VisibleV8: In-browser Monitoring of JavaScript in the Wild

Jordan Jueckstock
North Carolina State University
jjuecks@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

ABSTRACT

Modern web security and privacy research depends on accurate measurement of an often evasive and hostile web. No longer just a network of static, hyperlinked documents, the modern web is alive with JavaScript (JS) loaded from third parties of unknown trustworthiness. Dynamic analysis of potentially hostile JS currently presents a cruel dilemma: use heavyweight in-browser solutions that prove impossible to maintain, or use lightweight inline JS solutions that are detectable by evasive JS and which cannot match the scope of coverage provided by in-browser systems. We present **VisibleV8**, a dynamic analysis framework hosted inside V8, the JS engine of the Chrome browser, that logs native function or property accesses during any JS execution. At less than 600 lines (only 67 of which modify V8's existing behavior), our patches are lightweight and have been maintained from Chrome versions 63 through 72 without difficulty. VV8 consistently outperforms equivalent inline instrumentation, and it intercepts accesses impossible to instrument inline. This comprehensive coverage allows us to isolate and identify 46 JavaScript namespace artifacts used by JS code in the wild to detect automated browsing platforms and to discover that 29% of the Alexa top 50k sites load content which actively probes these artifacts.

ACM Reference Format:
Jordan Jueckstock and Alexandros Kapravelos. 2019. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of IMC'19*. ACM, New York, NY, USA, 15 pages. https://doi.org/TBA

1 INTRODUCTION

"Software is eating the world" [16], and that software is increasingly written in JavaScript (JS) [9, 10]. Computer applications increasingly migrate into distributed, web-based formats, and web application logic increasingly migrates to client-side JS. Web applications and services collect and control vast troves of sensitive personal information. Systematic measurements of web application behavior provide vital insight into the state of online security and privacy [35]. With modern web development practices depending heavily on JS for even basic functionality, and with increasingly rich browser APIs [50] providing an inviting attack surface for fingerprinting [13] and tracking [22], effective security and privacy measurement of the web must include some degree of JS behavioral analysis.

As if dealing with the quirky, analysis-hostile dynamism of JS itself were not enough, researchers are also locked in an arms race with evasive and malicious web content that frequently cloaks [24] itself from unwanted clients. The ad-hoc approaches to JS instrumentation common in the literature suffer obvious shortcomings. Heavy-weight systems built by modifying a browser's JS engine can provide adequate coverage and stealth, but suffer from high development costs (on the order of thousands of lines of C/C++ code [33, 41, 53]) and poor maintainability, with patches rarely or never ported forward to new releases. Lighter-weight systems built by injecting in-band, JS-based instrumentation hooks directly into the target application's namespace avoid these pitfalls but suffer their own drawbacks: structural and policy limits on coverage, and vulnerability to well-known detection and subversion techniques (Section 2).

We argue in this paper that a **maintainable** in-browser JS instrumentation that matches or exceeds in-band equivalents in coverage, performance, and stealth is possible. As proof, we present **VisibleV8** (VV8): a transparently instrumented variant of the Chromium browser[1] for dynamic analysis of real-world JS that we have successfully maintained across eight Chromium release versions (63 to 72). VV8 lets us passively observe *native* (i.e., browser-implemented) API feature usage by popular websites with fine-grained execution context (security origin, executing script, and code offset) regardless of how a script was loaded (via static script tag, dynamic inclusion, or any form of `eval`). Native APIs are to web applications roughly what system calls are to traditional applications: security gateways through which less privileged code can invoke

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IMC'19, October 21–23, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN TBA…$TBA
https://doi.org/TBA

[1] Chromium is the open-source core of Chrome, equivalent in functionality but lacking Google's proprietary branding and service integration. We use the names interchangeably in this paper.

more privileged code to access sensitive resources. As such, VV8 provides a JS analog to the classic Linux `strace` utility. VV8 can be used interactively like any other browser, but is primarily intended for integration with automated crawling and measurement frameworks like OpenWPM [22].

We demonstrate VV8 by recording native feature usage across the Alexa top 50k websites, identifying feature probes indicative of bot detection, and analyzing the extent of such activity across all domains visited. Our collection methodology takes inspiration from Snyder et al. [49], using an automated browser instrumentation framework to visit popular domains, to randomly exercise JS-based functionality on the landing page, and to collect statistics on JS feature usage. Our identification and analysis of bot detection artifacts used in the wild showcases VV8's unique advantages over traditional JS instrumentation techniques: improved stealth in the face of evasive scripts, and universal property access tracking on native objects that do not support proxy-based interposition.

Our contributions include:

- We demonstrate that state-of-the-art inline JS instrumentation fails to meet classical criteria for reference monitors [15] and cannot prevent evasive scripts from deterministic detection of instrumentation.
- We present the first maintainable in-browser framework for transparent dynamic analysis of JS interaction with native APIs. Using instrumented runtime functions and interpreter bytecode injection, VV8 monitors native feature usage from inside the browser without disrupting JIT compilation or leaving incriminating artifacts inside JS object space. We will open source our patches and tools upon publication.
- We use VV8's universal property access tracing to discover non-standard features probed by code that detects "bots" (i.e., automated browsers, such as those used by researchers). We find that on 29% of the Alexa top 50k sites, at least one of 49 identified bot artifacts is probed. This is clear evidence that web measurements can be affected by the type of browser or framework that researchers use.

2 BACKGROUND & MOTIVATION

2.1 Trends and Trade-offs

The state of the art in web measurements for security and privacy relies on full browsers, not simple crawlers or browser-simulators, to visit web sites. Experience with the OpenWPM framework [22] indicated measurements from standard browsers (e.g., Firefox, Chromium) to be more complete and reliable than those from lightweight headless browsers (e.g., PhantomJS [7]). However, the question of how best to monitor and measure JS activity within a browser remains open. Assuming an open-source browser, researchers can modify the implementation itself to provide in-browser (i.e., *out-of-band*) JS instrumentation. Alternatively, researchers can exploit the flexibility of JS itself to inject language-level (i.e., *in-band*) instrumentation directly into JS applications at run-time.

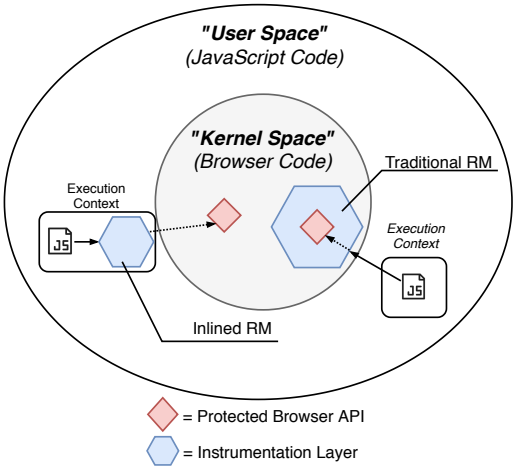

Figure 1: Reference monitors in traditional OS & application security

We provide a summary of recent security and privacy related research that measured web content using JS instrumentation in Table 1. Note that here "taint analysis" implies "dynamic analysis" but additionally includes tracking tainted data flows from source to sink. Fine-grained taint analysis is a heavy-weight technique, as is comprehensive forensic record and replay, so it is not surprising that these systems employed out-of-band (in-browser) implementations in native (C/C++) code. Lighter weight methodologies that simply log (or block) use of selected API features have been implemented both in- and out-of-band, but the in-band approach is more popular, especially in more recent works.

2.2 Fundamental Criteria

The problem of monitoring untrusted code dates to the very dawn of computer security and inspired the concept of a *reference monitor* [15], a software mediator that intercepts and enforces policy on all attempted access to protected resources. The traditional criteria of correctness for reference monitors are that they be **tamper proof**, be always invoked (i.e., provide **complete coverage**), and be **provably correct**, though this last element may be lacking in practical implementations. For security and privacy measurements, we add the additional criterion of **stealth**: evasive or malicious code should not be able to trivially detect its isolation within the reference monitor and hide its true intentions from researchers, since such an evasion could compromise the integrity of the derived results.

