covert communication detection techniques, such as **SOURCETRACK**, we adopt an alternative approach. First, we navigate to a website and allow it to load completely before monitoring the code for convergence. Initially, we drop network requests to the source maps while storing them in a list for later use. To assess convergence, we capture code coverage every second and compare it with the previous coverage data. If the coverage remains consistent for 10 consecutive iterations, we proceed to the next phase, where we sequentially make network requests to the previously collected source maps. If we observe any divergence in the code on the page following these network requests, or if the page navigates or redirects to a different URL, we mark the page as containing **SOURCETRACK**. We then repeat the same experiment three times for confirmation. The complete algorithm is specified in the appendix E. **Results** We observe that 115 websites change the source code or navigate to an entirely different address in response to the network request for sourcemaps. Figure 2 plots the categories of websites that deploy **SOURCETRACK** in which business category has the highest number of candidates.

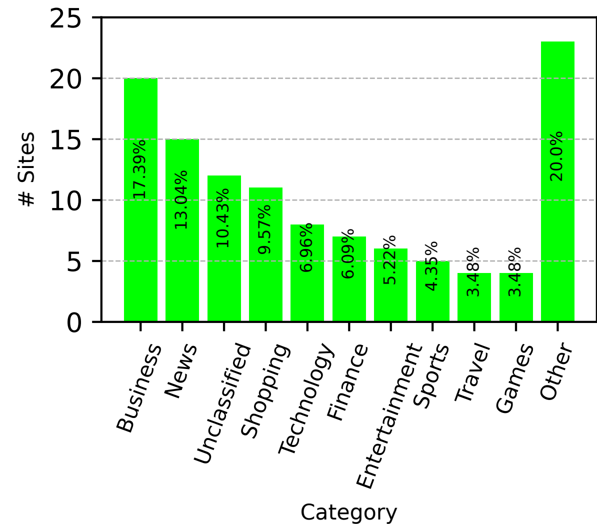


**Figure 2: Categories of sites that deploy SOURCETRACK**

## 6.4 StealthDev Evaluation

*6.4.1 Stealth.* We use a technique similar to that for detecting sophisticated anti-debugging tactics to validate the stealth and non-interference capabilities of *StealthDev*. Under *StealthDev*, if the stealth criteria are met, the code coverage on the websites should remain consistent in at least 3 out of 5 test runs. Specifically, statements executed under *StealthDev* should form a strict subset of the union of statements collected during the initial 50 replays. Additionally, *StealthDev* should execute all statements that appear in every replay. Our analysis shows that, after simulating `DevTools` with external timing constraints, code differences were observed in

only 35 out of the 2,000 severe BADTs. This discrepancy may be due to an unknown anti-debugging vector or false positives arising from the record-replay infrastructure. The results for stealth verification are tabulated in Figure 3.

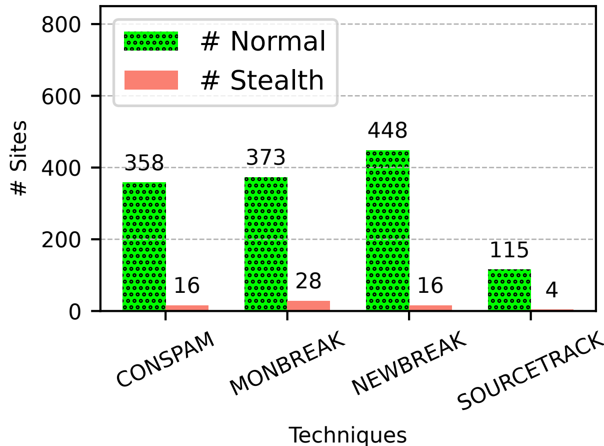


**Figure 3: Comparison of `DevTools` detection techniques found in normal and Stealth modes**

6.4.2 *Correctness.* For side channel timing-based detection techniques like **CONSPAM**, we simulate a typical scenario in which the timings of 10,000 console API invocations are monitored for 50 runs with and without `DevTools`. Figure 4 shows the timing difference (in milliseconds) with and without `DevTools` for an unmodified browser and *StealthDev* normalized with respect to the base values observed without `DevTools`. We observe that the mean difference in an unmodified browser is 301, whereas, for *StealthDev*, it is 0.06, demonstrating the ability of *StealthDev* to remain tolerant to time-based detection techniques. Since *StealthDev* tracks the time across the paused state, similar results are observed for the **MONBREAK** and **NEWBREAK** techniques.

6.4.3 *Performance.* To evaluate *StealthDev*'s performance, we navigated to the homepage of CruX 's top 1,000 websites with Chromium version 127 and *StealthDev* (built on top of Chromium version 127); we noted the time from `navigation start` to `load event` for every website three times and took the average. *StealthDev* has an average **overhead of less than 5% in the load times** for the top 1,000 websites measured using the Chrome remote interface over nodejs. To evaluate performance, we use the speedometer benchmark that major browser engine developers have developed to test performance on real-world workloads such as news sites, rich text editors, and to-do lists. We observe a 2.9× **slowdown**, which is comparable to other V8 engines' bytecode instrumentation frameworks such as VisibleV8 [22] (2.5×). Since this is just an initial prototype and a proof of concept, we did not optimize for performance. The biggest impact on the performance of StealthDev is the computation of the origin for each console and performance API call. Performance profiling of *StealthDev* using the Speedometer benchmark reveals that the primary source of computational overhead stems from

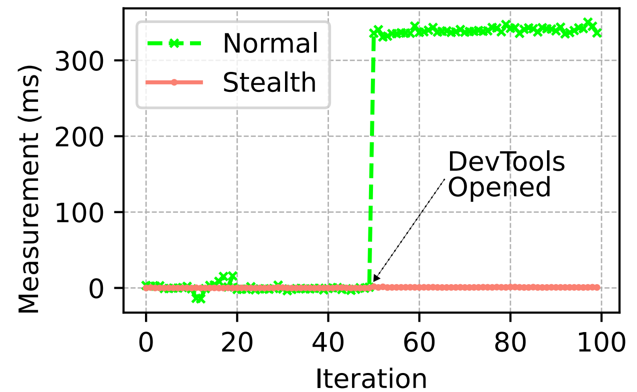


**Figure 4: Comparison of CONSPAM timings in normal and Stealth mode, `DevTools` spawned at 50th iteration**

origin tracking. Specifically, the `StealthOriginManager` runtime function, which is responsible for determining the provenance of each API call, accounts for 65.28% of the total additional CPU ticks. This is followed by the policy evaluation and action enforcement logic (`StealthPropertyGet` in the provided source code), which contributes 25.37% of the overhead. The remaining overhead is distributed across the custom API call infrastructure: the prologue (`StealthCallStart`) incurs 7.30%, the epilogue (`StealthPostGet`) accounts for 1.08%, and the core API call handler contributes a marginal 0.95%. . This impact can be minimized by having better context hints initiated by the developer tools pane during invocation and closing events. For computing spatial overhead we navigated the CruX top 1,000 websites 5 times and took a heap snapshot, using a normal browser and *StealthDev*. We observed that on average an overhead of less than 150KB (0.8 %) is incurred for each website (with an average heap size of 18MB), where the peak overhead was 4MB.

6.4.4 *Maintainability.* Since we limited the primary modifications to the bytecode generator of the V8 engine only, we were able to easily port the patches developed for Chromium version 118 to Chromium version 128. We make patches available for multiple versions of Chromium (with minimal design changes between versions) to demonstrate easy maintenance of *StealthDev*. The changes would be required only when the API responsible for invoking run-time functions changes, which is very infrequent.

6.4.5 *Comparison with State-of-the-Art.* The only known solution for the anti-debugging problem is the injection of custom anti-anti-debugging scripts through either the use of an extension "Anti-Anti-Debug" (AAD) [36] or programmatic injection [15] using the Greasemonkey [27] or the Tampermonkey [5] extension. Both approaches possess the same underlying capabilities and limitations. These scripts disable the console API calls using the JavaScript proxies and remove the `debugger` statement from each function. We compare *StealthDev* with AAD in terms of the ability to counteract known anti-debugging vectors (Table 6). The evasive website can easily detect and counteract AAD, as it modifies newly created functions to remove the debugger statement. Furthermore, since

the JavaScript-based solutions can not take the origin of the API call into account while disabling the API call, they inadvertently disable the analyst's ability to log sensitive data as well. In addition, they cannot handle timing-based attacks that monitor the use of breakpoints by the analyst.

**Table 6: Comparison of browser-extension-based solution and StealthDev**

| Technique | Extension | StealthDev |
|---|---|---|
| CONCLEAR | ✓ | ✓ |
| TRIGBREAK | ✗ | ✓ |
| SHORTCUT | ✗ | ✓ |
| MODBUILT | ✗ | ✓ |
| WIDTHDIFF | ✗ | ✓ |
| CONSPAM | ✗ | ✓ |
| ADVANCED CONSPAM | ✗ | ✓ |
| MONBREAK | ✓ | ✓ |
| NEWBREAK | ✗ | ✓ |
| SOURCETRACK | ✗ | ✓ |
| ADVANCED SOURCETRACK | ✗ | ✓ |
| TAMPER DETECTION | ✗ | ✓ |

# 7 Discussion

## 7.1 Reasons for Anti-Debugging

Eleven percent of the top 100 most severe sites that deploy sophisticated anti-debugging mechanisms were classified as malicious by at least one threat detection vendor in VirusTotal[42]. 27% sites included scripts known to be anti-devtool either as an `npm` package[1] or a publicly available library[36, 40] on github.