System	Implementation	Role	Problem	Platform	Availability
OpenWPM [21, 22, 38][1]	In-band	Dynamic analysis	Various	Firefox SDK[2]	●
Snyder *et al.*, 2016 [49]	In-band	Dynamic analysis	Attack surface	Firefox SDK	○
FourthParty [35]	In-band	Dynamic analysis	Privacy/tracking	Firefox SDK	●
TrackingObserver [45]	In-band	Dynamic analysis	Privacy/tracking	WebExtension	●
JavaScript Zero [47]	In-band	Policy enforcement	Side-channels	WebExtension	●
Snyder *et al.*, 2017 [50]	In-band	Policy enforcement	Privacy/tracking	Firefox SDK	○
Li *et al.*, 2014 [34]	Out-of-band	Dynamic analysis	Malware	Firefox (unspecified)	○
FPDetective [13]	Out-of-band	Dynamic analysis	Privacy/tracking	Chrome 32[3]	●
WebAnalyzer [48]	Out-of-band	Dynamic analysis	Privacy/tracking	Internet Explorer 8	○
JSgraph [33]	Out-of-band	Forensic record/replay	Malware/phishing	Chrome 48	◒[4]
WebCapsule [41]	Out-of-band	Forensic record/replay	Malware/phishing	Chrome 36	●
Mystique [19]	Out-of-band	Taint analysis	Privacy/tracking	Chrome 58	●
Lekies *et al.* [31, 32]	Out-of-band	Taint analysis	XSS	Chrome (unspecified)	○
Stock *et al.* [51]	Out-of-band	Taint analysis	XSS	Firefox (unspecified)	◒
Tran *et al.*, 2012 [53]	Out-of-band	Taint analysis	Privacy/tracking	Firefox 3.6	○

[1] Including only OpenWPM usage depending on JS instrumentation
[2] Supported only through Firefox 52 (end-of-life 2018-09-05)
[3] Used both PhantomJS and Chrome built from a common patched WebKit
[4] Binaries only

Table 1: Survey of published JS instrumentation systems

A classic example of a practical reference monitor is an operating system *kernel*: it enforces access control policies on shared resources, typically using a *rings of protection* scheme (Figure 1) assisted by hardware. In order to enforce security policies like access controls and audits, the kernel must run in a privileged ring. Untrusted *user* code runs in a less-privileged ring, where the kernel (i.e., the reference monitor) can intercept and thwart any attempt to violate system policy. Alternatively, *inlined reference monitors* (IRMs) [23] attempt to enforce policy while cohabiting the user ring with the monitored application, typically by rewriting and instrumenting the application's code on the fly at load or run time.

On the web, the browser and JS engine provide the equivalent of a kernel, while JS application code runs in a "user ring" enforced by the semantics of the JS language. JS instrumentation in general is a kind of reference monitor; implemented in-band, it constitutes an IRM. We argue that the JS language's inherent semantics and current implementation details make it impossible to build sound, efficient, general-purpose IRMs in JS on modern web browsers.

2.3 The Case Against In-Band JS Instrumentation

The standard approach to in-band JS instrumentation, which we call "prototype patching," is to *replace* references to target JS functions or objects with references to instrumented wrapper functions or proxy objects[2]. The wrappers can access the original target through references captured in a private scope inaccessible to any other code. Note that the target objects themselves are not replaced or instrumented, only the references to them (a potential pitfall highlighted in prior work [37]).

Structural Limits. The JS language relies heavily on a *global* object (`window` in browsers) which doubles as the top level namespace. There is no mutable root reference to the global object, and thus no way to replace it with a proxy version. Specific properties of the global object may be instrumented selectively, but this process naturally requires *a priori* knowledge of the target properties. In-band instrumentation cannot be used to collect **arbitrary** global property accesses, as required for our methodology in Section 5. This limitation means that in-band JS instrumentation fails the **complete coverage** criterion.

Policy Limits. Not all Chrome browser API features can be patched or wrapped **by design policy**. These features can be identified using the WebIDL [12] (interface definition language) files included in the Chromium sources. For Chrome 64, these files defined **5,755** API functions and properties implemented for use by web content (there are more available only to internal test suites). **21** are marked *Unforgeable* and cannot

[2]Other forms of JS IRM exist, like nested interpreters [18, 52] and code-rewriting systems [20], but these have not yet proven fast enough for real-world measurement work.

be modified at all. Notably, this set includes `window.location` and `window.document`, preventing in-band instrumentation of arbitrary-property accesses on either of these important objects. Again, such a restriction would have eliminated many of our results in Section 5.

Patch Detection. Prototype patches of native API functions (or property accessors) can be detected directly and thus fail the criterion of **stealth**. JS functions are objects and can be coerced to strings. In every modern JS engine, the resulting string reveals whether the function is a true JS function or a binding to a native function. Patching a native function (e.g., `window.alert`) with a non-native JS wrapper function is a dead giveaway of interposition.

```
/* from https://cdn.flashtalking.com/xre/275/
2759859/1948687/js/j-2759859-1948687.js */
/* (all variable names original) */
var badWrite = !(document.write instanceof
     Function && ~document.write.toString().
                indexOf('[native code]'));

/* (later on, among other logic checks) */
if (badWrite || o.append) {
    o.scriptLocation.parentNode.insertBefore(
    /* omitted for brevity */);
} else {
    document.write(div.outerHTML);
}
```

Listing 1: Prototype patch evasion in the wild

The function-to-string probe has been employed to detect fingerprinting countermeasures [55] and appears commonly in real-world JS code. In many cases, such checks appear strictly related to testing available features for browser compatibility. But there also exist cases like Listing 1, in which the script changes its behavior in direct response to a detected patch. Function-to-string probe evasions abound, from the obvious (patch the right `toString` function, too) to the subtle. In Listing 2, the `"[native code]"` string literal in the patch function appears in the output of `toString` and will fool a sloppy function-to-string probe that merely tests for the presence of that substring.

Let us assume a "perfect" patching system invisible to `toString` probes has been used to instrument `createElement`, the single most popular browser API observed in our data collection across the Alexa 50k (Section 4). Such a patch is still vulnerable to a probe that exploits JS's type coercion rules with a *Trojan argument* to detect patches on the call stack at runtime (Listing 3).

For brevity, the provided proof-of-concept calls the `Error` constructor, which could itself be patched, but there are other ways of obtaining a stack trace in JS. The Byzantine complexity of JS's pathologically dynamic type system offers many opportunities for callback-based exposure of patches and proxies via stack traces. Here prototype patches face a cruel

```
/* from https://clarium
    .global.ssl.fastly.net ("..." comment means
     irrelevant portions elided for brevity) */
patchNodeMethod: function(a) {
    var b = this,
        c = Node.prototype[a];
    Node.prototype[a] = function(a) {
        " [native code] ";
        var d = a.src || "";
        return /* ... */
            c.apply(this, arguments)
    }
    /* ... */
}
```

Listing 2: Function patches hiding in plain sight

```
function paranoidCreateElement(tag) {
 return document.createElement({
  toString: function() {
   var callers
        = new Error().stack.split('\n').slice(1);
   if (/at
       paranoidCreateElement /.test(callers[1])) {
    return tag; /* no patch */
   } else {
    throw new
         Error("evasive action!"); /* patched! */
   }},});}
```

Listing 3: Trojan argument attack (Chrome variant)

dilemma: either invoke the `toString` operation and open the gates to side-effects (allowing detection and evasion), or refuse to invoke it and break standard JS semantics (allowing detection and evasion).

Patch Subversion. Finally, prototype patches can be subverted through abuse of `<iframe>` elements. Each `<iframe>` is an independent browser window with its own global object, unaffected by prototype patches in other frames. We have observed scripts exploiting this fact to evade patches they have detected.

```
/* from https://an.
    yandex.ru/resource/context_static_r_4583.js */
/* (some names changed
     for clarity; cachedJSON is initially null) */
if (window.JSON
    && a.checkNativeCode(JSON.stringify)
    && a.checkNativeCode(JSON.parse))
    return window.JSON;
if (!cachedJSON) {
    var t = getInjectedIFrameElement();
    cachedJSON = t.contentWindow.JSON;
    var e = t.parentNode;
    e.parentNode.removeChild(e)
}
```

Listing 4: Patch subversion in the wild

In Listing 4, the script resorts to frame injection to avoid patched JSON encoding/decoding functions. Short of blocking the creation of all `<iframe>`s (and thus breaking much of the web), the only defense against this evasion is to ensure that prototype patching code is always run inside newly created frames before any other code can access its contents. Such an invariant can be established through some browser automation and debugging frameworks, like Chrome DevTools. But the web extension APIs of both Firefox and Chrome, as currently implemented, do not provide such a guarantee to extension authors, effectively crippling privacy or security (or research [45, 47, 49]) extensions that rely on this technique. (The author of a prior related work [50] reported this bug to both the Firefox [2] and Chrome [4] projects; neither report has been resolved at the time of this writing, and we have confirmed that Chrome 71 is still affected.)

2.4 Summary

Robust JS instrumentation systems must be tamper proof, must provide comprehensive coverage, and must not introduce unmistakable identifying artifacts. At present, JS language semantics and browser implementation details prevent in-band implementations from meeting these criteria. We believe that security-critical JS instrumentation, like traditional operating system auditing and enforcement logic, belongs in "kernel space," i.e., within the browser implementation itself. But to be useful such a system must be cost-effective, both to develop and to maintain.

3 SYSTEM ARCHITECTURE

We present cost-effective out-of-band JS instrumentation via **VisibleV8**, a variant of Chrome that captures and logs traces of all native API accesses made by any JS execution during browsing. Here we explain VV8's internal design, relate our experience maintaining it over several Chrome update cycles, evaluate its raw performance against several alternatives, and describe the data collection and analysis system we have built around VV8 to demonstrate its potential for real-world measurement work.

3.1 Chromium/V8 Internals

Chromium is a massive project (over 20 million lines of code at time of writing), actively developed, and frequently updated. Fortunately, the Chromium browser's architecture is modular, as is the design of its V8 JS engine, so we can restrict our changes to a tiny subset of the entire browser code base.

Modern versions of V8 handles JS parsing and execution via the *Ignition* bytecode interpreter and the *TurboFan* JIT compiler (Figure 2). Ignition parses JS source code, generates bytecode, and executes bytecode; its design is optimized for low latency, not high throughput. When run-time statistics indicate that JIT compilation is desired, TurboFan aggressively optimizes and translates the relevant bytecode into native machine code.

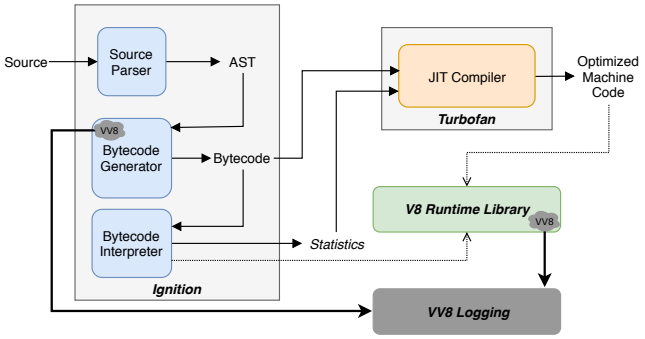

Figure 2: V8 architecture with VV8's additions

Ignition and TurboFan rely on a large supporting run-time library (RTL) that includes a foreign-function interface allowing JS code to call into native (i.e., C++) code via type-safe API function bindings created by the hosting application (e.g., Chrome). Functionality not implemented by JS code or by V8 built-in code (e.g., `Math.sqrt`) must use an API binding to native browser code. V8's RTL thus forms a software-enforced JS/native boundary not unlike a system call boundary in a traditional OS kernel.

In V8's 2-stage implementation, source code translation happens once, and the run-time behavior of that source code is fixed (modulo JIT compiler optimizations) at the point of bytecode generation. This predictable workflow keeps our patches to V8 small and self-contained.

3.2 VisibleV8 Implementation

VV8 intercepts and logs all *native* API access from JS execution during browsing. Native API accesses comprise API function calls and all get and set operations on properties of JS objects constructed by native (i.e., C++) code in the browser itself. The resulting traces include context information (e.g., the source code location triggering this event), feature names, and some abbreviated activity details like function call parameters and the value being stored during property writes. Our patches are small: 67 lines of code changed or added inside V8 itself to insert our instrumentation hooks, and 472 lines of new code for filtering and logging.

Instrumenting Native Function Calls. All foreign function calls through the JS/native boundary are routed through a single V8 runtime function that handles the transition from the bytecode interpreter to native execution and back. By adding a single call statement invoking our centralized tracing logic to this C++ function, we can hook all such calls made under Ignition bytecode interpretation. However, when the

bytecode is JITted to optimized machine code, one of TurboFan's hundreds of optimization transforms will reduce the call through that hooked runtime function into a more direct alternative. This transformation would disrupt our function call tracing, so we disabled that single specific reduction, leaving the rest of the JIT compiler untouched. This removal slows V8 down by **1.3%** on the Mozilla Dromaeo micro-benchmark suite (Section 3.3), with margins of error near 1% as well. For the cost of two trivial code modifications and very modest overhead, we gain full visibility of JS calls into native API bindings under both bytecode interpretation and JITted code execution.

Instrumenting Native Property Accesses. V8 provides no similarly convenient single choke-point from which all native object property accesses can be observed. Property access is a frequent and complicated operation in JS, and V8 has multiple fast-paths for different access scenarios. Therefore, we target not the *execution* of any bytecode here, but rather the *generation* of property-accessing bytecode.

JS code entering V8 is first processed by Ignition's source-to-bytecode compiler before any execution. The bytecode compiler uses a classic syntax-directed architecture. First, a parser constructs an abstract syntax tree (AST) from JS source code. Then, the bytecode generator walks this AST while generating bytecode to implement the semantics required by the original JS syntax.

We instrument property access to native objects by patching the bytecode generator. Specifically, we add statements to the AST visitor logic for property get and set expressions to emit additional bytecode instructions in each case. These instructions call a custom V8 runtime function containing our tracing logic. Since such runtime calls are effectively opaque black boxes to the TurboFan optimizer, our hook instruction cannot be automatically optimized away during JIT compilation. So our injected hook's semantics are preserved from bytecode generation, through interpretation and JIT compilation, to optimized machine code execution. For completeness, we also hook the built-in implementation of `Reflect.get` and `Reflect.set` in the RTL using the same approach as for native function calls. Thus, we also capture property accesses via calls to the JS Reflection API, not only through member-access expressions.

Capturing Execution Context. All of our hooks, whether in runtime functions or in injected bytecode, call into our central tracing logic. Written in C++ and compiled into V8, this code is responsible for filtering events and capturing execution context information for the trace log.

Native API calls are always logged. But since our property-access hooks intercept *all* syntax- and reflection-based property accesses, we must filter those events. We log only property accesses on native objects as indicated by V8's internal object metadata API. V8 treats the JS global object as a unique special case, but we treat it as a standard native object for logging purposes.