In manual analysis of these 100 websites from the list of SADTs, we observe that most sites with severe anti-debugging tactics cannot be analyzed with the `DevTools` present in Chromium, Edge, or Firefox. Moreover, anti-anti-debugging extensions address only basic techniques and are ineffective against sophisticated tactics. We used *StealthDev* for the analysis, as it is immune to anti-devtool measures. We could not find any malicious scripts / behaviors on 63 websites. This might indicate that the anti-devtools measures are primarily used to protect the proprietary JavaScript code of the website. 27 of these 63 websites were gambling/lottery sites. One of the reasons for this could be to protect against the development of bot players. We observed 11 websites in the pool of the 100 most severe websites where multiple delays and captchas were implemented before finally redirecting to the main website. Since delays are on the client side, these can easily be bypassed using `DevTools`. Hence, these 11 sites use anti-`DevTools` measures to **counter static analysis** approaches that merely analyze the source code. Three of these 11 websites are associated with an automated interaction farm that provides fake engagement in the form of likes, comments, and followers for various social platforms such as Instagram, TikTok, YouTube, and Facebook. 5 of the 11 anti-static-analysis websites offered money for remote access to WhatsApp (through a browser-based client). Since remote access is not easily detectable, this access can be used for proxy marketing and bypassing spam filters. Four websites from the pool of 100 websites implemented a roulette wheel to grant a joining bonus for betting games, the wheel always stopped at *try again* for the first attempt, and gave the bonus 70% on the second attempt. This was confirmed with the analysis of the source code of the website using *StealthDev*. These four websites deployed anti-devtools measures to **cloak this rigged lottery setup**. Six pirated video sharing sites created transparent overlays that inadvertently registered as ad clicks and then redirected to the advertised sites. Since transparent overlays can be easily removed using Developer Tools, these sites implement anti-DevTools measures to prevent **the removal of ad overlays**. One site embedded a third-party script associated with **cryptomining**. The reasons for the fifteen sites could not be concluded. We deobfuscated multiple anti-debugging scripts that were present on various sites and made one of them available for further analysis at the following link.

## 7.2 Advanced Techniques

Furthermore, we observed that a website created an empty script with a sourcemap URL that gets injected into the main page and then gets immediately removed, while observing the cookie with the name "DevTools". The response to the URL of the sourcemap contains a `set-cookie` header that sets the "DevTools" cookie. This is invoked continuously to detect `DevTools`, meanwhile, it has no side effect on the console panel (as with other techniques). Listing 4 presents the code for **ADVANCED SOURCETRACK** deployed on a real-world website that makes use of **SOURCETRACK** for detecting `DevTools`. Snippet 7 in the appendix F further shows the code for **ADVANCED CONSPAM** on a website that allocates `DevTools` detection to a web worker thread, because of which extension-based solutions will not be able to intercept `console` calls. This would be handled easily by *StealthDev* since it can track marked APIs in all contexts.

One of the websites blocks the IP of the analyst for 24 hours if `DevTools` are detected. Also, we observed a script on a website that detects if the source code is pretty printed (used to convert minified JavaScript code to a human-readable form). On detecting pretty-printing, the script stops executing. This anti-debugging measure is deployed to stop local testing of anti-debugging scripts.

```
1  function smap(e) {
2  var t = document.createElement("script");
3  t.textContent = "//# sourceMappingURL=" + e + "?v=" +
       Date.now(),
4  document.head.appendChild(t),
5  t.remove()}
6  Cookies.get("DevTools") && Cookies.remove("DevTools"),
7  smap("/devtools");
8  var devtoolsDetectInterval = setInterval(function() {
9  "1" === Cookies.get("DevTools") && ($(".watching_player-
       area").remove(),
10 window.location.href = "/",
11 clearInterval(devtoolsDetectInterval))}, 100)
```

**Listing 4: Cookie-and-source-mapping URL based devtools detection**

## 7.3 Limitations

*StealthDev*, similar to Shi and Mirkovic [37], cannot reliably detect timing manipulations when time is obtained through network APIs.

Because our `DevTools`-based detection relies on continuously observing local system time, network-based measurements exhibit high variance due to non-deterministic delays, which can result in false positives. Attempting to actively handle such cases would likely interfere with website functionality and introduce instability.

Another limitation of our methodology is that we do not interact with the visited websites in any way. This prevents us from capturing anti-debugging techniques that are triggered only upon user interaction, such as mouse movements, clicks, or form submissions. Consequently, some evasive behaviors may go undetected, leading to false negatives.