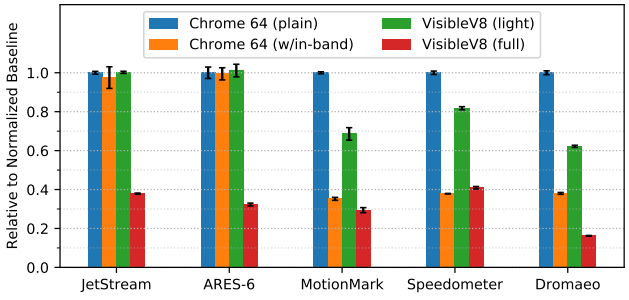

Figure 3: Instrumentation performance on BrowserBench.org [1] and Mozilla Dromaeo [3]

Fine-grained feature-usage analysis requires a significant amount of execution context to be logged along with each function call or property access. We link feature usage not just with a visited domain, but also with the active security origin, active script, and location within that script. We use V8's C++ APIs to extract the invoking script and location from the top frame of the JS call stack and the security origin from the `origin` property of the active global object. V8 and Blink sometimes execute internal JS code in a non-Web context, where the global object has no `origin` property or it has a non-string value. In this case an "unknown" origin is recorded (and we can later discard this activity from our analysis). The visit domain (i.e., from the URL displayed in the browser's address bar) is associated with the log during post-processing.

Logging Trace Data. Reliably recording JS trace data at low cost introduced its own engineering challenges. Repeatedly looking up and logging identical context for successive events wastes CPU time, I/O bandwidth, and storage space. We therefore track execution context state (such as active script) over time, and log it only when it has changed since the last logged event. This optimization introduces state-tracking and synchronization issues.

Chrome uses multiple processes and threads to achieve good performance and strong isolation. Even with just a single browser tab open, JS code can be executing simultaneously across multiple threads. To keep our traces coherent, we must track context and log events on a per-thread basis. To store our separate trace log streams without races or synchronization bottlenecks, we create per-thread log files.

3.3 Performance

With every JS object property access intercepted and possibly logged, we expected VV8 to be significantly slower than stock Chrome in the worst case. We verified this expectation by measuring the overhead by benchmarking a set of Chrome and

VV8 variants with both the WebKit project's BrowserBench [1] and Mozilla's Dromaeo [3]. Unless otherwise noted, tests were performed under Linux 4.19.8 on an Intel Core i7-7700 (4 cores, 3.6GHz) with 16GiB RAM and an SSD. See Figure 3.

We tested four variants of Chrome 64 for Linux, including a baseline variant with no JS instrumentation at all (**plain**). We include two VV8 builds: the complete system as described above (**full**) and a variant with property access interception disabled (**light**). VV8-light is roughly equivalent in coverage and logging to our final variant, stock Chrome running a custom prototype-patching extension roughly equivalent to the (unreleased) instrumentation used by Snyder et al. to measure browser feature across the Alexa top 10K [49]. This last variant (**w/in-band**) attempts to provide an apples-to-apples comparison of in-band and out-of-band instrumentation instrumenting a comparable number of APIs (functions only) and recording comparably rich trace logs. All Chrome builds were based on the Chrome 64 stable release for Linux and use the same settings and optimizations.

BrowserBench tests the JS engine in isolation (JetStream, ARES-6), JS and the DOM (Speedometer), and JS and Web graphics (MotionMark). VV8-light either meets or decisively beats its in-band equivalent in every case. VV8-full consistently suffers 60% to 70% overhead *vs.* the baseline, but on the whole-browser tests (Speedometer, MotionMark) it performs comparably to the in-band variant, which captures significantly less data (i.e., no property accesses). These numbers match our experience interacting with VV8, where we observe it providing acceptable performance on real-world, JS-intensive web applications like Google Maps. Significantly, VV8-**full** on the workstation compared favorably (i.e., equal or better BrowserBench scores) to Chrome 64 **plain** on a battery-throttled laptop running Linux 4.18.15 on an Intel Core i7-6500 (2 cores, 2.5GHz) with 16GiB RAM and an SSD.

The Mozilla Dromaeo suite of micro-benchmarks focuses exclusively on JS engine performance. It avoids the browser's layout and render logic as much as possible, and reveals more slowdowns for all instrumented variants. Dromaeo's recommended test suite comprises 49 micro-benchmarks, too many to effectively visualize in a single figure, so we provide only the reported aggregate score.[3] VV8-light still handily outperforms in-band instrumentation, but VV8-full is significantly slower than the baseline (6x in aggregate). VV8-full showed a wide range of performance on Dromaeo micro-benchmarks, from six showing no slowdown at all to three showing pathological slowdown over 100x.

[3]The full results can be viewed at http://dromaeo.com/?id=276022,276023,276026,276027; the four columns are Chrome (plain), Chrome (w/in-band), VisibleV8 (light), and VisibleV8 (full), respectively.

3.4 Maintenance & Limitations

Thanks to the small size and minimal invasiveness of VV8's patches, maintenance has thus far proved inexpensive. Development began on Chrome 63, then easily transitioned to Chrome 64, which was used for primary data collection. We have since ported our patches through Chrome 72 and encountered only trivial issues in the process (e.g., whitespace changes disrupting patch merge, capitalization changes in internal API names).

Our trace logs must be created on the fly as new threads are encountered. Since the Chrome sandbox prevents file creation, we currently run VV8 with the sandbox disabled as an expedience. In production, we run VV8 inside isolated Linux containers, mitigating the loss of the sandbox somewhat. Future development will include sandbox integration should the need arise.

Past work [39, 40] on fingerprinting JS engines indicates that sophisticated adversaries could use *relative* scores across micro-benchmarks as a side-channel to identify VV8. However, such benchmarks and evasions would be detectable in VV8's trace logs, and JS timing side-channel attacks can be disrupted [17, 47]. In any case, it is unlikely that an adversary sophisticated enough to fingerprint VV8 in the wild would not also be able to fingerprint in-band instrumentation, which also shows measurable deviation from baseline performance.

Furthermore, we expect to improve VV8 performance in future iterations by exploring asynchronous log flushing, log-filtering tests placed in the injected bytecode (where they can be JIT optimized), and cheaper forms of context tracking.

3.5 Collection System

To collect data at large scale using VV8, we built the automated crawling and post-processing system diagrammed in Figure 4. Worker nodes (for collection, post-processing, and work queues) are deployed across a Kubernetes cluster backed by 80 physical CPU cores and 512GiB of RAM distributed across 4 physical servers. Initial jobs (i.e., URLs to visit) are placed in a Redis-based work queue to be consumed by collection worker nodes. Post-processing jobs (i.e., archived logs to parse and aggregate) are placed in another work queue to be consumed by post-processing worker nodes. Collection metadata and trace logs are archived to a MongoDB document store. Aggregate feature usage data is stored in a PostgreSQL RDBMS for analytic queries.

The collection worker node Docker image contains the VV8 binary itself and a pair of accompanying programs written in Python 3: Carburetor and Manifold. Carburetor is responsible for fueling VV8: using the Chrome DevTools API to open a tab, navigate to a URL, and monitor progress of the job. Manifold handles the byproducts of execution, compressing and archiving the trace log files emitted during automated browsing.