Finally, we rely on web categorization services such as Symantec SiteReview[39] to classify websites. These services can occasionally misclassify sites, and while the impact of individual errors is often small, such misclassifications can propagate and bias results in large-scale studies. We therefore acknowledge this as a potential source of noise in our dataset.

# 8 Related Work

Our work largely relates to the field of detecting and bypassing the analysis of malware in an analysis environment (virtual machine, debugger, emulator) and the field of hiding the detection of the analysis environment from malware. We start this section with the works in the field of Anti-Debugging (with a focus on web-based anti-debugging); we then follow up with an overview of the works that deal with the detection and resolution of anti-debugging.

## 8.1 Anti-Debugging

Prior works have primarily focussed on anti-debugging in the case of native malware. Works by Chen et al. [31] and Paleari et al. [8] use various hardware-level registers and counters to detect the analysis environment. Since the language semantics of JavaScript do not allow low-level access to these registers, these techniques cannot be applied directly to websites. Ho et al. [17] demonstrated the first browser-only red pills (code used to identify if the browser is running in a virtual or emulated environment) based on the timing differences in usual browser operations like spawning web workers, writing to the console, and so on. They used JavaScript timing APIs (accurate to milliseconds) to calculate timing differences. *StealthDev* can be used inside the emulator or virtual machine to counter the red-pills and help remediate the challenges associated with virtualizing a browser environment. Johh et al. [20] studied the sites that analyzed the user-agent string in the HTTP request to present a masqueraded version of the site to a search engine crawler to avoid being flagged and obtain a higher search ranking to distribute malware. Since *StealthDev* is a real browser, these masked techniques against a crawler fail. Musch and Johns [29] collected and analyzed the techniques used by websites to obstruct, alter, or detect the analysis using browser developer tools. Our work builds upon the techniques studied in this paper and remediates all the anti-debugging techniques mentioned in their research.

## 8.2 Anti Anti-Debugging

Wang et al. [43] studied the prevalence of script self-deletion in the top 1 million sites using a custom browser JSRAY that can monitor the dynamic operations performed by the script, like deleting the host attribute or the host element itself. Their solution can only detect if the website deletes the script and is not concerned with hiding the analysis environment (devtools) from the site. Cobra[41] executes native code in units of basic blocks with additionally implanted stubs and hooks that transfer control to Cobra for detailed analyses. Furthermore, Cobra maintains a copy of system memory that approximates the state of memory without the debugger. This copy is used for fetching values requested by the malware. They implemented various stealth implants to hide the environment. Their approach can be implemented in the bytecode builder of Chromium's V8 engine but would require significant engineering effort for implementation and maintenance. Instead of implementing a custom analysis environment using the Cobra API, *StealthDev* reuses the existing devtools for debugging, albeit with additional stealth capabilities. BareCloud[24] compared the execution behaviour profile of native malware in multiple analysis environments (virtualization, emulation and hypervisor) against a bare-metal environment to capture deviation in the execution to detect evasive samples. Rozzle [25] solves the problem of fragility (environment dependency) by executing all the paths of a control flow statement with weak updates (append to common variables instead of overwrite); this allows the framework to trigger code that might have remained unexecuted otherwise. A website can easily detect Rozzle by having code in the form of canaries in impossible locations that should never be executed.

HybridEMU [9] is a CPU simulator supporting Windows API calls. During the simulated execution of instructions, any API call is passed to a real system for evaluation, and the returned value is forwarded to the malware under analysis. Any API call that reveals the debugger is handled by modifying the returned value. *StealthDev* modifies the actual bytecode generated and does not need a custom simulator for analysis.

# 9 Conclusion

In this paper, we presented a systematic investigation into isolated and collaborative anti-debugging techniques employed by evasive websites and introduced **StealthDev**, the first in-browser debugging framework engineered to overcome all known in-scope anti-debugging tactics. We demonstrated that it is possible to develop a robust anti-anti-debugging framework for browser-based forensic analysis by introducing a policy-based enforcement layer within a modified Chromium environment. By selectively bypassing, modifying, or monitoring API calls through carefully crafted policies, *StealthDev* provides a stealthy and resilient debugging environment capable of countering advanced anti-debugging tactics. Furthermore, we provide open-source releases of *StealthDev*, with patches compatible across 10 recent Chromium versions (118 through 128), to support ongoing analysis and development in anti-evasion research.

# Acknowledgements

The authors thank the anonymous reviewers for their constructive feedback. This work was supported by the Department of Science and Technology (DST), Government of India, under SERB project grant CRG/2023/008919.

# References


[1] aepkill. 2024. devtools-detector. https://www.npmjs.com/package/devtools-detector [Online; accessed 25. Oct. 2024].

[2] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–36.

[3] Kayce Basques. 2019. Console overview. https://developer.chrome.com/docs/devtools/console [Online; accessed 26. Aug. 2024].

[4] Alex Belkin, Nethanel Gelernter, and Israel Cidon. 2019. The Risks of WebGL: Analysis, Evaluation and Detection. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II* (Luxembourg, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 545–564. doi:10.1007/978-3-030-29962-0_26

[5] Jan Biniok. 2026. Home | Tampermonkey. https://www.tampermonkey.net [Online; accessed 20. Feb. 2026].

[6] BrowserBench. 2025. BrowserBench.org — Browser Benchmarks. https://browserbench.org [Online; accessed 3. Jul. 2025].

[7] Yegui Cai, George O.M. Yee, Yuan Xiang Gu, and Chung Horng Lung. 2020. Threats to Online Advertising and Countermeasures. *Digital Threats: Research and Practice* 1 (5 2020), 1–27. Issue 2. doi:10.1145/3374136

[8] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. Institute of Electrical and Electronics Engineers Inc., USA, 177–186. doi:10.1109/DSN.2008.4630086

[9] Seokwoo Choi, Taejoo Chang, Sung-woo Yoon, and Yongsu Park. 2021. Hybrid emulation for bypassing anti-reversing techniques and analyzing malware. *The Journal of Supercomputing* 77, 1 (2021), 471–497.

[10] Chrome. 2024. Chrome DevTools. https://developer.chrome.com/docs/devtools [Online; accessed 1. Aug. 2024].

[11] Chromium. 2024. Catapult - Web Page Replay. https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/README.md [Online; accessed 25. Oct. 2024].

[12] 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, Association for Computing Machinery, Berlin Germany, 1–18.

[13] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. 2018. A First Look at Browser-Based Cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. Institute of Electrical and Electronics Engineers Inc., London, UK, 58–66. doi:10.1109/EuroSPW.2018.00014

[14] Chrome for Developers. 2024. Overview of CrUX. https://developer.chrome.com/docs/crux [Online; accessed 14. Oct. 2024].

[15] Cristion Kjeldgard (Frisk). 2026. Bypass DevTools Detection. https://greasyfork.org/en/scripts/534968-bypass-devtools-detection [Online; accessed 20. Feb. 2026].

[16] Gal Weizman. 2026. Javascript Anti Debugging - Abusing SourceMappingURL. https://weizmangal.com/2019/12/18/js-anti-debug-1 [Online; accessed 20. Feb. 2026].

[17] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. 2014. Tick tock: building browser red pills from timing side channels. In *Proceedings of the 8th USENIX Conference on Offensive Technologies* (San Diego, CA) *(WOOT'14)*. USENIX Association, USA, 2.

[18] Open Hub. 2024. The Chromium (Google Chrome) Open Source Project on Open Hub: Languages Page. https://openhub.net/p/chrome/analyses/latest/languages_summary [Online; accessed 25. Oct. 2024].

[19] Sofia Emelianova Jecelyn Yeen. 2021. Record, replay, and measure user flows. https://developer.chrome.com/docs/devtools/recorder [Online; accessed 26. Aug. 2024].

[20] John P. John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. 2011. deSEO: Combating Search-Result Poisoning. https://www.usenix.org/conference/usenix-security-11/deseo-combating-search-result-poisoning

[21] Jscrambler. 2024. Pioneering Client-Side Protection Platform | Jscrambler. https://jscrambler.com [Online; accessed 23. Jul. 2024].

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

[23] Minho Kim, Haehyun Cho, and Jeong Hyun Yi. 2022. Large-scale analysis on anti-analysis techniques in real-world malware. *IEEE access* 10 (2022), 75802–75815.

[24] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 287–301. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat

[25] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *2012 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers Inc., San Francisco, CA, USA, 443–457. doi:10.1109/SP.2012.48

[26] Xigao Li, Anurag Yepuri, and Nick Nikiforakis. 2023. Double and nothing: Understanding and detecting cryptocurrency giveaway scams.

[27] Anthony Lieuallen. 2005. Greasemonkey – Get this Extension for xn–9s9h Firefox (en-US). https://addons.mozilla.org/en-US/firefox/addon/greasemonkey [Online; accessed 20. Feb. 2026].

[28] Dale St. Marthe. 2024. Network panel: Analyze network load and resources. https://developer.chrome.com/docs/devtools/network/overview [Online; accessed 26. Aug. 2024].

[29] Marius Musch and Martin Johns. 2021. U Can't Debug This: Detecting {JavaScript} {Anti-Debugging} Techniques in the Wild. 2935–2950 pages. https://www.usenix.org/conference/usenixsecurity21/presentation/musch [Online; accessed 14. Mar. 2023].