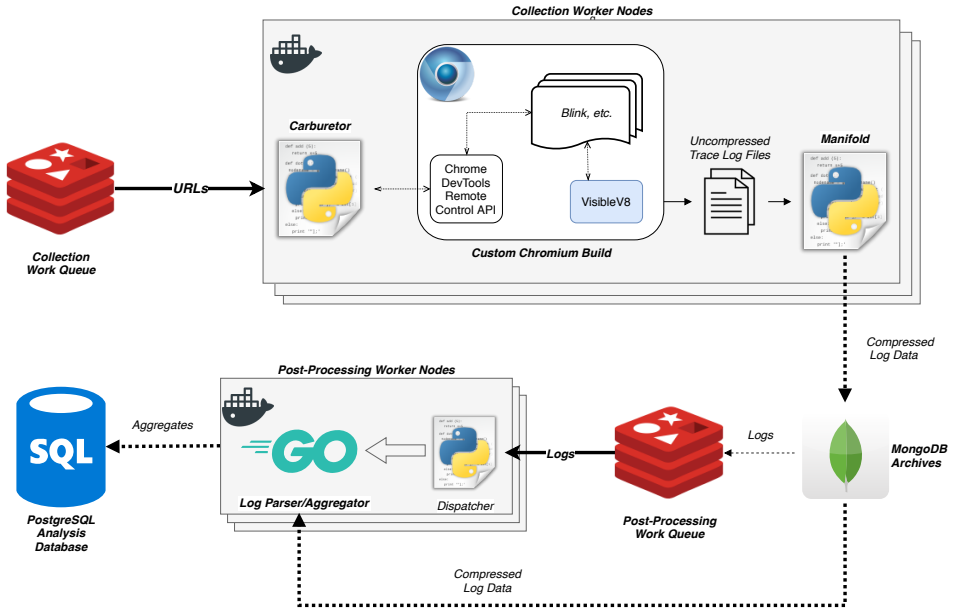

Figure 4: The complete data collection and post-processing system

The post-processor worker node Docker image contains a work queue dispatcher and the main post-processor engine. The dispatcher interfaces with our existing work queue infrastructure and is written in Python 3. The post-processor engine is written in Go, which provides ease-of-use comparable to Python but significantly higher performance.

4 DATA COLLECTION

4.1 Methodology

Overview. We collected native feature usage traces and related data by automating VV8 via the Chrome DevTools interface to visit the Alexa top 50k web domains. We began each visit to *DOMAIN* using the simple URL template `http://DOMAIN/`. We visited each domain in our target list 5 times (see below); each planned visit constituted a *job*. We recorded headers and bodies of all HTTP requests and responses along with the VV8 trace logs. Trace log files were compressed and archived immediately during jobs, then queued for post-processing. Post-processing associated logs with the originating job/domain and produced our analytic data set.

User Input Simulation. Simply visiting a page may result in much JS activity, but there is no guarantee that this activity is representative. The classic challenge of dynamic analysis—input generation—rears its head. We borrowed a solution to this problem from Snyder et al. [49]: random "monkey testing" of the UI using the open source `gremlins.js` library [6]. To preserve some degree of reproducibility, we used a deterministic, per-job seed for `gremlins.js`'s random number generator.

Once a page's DOM was interaction-ready, we unleashed `gremlins.js` for 30 seconds. We blocked all main-frame navigations if they led to different domains (e.g., from *example.com* to *bogus.com*). When allowing intra-domain navigation (e.g., from *example.com* to *www.example.com*), we stopped counting time until we loaded the new destination and resume the monkey testing. We immediately closed any dialog boxes (e.g., *alert()*) opened during the monkey testing to keep JS execution from blocking. This 30 second mock-interaction procedure was performed 5 times, independently, per visited domain. (Snyder et al. [49] arrived at these parameters experimentally.)

4.2 Data Post-Processing

We parsed the trace logs to reconstruct the execution context of each recorded event and to aggregate results by that context. The resulting output included all the distinct scripts encountered and aggregate feature usage tuples.

Script Harvesting. VisibleV8 records the full JS source of every script it encounters in its trace log (exactly once per log). We extracted and archived all such scripts, identifying them by *script hash* and *lexical hash*. Script hashes are simply the SHA256 hash of the full script source (encoded as UTF-8); they served as the script's unique ID. Lexical hashes were computed by tokenizing the script and SHA256-hashing the sequence of JS token type names that result. These are useful because many sites generate "unique" JS scripts that differ only by timestamps embedded in comments or unique identifiers in string literals. Such variants produced identical lexical hashes, letting us group variant families.

Feature Usage Tuples. We recorded a feature usage tuple for each distinct combination of log file, visit domain, security origin, active script, access site, access mode, and feature name. The *log file* component let us distinguish collection runs. The *visit domain* is the Alexa domain originally queued for processing. The *security origin* was the value of `window.origin` in the active execution context, which may be completely different from the visit domain in the context of an `<iframe>`. The *active script* is identified by script hash. The *access site* is the character offset within the *script* that triggered this usage event. The *access mode* is how the feature was used (`get`, `set`, or `call`). The *feature name* is a name synthesized from the name of the receiver object's constructor function (effectively its type name) and the name of the accessed member of that object (i.e., the property or method name).

4.3 Results

Success and Failure Rates. Our methodology called for 5 visits to the Alexa 50k, so the whole experiment consisted of 250,000 distinct jobs. Successful jobs visited the constructed URL, launched `gremlins.js`, and recorded at least 30 seconds of pseudo-interaction time. Jobs resulting in immediate redirects (by HTTP or JS) to a different domain before any interaction began were deemed "dead ends".

From job status we extract the per-domain coverage listed in Table 2. For "active" domains, all 5 jobs succeed and we observed native JS API usage. For "silent": all 5 jobs succeed, but we observed no native JS API usage. For "facade": all 5 jobs were "dead ends" (i.e., the domain is an alias). All of the above are considered "successful" domains.

Some domains were "broken," with all 5 jobs failing; a tiny number were "inconsistent," with a mix of failed/succeeded jobs. This failure rate is not out of line with prior results crawling web sites. Snyder et al. [49] reported a lower per-domain failure rate (2.7%), but this was over the Alexa top 10,000 only. On the other extreme, a recent measurement study by Merzdovnik et al. [36] reported successful visits to only about 100k out of the top Alexa 200k web domains.

Aggregate Feature Usage. Over the entire Alexa 50k, we observed 53% of Chrome-IDL-defined standard JS API features used at least once. Note that our observations comprise a *lower bound* on usage, since we did not crawl content requiring authentication (e.g., Google Documents). Most modern

Consistent	Success	Active	42,845	85.69%		
		Silent	1,702	3.40%	92.11%	
		Facade	1,508	3.02%		98.54%
		Broken	3,214	6.43%		
		Inconsistent	731	1.46%		
		TOTAL	50,000	100.00%		

Table 2: Final domain status after collection

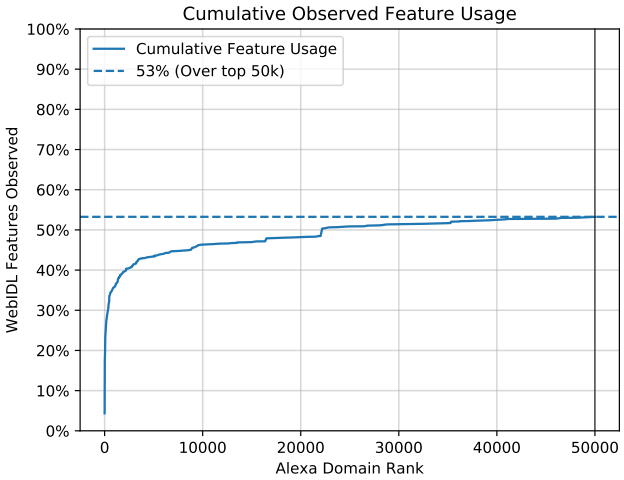

Figure 5: Cumulative feature use over the Alexa 50k

sites use JS heavily, but no site uses all available features. The plot in Figure 5 thus climbs steeply before leveling out into a gentle upward slope. The small but distinctive "cliffs" observed at rocket-league.com (Alexa 16,495) and noa.al (Alexa 22,184) are caused by large clumps of SVG-related features being used for the first time.

5 BOT DETECTION ARTIFACTS

Modern websites adapt their behavior based on the capabilities of the browser that is visiting them. The identification of a specific browser implementation is called *user-agent fingerprinting* and it is often used for compatibility purposes. To provide a case study of VV8's unique abilities, we use it to automatically discover artifacts employed by a form of user-agent fingerprinting used by some websites in the wild detect automated browser platforms.

The technique we study exploits the presence of distinctive, non-standard features on API objects like `Window` (which doubles as the JS global object) and `Navigator` as provided by automated browsers and browser simulacra. (Since even modern search engine indexers need some degree of JS support[11], we do not consider mechanisms used to identify "dumb," non-JS-executing crawlers like `wget`.) Here VV8's ability to trace native property accesses without *a priori* knowledge of the properties to instrument sets it apart from in-band instrumentation, which cannot wrap a proxy around the global object or the unforgeable `window.document` property. Note that "native API property access" here means a property access on an object that crosses the JS/native API boundary, regardless of whether or not that specific property is standardized or even implemented.

Bot detection is a special case of user-agent fingerprinting, where "bots" are automated web clients not under the direct

Artifact Name	*Bot Platform Indicated*
`Window._phantom`	PhantomJS [7]
`Window.webdriver`	Selenium [8] WebDriver
`Window.domAutomation`	ChromeDriver (WebDriver for Chrome)

Table 3: Bot detection seed artifacts

control of a human user (e.g., web scrapers). Bots may be a nuisance or even a threat to websites [25], and they may cause financial loss to advertisers via accidental (or intentional) impression and/or click fraud. If the visitor's user-agent fingerprint matches a known bot, a site can choose to "defend" itself against undesired bot access by taking evasive action (e.g., redirecting to an error page) [24]. Non-standard features distinctive to known bot platforms, then, constitute bot *artifacts*. We exploit VV8's comprehensive API-property-access tracing to systematically discover novel artifacts.

5.1 Artifact Discovery Methodology

We discover previously unknown bot artifacts by clustering the *access sites* (i.e., script offsets of feature accesses) for candidate features near those of known "seed" artifacts. The key insight underlying our approach is *code locality*: in our experience, artifact tests tend to be clustered near each other in user-agent fingerprinting code encountered across the web. We exploit this locality effect to automate the process of eliminating noise and identifying a small set of candidates for manual analysis.

Candidate Feature Pool. Before searching for artifacts, we prune our search space to eliminate impossible candidates. We eliminate features defined in the Chrome IDL files, since these are standard-derived features unlikely to be distinctive to a bot platform. We also eliminate features seen *set* or *called*: these are likely distinctive to JS libraries, not the browser environment itself. This second round of pruning is especially important because JS notoriously conflates its global namespace with its "global object." Thus, in web browsers, global JS variables are accessible as properties of the *window* object along with all the official members of the *Window* interface. Retaining only features we never see set or called eliminates significant noise (e.g., references to the *Window.jQuery* feature) from our pool of candidate features: from **7,928,522** distinct names to **1,907,499**.

Seed Artifact Selection. We further narrow our candidate pool using access site locality clustering around "seed" artifacts (Table 3). These features are among the most commonly listed in anecdotal bot detection checklists found in developer hubs like Stack Exchange [5], reflecting the popularity of Selenium's browser automation suite and the lighter-weight PhantomJS headless browser.

Candidate Artifact Discovery. With a pruned candidate feature pool and a set of seed artifacts in hand, we can automatically discover *candidate artifacts* by following these steps: (1) find all distinct access sites for seed artifacts in all archived scripts, (2) find all candidate feature access sites no more than 1,024 characters away from one of the located seed access sites and (3) extract the set of all candidate features whose access sites matched the seed locality requirement. From our initial set of **3** seed artifacts, the above process yields a set of **209** candidate artifacts (**0.01%** of the candidate pool) found near seed access sites in **7,528** distinct scripts (of which only **1,813** scripts were lexically-distinct).

Modern Browser Artifacts. We next eliminated from our candidates any artifacts found in a current, major web browser. We tested a total of nine browser variants manually: two for Chrome (v70 on Linux, v69 on macOS), three for Firefox (v63 on Linux, v62 on macOS, v63 on macOS), one for Safari (v12.0 on macOS), one for Edge (v17 on Windows 10), and two for Internet Explorer (v8 on Windows 7, v11 on Windows 10). In total we found **61** of the candidates present on at least one tested browser, leaving **148** candidates that might be indicative of a distinctive bot platform. The only **2** present on all 9 browsers were in fact standard JS (but not WebIDL-defined) features in the global object: `Object` and `Function`.

Manual Classification. The remaining 148 candidate artifacts we classified manually. Intuitively, if for every one of a candidate artifact's access sites there exists a data flow from that site to apparent data exfiltration or evasion logic, we consider that candidate a true artifact. If there exist benign or inconclusive examples, we conservatively assume the candidate is not a true artifact. (We also attempt to categorize false positives, but that process often depends on subjective judgment of programmer intent.)

To assist this process, we classified artifact access sites into 3 categories: **direct** if the feature name appears in the source code at the exact offset of the access site; **indirect** if the name appears only elsewhere in the code; and **hidden** if the name does not appear at all. A candidate found in few distinct scripts and accessed mostly via hidden, monomorphic access sites almost always proved to be a bot detection artifact. Conversely, candidates found far and wide and accessed mostly via direct or polymorphic sites usually proved to not be true bot artifacts.

Table 4 shows the breakdown of manual classifications. We identified a total of **49** artifacts (including our seeds) used exclusively, as far as we could tell, for bot detection. We identified 10 more that we did see used for bot detection activity but not exclusively so. (To avoid false positives, we exclude these "maybe" artifacts from our aggregate results.) An additional 19 appeared to be known or suspected fingerprinting artifacts of specific browsers or devices (e.g., standard features with vendor prefixes like `moz-` and WebGL information query constants).

Count	*Category*
3	Seed Bot Artifact
46	New Bot Artifact
10	Possible Bot Indicator
19	Device/Browser Fingerprint
46	Property Pollution/Iteration
11	Type Error/Misspelling
8	Missing Dependency
5	Other

Table 4: Candidate artifacts classified

Almost all of the remaining candidate artifacts appear to be side-effects of JS language quirks and sloppy programming. An example, extracted from a lightly obfuscated bot detection routine, explains some of the **46** artifacts we attribute to property pollution in iteration (Listing 5). This code iterates over an array of property names to check (in this case, all true bot artifacts). However, JS arrays intermingle indexed values with named properties, and this code fails to exclude properties (e.g., `findAll`)inherited from the array's prototype object. As a result, a single polymorphic access site within our clustering radius would access both true bot artifacts and unrelated array method names, bloating our initial candidate artifact pool with spurious features that had to be weeded out manually.

Fortunately, the small size of the candidate set combined with the insights provided by access site classification made this task straightforward and tractable. Other identifiable sources of noise in the final candidate set include obvious type errors or misspellings (**11**) and what appear to be missing dependencies (**8**).

5.2 Artifact Analysis Results

Across Visited Domains. Our trace logs recorded probes of at least one definite bot artifact during visits to **14,575** (**29%**) of the Alexa top 50k. This number includes artifact accesses from both monomorphic (sites accessing only one feature; 24%) and polymorphic access sites (those accessing more than one feature name; 5%). If we consider only monomorphic access

```
/* (deobfuscated excerpt) */
var d = ["_phantom
    ","__nightmare","_selenium","callPhantom
    ","callSelenium","_Selenium_IDE_Recorder"],
    e = window;
for (var l in d) {
    var v = d[l];
    if (e[v]) return v
}
```

Listing 5: Noisy artifact probing

Visit Domain	*Alexa Rank*
youtube.com	2
yahoo.com	6
reddit.com	7
amazon.com	11
tmall.com	13
weibo.com	21
google.de	23
ebay.com	45
mail.ru	50
stackoverflow.com	55

Table 5: Highest ranked visit domains probing identified bot artifacts

Origin Domain	*Visit Domains*
tpc.googlesyndication.com	10,291
googleads.g.doubleclick.net	3,980
ad.doubleclick.net	1,853
secure.ace.advertising.com	1,150
www.youtube.com	1,041
nym1-ib.adnxs.com	699
media.netseer.com	321
adserver.juicyads.com	175
openload.co	168
aax-us-east.amazon-adsystem.com	121

Table 6: Top security origin domains probing bot artifacts

sites, the number drops to 11,830 visited sites, which is under 24% of the top 50k. The modest size of that drop implies that most bot detection scripts, even if obfuscated, perform artifact probes on a one-by-one basis rather than through changing, loop-carried indirect member accesses. Table 5 shows the top 10 visited domains (by Alexa ranking at the time of data collection) on which bot artifact probes were detected.