[30] Den Odell. 2014. Browser Developer Tools. In *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Apress, Berkeley, CA, Berkeley, CA, USA, 423–437. doi:10.1007/978-1-4302-6269-5_16

[31] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: how to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (Montreal, Canada) *(WOOT'09)*. USENIX Association, USA, 2.

[32] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. 2023. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation - NDSS Symposium. https://www.ndss-symposium.org/ndss-paper/master-of-web-puppets-abusing-web-browsers-for-persistent-and-stealthy-computation [Online; accessed 3. Apr. 2026].

[33] PreEmptive. 2024. Online Javascript Obfuscator - JSDefender Demo – PreEmptive. https://www.preemptive.com/online-javascript-obfuscator [Online; accessed 23. Jul. 2024].

[34] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, Copenhagen Denmark, 3198–3212.

[35] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. 2022. Toppling top lists: evaluating the accuracy of popular website lists. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 374–387. doi:10.1145/3517745.3561444

[36] Andrew S. 2024. Anti-Anti-Debug. https://github.com/Andrews54757/Anti-Anti-Debug [Online; accessed 25. Oct. 2024].

[37] Hao Shi and Jelena Mirkovic. 2017. Hiding debuggers from malware with apate. In *Proceedings of the Symposium on Applied Computing*. Association for Computing Machinery, Marrakech Morocco, 1703–1710.

[38] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–34.

[39] Symantec. 2024. Symantec Sitereview. https://sitereview.bluecoat.com/# [Online; accessed 25. Oct. 2024].

[40] thejack. 2024. disable-devtool. https://github.com/theajack/disable-devtool [Online; accessed 25. Oct. 2024].

[41] A. Vasudevan and R. Yerraballi. 2006. Cobra: fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, USA, 15 pp.–279. doi:10.1109/SP.2006.9

[42] VirusTotal. 2024. VirusTotal - Home. https://www.virustotal.com/gui/home/upload [Online; accessed 19. Oct. 2024].

[43] Xinzhe Wang, Zeyang Zhuang, Wei Meng, and James Cheng. 2024. Detecting and Understanding Self-Deleting JavaScript Code. In *Proceedings of the ACM on Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1768–1778. doi:10.1145/3589334.3645540

[44] X-C3ll. 2018. JavaScript AntiDebugging Tricks. https://x-c3ll.github.io/posts/javascript-antidebugging

[45] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. 2014. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 373–380. doi:10.1145/2663716.2663719

## A Distribution of source maps

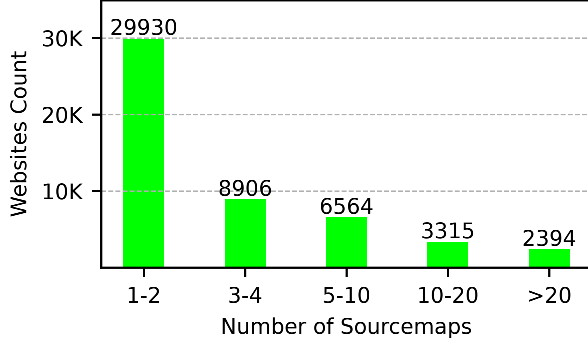


**Figure 5: Distribution of the number of source maps included across the top 100k websites**

## B Instrumented Bytecode

Listing 5 shows the original bytecode emitted by V8, while Listing 6 shows the bytecode generated by *StealthDev*. Line 9 calls the runtime API responsible for computing the action, and based on the return value, the execution jumps to line 13, which invokes a renamed API, whereas the original API is invoked if the runtime API returns `false`.

```
1 LdaConstant [0]
2 Star0
3 LdaGlobal [1], [0]
4 Star2
5 GetNamedProperty r2, [2], [2]
6 Star1
7 CallProperty1 r1, r2, r0, [4]
8 LdaUndefined
9 Return
```

**Listing 5: Default bytecode generated for API calls**

```
1 LdaConstant [0]
2 Star0
3 LdaGlobal [1], [0]
4 Star2
5 LdaConstant [2]
6 Star5
7 Mov r2, r3
8 Mov r3, r4
9 CallRuntime [HandlePropertyGet], r3-r5
10 JumpIfToBooleanTrue [8] (*)
11 GetNamedProperty r2, [3], [2]
12 Jump [6] (+)
13 * GetNamedProperty r2, [2], [4]
14 + Star1
15 CallProperty1 r1, r2, r0, [6]
16 Star4
17 LdaConstant [2]
18 Star5
19 Mov r0, r3
20 CallRuntime [TraceAPICall], r2-r5
21 LdaUndefined
22 Return
```