Across Security Origin Domains. When we consider security origin domains as well as visit domains, we find that the majority (over **73%**) of bot artifact accesses happen inside third-party sourced `<iframe>`s, as is typical of advertisements, third-party widgets, and trackers. Here we are defining "first-party" as having a security origin domain containing the visit domain as a suffix and "third-party" as everything else. Using a stricter, exact-match definition like the browser's Same-Origin Policy would result in an even higher third-to-first-party ratio.

We found bot artifact probes in the contexts of **6,257** distinct security origin domains. Table 6 lists the top 10 origin domains for bot detection activity. Naturally, four of the top five are affiliated with Google's advertising platform.

Artifact Feature Name	*Visit Domains*	*Security Origins*
HTMLDocument.$cdc_asdjflasutopfhvcZLmcfl_	11,409	887
Window.domAutomationController	11,032	2,317
Window.callPhantom	10,857	5,088
Window._phantom	10,696	5,052
Window.awesomium	10,650	203
HTMLDocument.$wdc_	10,509	18
Window.domAutomation	7,013	2,674
Window._WEBDRIVER_ELEM_CACHE	6,123	1,803
Window.webdriver	2,756	1,832
Window.spawn	1,722	1,559
HTMLDocument.__webdriver_script_fn	1,526	1,390
Window.__phantomas	1,363	1,103
HTMLDocument.webdriver	1,244	529
Window.phantom	953	820
Window.__nightmare	909	628

Table 7: Most-probed bot artifacts

Scripts running in the context of the top domain, `tpc.google-syndication.com`, probed no less than **42** of our 49 confirmed artifacts (85%).

We believe most of these instances to be benign in intent. Advertisers have legitimate incentive to avoid paying for pointless ad impressions by blocking bots. But large-scale (i.e., automated) web measurement accuracy may become collateral damage in this arms race. The future is not bright for naive, off-the-shelf web crawling infrastructure.

Popular Artifacts. In Table 7 we list our 15 most popular (by visit domain cardinality) bot detection artifacts. Unsurprisingly, given our seed artifacts, most results appear associated with variants of Selenium and PhantomJS. But our locality search pattern also discovered artifacts of additional automation platforms: Awesomium, NightmareJS, and Rhino/HTMLUnit. The full list of discovered artifacts includes a superset of all the Selenium and PhantomJS artifacts tested for in the latest available version [54] of Fp-Scanner [55].

Most of the artifact names are highly suggestive and/or self-explanatory, with the single most common association being Selenium, but a few require explanation. The `$cdc_...` artifact is an indicator of ChromeDriver, as `$wdc_` is of WebDriver; notably, these are among the relative minority of artifacts found on non-global objects like `window.document`. `spawn` is an artifact of the Rhino JS engine, which is itself an indicator of the HTMLUnit headless browser system.

5.3 Case Studies

Explicit Bot Identification. Listing 6 shows part of a script loaded from http://security.iqiyi.com/static/cook/v1/cooksdk.js and observed on visits to `iqiyi.com`, `qiyi.com`, `zol.com`, and `pps.tv`. The script, which appeared to be the result of automatically bundling many related library modules together,

```
detectExecEnv: function() {
    var e = "";
    return
        (window._phantom
            || /* more PhantomJS probes */)
            && (e += "phantomjs"),
        window.Buffer && (e += "nodejs"),
        window.emit && (e += "couchjs"),
        window.spawn && (e += "rhino"),
        window.webdriver && (e += "selenium"),
        (window.domAutomation
            || window.domAutomationController)
            && (e += "chromium
                -based-automation-driver"), e
},
```

Listing 6: Artifact attribution in the wild

was minified but not obfuscated. It provides a rare example in which the attribution logic is fairly obvious: the presence of specific artifacts directly triggers what appears to be bot labeling via string concatenation. Note that this example uses `Window.Buffer`, one of our "possible" bot artifacts, which implies execution in the Node.js environment. Code locality strikes again: the code immediately adjacent to this excerpt includes functions that collect attributes of a containing `<iframe>` and and detect the activation of "private browsing."

Evasive Action. Listing 7 includes the core of an aggressively obfuscated script loaded exactly once, from http://www.school.kotar.co.il/. 73 other scripts in our collection share the same lexical hash. These were loaded on visits to 10 different Alexa domains, including https://www.payoneer.com/ and several `.il` domains. This script provides a clear example of artifact-based bot deflection. The obfuscation is distinctive, layering typical string garbling techniques behind a tangle of trivial functions performing simple operations like addition, comparison, or nothing at all. The deobfuscated script is trivial: if any of its artifact probes succeed, *it does nothing*; if they all fail, it reloads the current frame/page (presumably, with new content deemed too valuable for consumption by bots).

```
/* Original obfuscated code excerpt */
_ = window;
if (u82222.w(u82222.O(/* ... */))) {}
else location[u82222.f(u82222.r(11)+/* ... */]();
/* Deobfuscated version */
if (_["phantom"] || /* more PhantomJS probes */
  || _["Buffer"] || _["emit"] || _["spawn"]
  || _["webdriver"] || _["domAutomation"]
  ||  _["domAutomationController"]) {}
else location["reload"]();
```

Listing 7: Bot deflection in the wild

6 RELATED WORK

Security and Sandboxing. JStill [56] used static code signatures to detect known classes of JS obfuscation commonly employed by malware. Revolver [28] employed static lexical fingerprints and spatial clustering to detect and track the evolution of JS malware samples as their authors modified them to evade detection by the Wepawet browser honeypot/sandbox. Saxena et al. used off-line symbolic execution [46] to discover cross-site-scripting (XSS) vulnerabilities in JS code from web applications. The Rozzle [29] malware detection system employed a pragmatic form of symbolic execution to dramatically enhance the effectiveness of existing malware classifiers based on both dynamic and static analysis. Hulk [27] employed JS dynamic analysis techniques to elicit and detect malicious behavior of extensions for the Chrome browser.

Taint analysis of JS has been used to identify cross-site-scripting (XSS) vulnerabilities [31, 51] or leaks of private data to third parties [19, 53]. Taint analysis typically depends on substantial patches to a fixed (and soon obsolete) version of a browser; an exception [20] uses JS source rewriting to achieve inline flow monitoring without JS engine modifications, but the overhead is prohibitive.

Ambitious forensic browsing record and replay systems built via browser modification include WebCapsule [41] and JSgraph [33]. JSgraph in particular provides sophisticated causality tracking across related HTTP, DOM, and JS events (although it does not provide the breadth of API logging VV8 does). These systems provide impressive capabilities, but they quickly become obsolete as the upstream browser code bases rapidly evolve and the patches are left unmaintained (if made available at all).

Published JS sandboxing systems include both in-band systems like JSand [14] and Phung et al. [43] and out-of-band systems like ConScript [37]. Attempts [18, 52] to fully sandbox JS execution inside a JS engine implemented in JS, while technically sound, inevitably exhibit unacceptable performance.

Measurements. Richards' ironically titled survey [44] of real-world usage of JS's infamous `eval` feature provides an exhaustive catalog of uses and abuses and prompted at least one direct follow-up mitigation effort [26]. Nikiforakis's measurement [42] of remote JS script inclusions on top web sites, while not technically an analysis of JS code per se, clearly documented the distributed nature of JS web applications and many practical trust and security issues raised by that structure. Mayer and Mitchell produced the influential FourthParty web measurement framework and demonstrated the value of comprehensive web measurements while measuring third-party web tracking [35].