**Listing 6: Instrumented bytecode generated for API calls**

## C Severity Metrics

**Severity Metrics** These metrics allow us to assess the severity of the techniques deployed. The confidence score metric captures how frequently a particular anti-debugging technique is observed on a given site compared to other sites. The "confidence score" is based on the percentile rank of the site within the distribution of sites that employ the same technique. The percentile ($p_i$) for a technique $i$ at a site is given by the following:

$$p_i = \frac{\text{Number of sites with lesser rank in technique } i}{\text{Total number of sites with technique } i} \tag{1}$$

The confidence score is defined as the square of the percentile:

$$\text{Confidence Score}_i = p_i^2 \tag{2}$$

We specify the weight of each anti-debugging technique using the Inverse Document Frequency (IDF) metric, which highlights the rarity of a given technique across all sites. For a technique $i$, the weight is calculated as:

$$\text{Weight}_i = \ln\left(\frac{N}{n_i}\right) \tag{3}$$

where:

- $N$ is the total number of sites in the dataset,
- $n_i$ is the number of sites using technique $i$.

**Severity Score:** The severity score combines the confidence score and weight to quantify the overall seriousness of a technique on a site. It is computed as follows:

$$\text{Severity Score} = \sum_i \left(\text{Confidence Score}_i \times \text{Weight}_i\right) \tag{4}$$

where the sum is taken over all anti-debugging techniques $i$ observed on the site.

## D Policy Enforcement Layer

Let $\chi$ denote the name of a browser API (e.g., `console.log`), which follows the `object.property` notation. The API name $\chi$ can refer either to the object itself or to a specific property (e.g., `console` or `console.log`). Let $P_\chi$ represent the origin of the API invocation. The API origin $P_\chi$ can take one of the following forms:

- `devtools:\\` — the Developer Tools pane,
- `chrome:\\` — an internal Chrome page,
- A full URL (e.g., `https://a.com/test.js`) — general case for script origins.

We represent an API invocation as a tuple $T_\chi = \langle \chi, P_\chi \rangle$: the API and its provenance. Let the scope $\rho$ define the context in which a rule applies. It can take values from the set:

- **SITES** ($\rho_{SI}$) — all standard websites,
- **DEVTOOLS** ($\rho_{DT}$) — Developer Tools,
- **CHROME** ($\rho_{CR}$) — internal Chrome browser pages,
- **EXTENSION** ($\rho_{EX}$) — Chrome extensions,
- **CUSTOM_URL** ($\rho_{CU}$) — a specified URL, e.g., `https://example.com`.

Let $R_n$ denote the $n^{\text{th}}$ rule in the policy specification, and let $\omega$ represent the action assigned to the tuple $T_\chi$ within the scope $\rho$. We can formally express $R_n$ as:

$$R_n\langle T_\chi, \rho \rangle : T_\chi \xrightarrow{\rho} \omega \tag{5}$$

This notation specifies that the API invocation, identified by its provenance and evaluated within the context defined by $\rho$, is mapped to a corresponding action $\omega$ by the policy rule.

In this way, each API call is systematically governed by a specific action based on both its origin and execution scope. We now provide a detailed explanation of the allowable actions ($\omega$) for each rule.

- **ACCESS**($\omega_{AC}$):
$$\frac{\omega = \omega_{AC}}{T_\chi \longrightarrow \textbf{EXECUTE}(\chi)} \tag{6}$$
where **EXECUTE**($\chi$) executes the API $\chi$ in the current scope without any modifications.
- **BYPASS**($\omega_{BP}$):
$$\frac{\omega = \omega_{BP}}{T_\chi \longrightarrow \langle \textbf{return} \text{ undefined} \rangle} \tag{7}$$
- **MONITOR**($\omega_{MO}$): Let $C_\rho(\chi)$ denote the time taken for API $\chi$ to execute within the contextual scope $\rho$, and let $\Delta$ represent the cumulative time spent executing all monitored APIs. Formally:
$$\frac{\omega = \omega_{MO}}{T_\chi \longrightarrow \langle \textbf{EXECUTE}(\chi) \rangle, \Delta \longrightarrow \Delta + C_\rho(\chi)} \tag{8}$$
- **ALTER**($\omega_{AL}$):
$$\frac{\omega = \omega_{AL}}{T_\chi \longrightarrow \langle \textbf{EXECUTE}(\chi), \textbf{return}(d_v - \Delta) \rangle} \tag{9}$$
where $d_v$ denotes the default value that would have been returned if the rule was not in place and $\Delta$ is the cumulative time spent in API execution (as defined in the **MONITOR** action).
- **PROXY**($\omega_{PX}$): As the API is proxied in the policy specification, $\chi$ is mapped to a different unique API $\chi'$ for the scope defined by $\rho$ via the proxying function $M_\rho$:
$$M_\rho\langle\chi\rangle : \chi \longrightarrow \chi' \tag{10}$$
**Getters:** Let $\chi$ be the requested API and $\alpha$ be the target receiver (variable that holds the value of the requested API).
$$\frac{\omega = \omega_{PX}}{M_\rho\langle\chi\rangle = \chi'} \tag{11}$$
If $M_\rho(\chi) \neq$ `null`, then:
$$\frac{\text{assign proxied API } \chi'}{\alpha \Leftarrow \chi'} \tag{12}$$
Otherwise, if $M_\rho(\chi) =$ `null`:
$$\frac{\text{assign original built-in API } \chi}{\alpha \Leftarrow \chi} \tag{13}$$
**Setters:** Let $\chi$ be an internal API to which a custom function ($\beta$) is assigned. The proxying function $M_\rho$ is always invoked to map $\chi$ to a unique proxy API $\chi'$.
$$\frac{\omega = \omega_{PX}}{M_\rho\langle\chi\rangle = \chi', \chi' \Leftarrow \beta} \tag{14}$$

Since multiple actions may apply to a given API - provenance pair ($T_\chi$), we require a systematic way to combine them into a single consolidated action. To address this, we define an algebraic structure $(\omega, \circ, \omega_{AC})$ on the set of actions $\omega$, where $\circ : \omega \times \omega \longrightarrow \omega$ is a binary operator that combines actions, and $\omega_{AC}$ is the identity element in this operation.

The structure for the resultant monoid is as follows:

$$\circ(\omega_A, \omega_B) = \begin{cases} \omega_A & \text{if } (\omega_A = \omega_B) \\ \omega_A & \text{if } (\omega_B = \omega_{\text{AC}}) \\ \omega_B & \text{if } (\omega_A = \omega_{\text{AC}}) \\ \omega_{\text{BP}} & \text{if } (\omega_A = \omega_{\text{BP}} \vee \omega_B = \omega_{\text{BP}}) \\ \omega_{\text{PX}} & \text{if } (\omega_A = \omega_{\text{MO}} \wedge \omega_B = \omega_{\text{PX}}) \\ \omega_{\text{AL}} & \text{if } (\omega_A = \omega_{\text{AL}} \wedge \omega_B = \omega_{\text{MO}}) \\ \omega_{\text{AL}} & \text{if } (\omega_A = \omega_{\text{PX}} \wedge \omega_B = \omega_{\text{AL}}) \end{cases} \tag{15}$$

The **MONITOR**($\omega_{MO}$) action when combined with the **PROXY**($\omega_{PX}$) action ($\omega_{\text{PX}}$), tracks the execution time only in the original API($\chi$) invoked from the `DevTools`. Additionally, the **ALTER**($\omega_{AL}$) action modifies the reported timings exclusively in the proxied API (the site's copy $\chi'$) when used in conjunction with the **PROXY**($\omega_{PX}$) action. A special case arises with proxy actions, where multiple copies of the same API may be created. In such scenarios, the final consolidated action always corresponds to the copy within the same execution scope as the original API.

# E SOURCETRACK Evaluation Algorithm

**Algorithm 1** SOURCETRACK Evaluation Algorithm

```
1:  total_coverage ← ∅
2:  no_change ← 0
3:  navigate_page()
4:  wait_for_load_event()
5:  SMAP_DB ← requested_sourcemaps
6:  for i ← 1 to 50 do
7:      wait_extra_5_seconds()
8:      x ← take_coverage()
9:      total_coverage ← total_coverage ∪ x
10:     common_coverage ← total_coverage ∩ x
11:     if x ∩ total_coverage = ∅ then
12:         no_change ← no_change + 1
13:     else
14:         no_change ← 0
15:     end if
16:     if no_change = 10 then
17:         break
18:     end if
19: end for
20: for each s ∈ SMAP_DB do
21:     request_sourcemap(s)
22:     wait_for_sourcemap_load()
23:     x ← take_coverage()
24:     if x ⊅ common_coverage and x ⊈ total_coverage then
25:         SOURCETRACK_flag ← true
26:     end if
27: end for
```

# F Advanced CONSPAM

```
1  // Main Thread
2  const myWorker = new Worker('worker.js');
3  // Send a message to the worker every second
4  setInterval(() => {
5      myWorker.postMessage('Ping from main thread');},
           1000);
6  // Listen for messages from the worker
7  myWorker.onmessage = function (event) {
8      let element = document.getElementById("
           devtools_status");
9      element.textContent = event.data;
10 };
11 // Worker Thread (worker.js)
12 onmessage = function (event) {
13     const threshold = 500;
14     const start = performance.now();
15     for (let i = 0; i < 50000; i++) {
16         console.log("DEVTOOLS");
17         console.clear();}
18     const time = performance.now() - start;
19     if (time > threshold) {
20         postMessage('OPEN');}
21     else {
22         postMessage('CLOSED');}
23 };
```

**Listing 7: Web Worker based devtools detection**