Acar et al. used in-browser instrumentation of select features to detect and measure browser fingerprinting with FPDetective [13]. Englehardt and Narayanan's survey of online trackers [22] served as a showcase for the mature and popular [21, 38] OpenWPM web privacy measurement platform built around Firefox. Like FourthParty before it, OpenWPM favors the flexibility of JS-based instrumentation over the in-browser approach taken by VV8. For the specific measurement goals of this paper, our in-browser approach provided coverage OpenWPM's in-band instrumentation could not match (Section 2). Merzdovnik et al. [36] measured the effectiveness of tracker blocking tools like *Ghostery* and *AdBlock Plus* while visiting over 100,000 sites within the Alexa top 200,000 domains. Their focus was on identifying sources of 3rd-party tracking and measuring the success or failure of blockers; ours is on fine-grained feature usage attribution and analysis on a script-by-script basis.

Lauinger et al. [30] surveyed 133,000 top sites and discovered widespread use of outdated or vulnerable JS libraries using a browser automation system like ours but without instrumenting or logging API usage. Snyder's measurement [49] of JS browser API usage on top web sites found approximately 50% of the available features completely unused of the Alexa top 10K at the time of measurement. A follow-up work [50] explored the degree to which potentially dangerous or undesirable JS browser API features could be disabled to reduce the browser's attack surface without disrupting the user's web browsing experience.

7 CONCLUSION

We have made the case for choosing out-of-band over in-band JS instrumentation when measuring the web for security and privacy concerns. We also presented **VisibleV8**, a custom variant of Chrome for measuring native JS/browser features used on the web. VV8 is modern, stealthy, and fast enough for both interactive use and web-scale automation. Our implementation is a small, highly maintainable patch easily ported to new browser versions. The resulting instrumentation, hidden inside the JS engine itself, is transparent to the visited pages, performs as well or better than in-band equivalents, and provides fine-grained feature tracking by source script and security origin. With VV8 we have observed JS code loaded directly or by frames on 29% of the Alexa top 50k sites actively testing for common automated browser frameworks. As many web measurements rely on such tools, this result marks a concerning development for security and privacy research on the web. **VisibleV8** has proven itself a transparent, efficient, and effective observation platform. We are publishing the code freely at https://github.com/jueckstock/visiblev8 in the hope it proves useful to the web research community.

REFERENCES

[1] [n. d.]. BrowserBench.org. https://browserbench.org/. ([n. d.]). Accessed: 2019-1-25.

[2] [n. d.]. Bug 1424176. https://bugzilla.mozilla.org/show_bug.cgi?id=1424176. ([n. d.]). Accessed: 2018-11-15.

[3] [n. d.]. Dromaeo. http://dromaeo.com/?recommended. ([n. d.]). Accessed: 2019-1-25.

[4] [n. d.]. Issue 793217. https://bugs.chromium.org/p/chromium/issues/detail?id=793217. ([n. d.]). Accessed: 2018-11-15.

[5] [n. d.]. javascript - Can a website detect when you are using selenium with chromedriver? https://stackoverflow.com/a/41220267. ([n. d.]). Accessed: 2018-11-15.

[6] [n. d.]. marmelab/gremlins.js: Monkey testing library for web apps and Node.js. https://github.com/marmelab/gremlins.js. ([n. d.]). Accessed: 2018-11-15.

[7] [n. d.]. PhantomJS - Scriptable Headless Browser. http://phantomjs.org/. ([n. d.]). Accessed: 2019-2-1.

[8] [n. d.]. Selenium - Web Browser Automation. https://docs.seleniumhq.org/. ([n. d.]). Accessed: 2019-2-1.

[9] [n. d.]. The RedMonk Programming Language Rankings: January 2019. https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/. ([n. d.]). Accessed: 2019-5-8.

[10] [n. d.]. The State of the Octoverse: top programming languages of 2018. https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/. ([n. d.]). Accessed: 2019-5-8.

[11] [n. d.]. Understanding web pages better. https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html. ([n. d.]). Accessed: 2019-8-19.

[12] [n. d.]. WebIDL Level 1. https://www.w3.org/TR/WebIDL-1/. ([n. d.]). Accessed: 2018-11-15.

[13] 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 ACM Conference on Computer and Communications Security (CCS)*.

[14] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM.

[15] James P Anderson. 1972. *Computer Security Technology Planning Study. Volume 2*. Technical Report. Anderson (James P) and Co Fort Washington PA.

[16] Marc Andreessen. 2011. Why Software is Eating the World. https://www.wsj.com/articles/SB10001424053111903480904576512250915629460. (2011). Accessed: 2018-04-20.

[17] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. 2017. Deterministic Browser. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[18] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. 2012. Virtual Browser: A Virtualized Browser to Sandbox Third-party JavaScripts with Enhanced Security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM.

[19] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3243734.3243823

[20] Andrey Chudnov and David A Naumann. 2015. Inlined information flow monitoring for javascript. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

[21] 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 ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3243734.3243860

[22] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM. https://doi.org/10.1145/2976749.2978313

[23] Úlfar Erlingsson. 2003. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical Report. Cornell University.

[24] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. 2016. Cloak of Visibility: Detecting When Machines Browse A Different Web. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[25] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2012. PUBCRAWL: Protecting Users and Businesses from CRAWLers. In *Proceedings of the USENIX Security Symposium*.

[26] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM. https://doi.org/10.1145/2338965.2336758

[27] Alexandros Kapravelos, Chris Grier, Neha Chachra, Chris Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the USENIX Security Symposium*. USENIX.

[28] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Chris Kruegel, and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the USENIX Security Symposium*.

[29] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE.

[30] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[31] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. 2017. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3133956.3134091

[32] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM. https://doi.org/10.1145/2508859.2516703

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

[34] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq. 2014. Hunting the Red Fox Online: Understanding and Detection of Mass Redirect-Script Injections. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[35] J. R. Mayer and J. C. Mitchell. 2012. Third-Party Web Tracking: Policy and Technology. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[36] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. 2017. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE.

[37] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[38] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. 2017. Dial One for Scam: A Large-Scale Analysis of Technical Support Scams. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[39] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. 2011. Fingerprinting information in JavaScript implementations. In *Proceedings of W2SP*.

[40] Martin Mulazzani, Philipp Reschl, Markus Huber, Manuel Leithner, Sebastian Schrittwieser, and Edgar Weippl. 2013. Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting. In *Proceedings of W2SP*.

[41] Christopher Neasbitt, Bo Li, Roberto Perdisci, Long Lu, Kapil Singh, and Kang Li. 2015. WebCapsule: Towards a Lightweight Forensic Engine for Web Browsers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM. https://doi.org/10.1145/2810103.2813656

[42] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. ou Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[43] Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight Self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS '09)*. ACM. https://doi.org/10.1145/1533057.1533067

[44] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The eval that men do. *ECOOP 2011–Object-Oriented Programming* (2011).

[45] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation*.

[46] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE.

[47] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.

[48] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. 2010. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[49] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser Feature Usage on the Modern Web. In *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM.

[50] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[51] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. 2015. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM. https://doi.org/10.1145/2810103.2813625

[52] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. 2012. JavaScript in JavaScript (js.js): Sandboxing Third-Party Scripts. In *Proceedings of the 3rd USENIX Conference on Web Application Development*. USENIX.

[53] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. 2012. Tracking the trackers: Fast and scalable dynamic analysis of web content for privacy violations. In *International Conference on Applied Cryptography and Network Security*. Springer.

[54] Antoine Vastel. [n. d.]. Fingerprint-Scanner. https://github.com/antoinevastel/fpscanner/. ([n. d.]). Accessed: 2019-2-1.

[55] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *Proceedings of the USENIX Security Symposium*.

[56] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2013. JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM.