SoftwareX 31 (2025) 102301

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Fakeium: A dynamic execution environment for JavaScript program analysis

José Miguel Moreno [a,*], Narseo Vallina-Rodriguez [b], Juan Tapiador [a]

[a] Universidad Carlos III de Madrid, Avda. de la Universidad 30, Leganés, Spain
[b] IMDEA Networks Institute, Avda. del Mar Mediterráneo 22, Leganés, Spain

ARTICLE INFO

Keywords:
Web security
Dynamic analysis
Sandbox
Obfuscation
Chromium
JavaScript

ABSTRACT

The JavaScript programming language, which began as a simple scripting language for the Web, has become ubiquitous, spanning desktop, mobile, and server applications. This increase in usage has made JavaScript an attractive target for nefarious actors, resulting in the proliferation of malicious browser extensions that steal user information and supply chain attacks that target the official Node.js package registry. To combat these threats, researchers have developed specialized tools and frameworks for analyzing the behavior of JavaScript programs to detect malicious patterns. Static analysis tools typically struggle with the highly dynamic nature of the language and fail to process obfuscated sources, while dynamic analysis pipelines take several minutes to run and require more resources per program, making them unfeasible for large-scale analyses. In this paper, we present Fakeium, a novel, open source, and lightweight execution environment designed under the twofold purpose of serving as a fast, fully automated tool for analyzing JavaScript at scale and as a flexible sandbox for manual investigation of complex sources. Built on top of the popular V8 engine, Fakeium complements traditional static analysis by providing additional API calls and string literals that would otherwise go unnoticed without the need for resource-intensive instrumented browsers or synthetic user input. Besides its negligible execution overhead, our tool is highly customizable and supports hooks for advanced analysis scenarios such as network traffic emulation. Fakeium's flexibility and ability to detect hidden API calls, especially in obfuscated sources, highlights its potential as a valuable tool for security analysts to detect malicious behavior.

Code metadata

Current code version	v0.0.7
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00357
Permanent link to Reproducible Capsule	https://github.com/josemmo/fakeium/tree/v0.0.7
Legal Code License	MIT License
Code versioning system used	Git (GitHub)
Software code languages, tools, and services used	TypeScript
Compilation requirements, operating environments & dependencies	Node.js 20 (LTS)
If available Link to developer documentation/manual	https://github.com/josemmo/fakeium/blob/main/README.md
Support email for questions	josemore@pa.uc3m.es

Software metadata

Current software version	v0.0.7
Permanent link to executables of this version	https://www.npmjs.com/package/fakeium/v/0.0.7
Permanent link to Reproducible Capsule	–
Legal Software License	MIT License
Computing platforms/Operating Systems	Linux, macOS, Microsoft Windows
Installation requirements & dependencies	Node.js 20 (LTS)
If available, link to user manual - if formally published include a reference to the publication in the reference list	https://www.npmjs.com/package/fakeium/v/0.0.7?activeTab=readme
Support email for questions	josemore@pa.uc3m.es

* Corresponding author.
E-mail addresses: josemore@pa.uc3m.es (José Miguel Moreno), narseo.vallina@imdea.org (Narseo Vallina-Rodriguez), jestevez@inf.uc3m.es (Juan Tapiador).

https://doi.org/10.1016/j.softx.2025.102301
Received 29 May 2025; Received in revised form 19 July 2025; Accepted 5 August 2025
Available online 21 August 2025
2352-7110/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Motivation and significance

Since its creation in late 1995, JavaScript has grown from a simple browser scripting language to a ubiquitous technology powering web, server, desktop, and mobile applications, dramatically expanding its attack surface. This increased adoption has led to a surge in malicious activities, such as browser extensions exfiltrating user data [1–4] and supply chain attacks targeting Node.js packages [5–7]. The development of effective JavaScript analysis tools is therefore critical for the research community. These tools are important not only for security and threat intelligence professionals, helping them find malware and vulnerable programs, but also in the field of privacy, aiding in the detection of information leaks and fingerprinting attempts.

As with other research fields, existing state-of-the-art approaches to JavaScript analysis can be grouped into two categories: static and dynamic. Static analysis examines source code without executing it, most commonly by lifting it to an Abstract Syntax Trees (AST). Although fast and reliable tooling backed by the web industry is already available [8], it has difficulties dealing with dynamically executed code (e.g., `eval`, `new Function()`) and obfuscation, both prevalent in malicious code [9,10]. Dynamic analysis executes code in an instrumented environment to observe its behavior, overcoming the limitations of static analysis [9]. However, traditional dynamic approaches typically require full browser instances [11–14], manual or synthetic user input to trigger specific code paths [15], and are computationally expensive and time-consuming, rendering them impractical for large-scale analyses of millions of sources.

To address this gap left by current JavaScript analysis paradigms, we present Fakeium, a novel, open source, and lightweight dynamic execution environment that serves a twofold purposes: automatically inferring the behavior of JavaScript programs at scale and enabling analysts to set up tailored sandboxes for the complex sources that require them. Built upon the V8 JavaScript engine [16], Fakeium complements static analysis by revealing API calls and string literals that emerge only at runtime, without the need for resource-heavy instrumented browsers or complex synthetic user input generation. This allows researchers and security analysts to perform deeper behavioral analyses at a scale previously unachievable with traditional dynamic analysis methods, contributing to scientific discovery in areas like malware detection and privacy auditing. Fakeium has already proven useful in analyzing the behavior of browser extensions of 900k+ extension versions from the Chrome Web Store [17].

Fakeium is distributed as an npm package, with users interacting with its sandbox programmatically. The typical workflow involves (*i*) instantiating the Fakeium sandbox, (*ii*) providing JavaScript code as a string or via a module resolver, (*iii*) customizing the execution environment with optional hooks (e.g., to define global variables or mock function behaviors), (*iv*) running the code, and (*v*) retrieving a detailed report of runtime events (e.g., invoked API calls). Listing 1 demonstrates this basic usage pattern.

1.1. Related work

The development of JavaScript analysis tools have been a key focus in web security, as surveyed by Sun and Ryu [18] and Andreasen et al. [9]. Static tools such as TAJS [19–21], Actarus [22], JSAI [23], Nodest [24], and Mininode [25] focus on type checking, taint analysis, or dependency reduction. Other tools like JStap [26], JSRevealer [27], and the work by Moog et al. [28] apply classification or obfuscation-resilient methods. Dynamic approaches use instrumentation or isolation techniques, such as Rozzle [29], TreeHouse [30], BreakApp [31], Hulk [11], J-Force [32], JSgraph [12], VisibleV8 [33], FV8 [34], and Arcanum [14], to uncover runtime behavior or enforce control policies [35]. Mixed techniques combine static and dynamic methods, including Zozzle [36], Mir [37] and the work of Sarker et al. [38]. They are used for malware detection, enforcing execution policies at runtime and measuring obfuscated sources on popular websites, respectively.

```
import { Fakeium } from "fakeium";

(async () => {
  // Setup instance
  const fakeium = new Fakeium({
    sourceType: "script",
    origin: "https://example.com/",
  });
  await fakeium.run("specifier.js", "alert(1)");

  // Print all logged events
  const events = fakeium.getReport().getAll();
  console.dir(events, { depth: null });

  // Free resources
  fakeium.dispose();
})();
```

Listing 1 Simple example program for executing code in the Fakeium sandbox.

Table 1 summarizes these tools. Compared to them, Fakeium has several advantages like support for extensive customization (e.g., hooks), can be installed via npm without requiring building large dependencies from source (i.e., Chromium), and provides comprehensive documentation through JSDoc comments and TypeScript types. Unlike other sandboxes, Fakeium attempts to find all behavioral signals (like in static analysis) rather than observing one particular execution.

2. Software description

Fakeium (short for *Fake Chromium*) is a lightweight, general-purpose sandbox designed for the dynamic execution of untrusted JavaScript code. It aims to offer a scalable alternative to traditional dynamic analysis when resource or time constraints make the latter impractical [9], thereby complementing static analysis by uncovering runtime-specific behaviors, especially in obfuscated sources. Fakeium prioritizes efficiency (low execution time, CPU and memory usage), usability (works out-of-the-box and is released as a npm package), extensibility (supports custom hooks), and correctness (features an extensive test suite to minimize regressions).

2.1. Software architecture

Fig. 1 shows an overview of the architecture used by Fakeium. Developers write a Node.js program (i.e., controller) that interacts with Fakeium's APIs to instrument the sandbox, execute code, and retrieve reports. To securely run untrusted code, Fakeium uses the isolated-vm library [39], which enables the creation of separate `v8::Isolate` instances. An *isolate* is a V8 engine instance with its own heap and objects [40]. This allows for the safe execution and disposal of untrusted code, as well as the enforcement of memory and time limits.

The actual Fakeium sandbox is a V8 isolate with a barebones execution context, exposing only standard ECMAScript built-in objects (e.g., `Error`, `JSON`, `eval`). It does not expose the Document Object Model (DOM) or direct network and file system APIs for safety. Fig. 2 shows a sequence diagram depicting the steps that Fakeium takes to execute a minimal example program. Before any user-provided code runs, Fakeium executes a bootstrapping script in the new isolate, hijacking the global scope and establishing mechanisms to report events outside the sandbox. This communication uses callback references that are only passed to the bootstrap script and kept inaccessible to subsequently executed untrusted code. Using isolates is arguably safer than

Table 1
Comparison of related work with our tool.

Took	Purpose	Target	License	Active	Linted	Tested	Cust.
TAJS [19]	Analyzer	Any	Apache	✗	✗	✓	✗
Actarus [22]	Find vulns.	Web	Proprietary	✓	–	–	–
Zozzle [36]	Find malware	Web	Proprietary	–	–	–	–
SandTrap [35]	Runtime	TAPs	BSD	✗	✗	✗	✗
Rozzle [29]	Find malware	Web	Proprietary	–	–	–	–
TreeHouse [30]	Runtime	Web	No license	✗	✗	✓	✗
Hulk [11]	Analyzer	Extensions	Proprietary	–	–	–	–
JSAI [23]	Analyzer	Any	No license	✗	✗	✗	✗
J-Force [32]	Runtime	Any	Proprietary	–	–	–	–
JSgraph [12]	Analyzer	Extensions	GPLv2	✗	✗	✗	✗
BreakApp [31]	Runtime	Node.js	GPLv2	✗	✗	✗	✗
Nodest [24]	Find vulns.	Node.js	Proprietary	–	–	–	–
VisibleV8 [33]	Analyzer	Web	BSD	✓	✗	✓	✗
JStap [26]	Analyzer	Any	AGPLv3	✗	✗	✗	✗
Mininode [25]	Runtime	Node.js	BSD	✗	✗	✓	✗
Mir [37]	Runtime	Node.js	MIT	✗	✗	✓	✗
JSRevealer [27]	Detect obfs.	Any	Proprietary	–	–	–	–
FV8 [34]	Analyzer	Any	MIT	✓	✗	✗	✗
Arcanum [14]	Analyzer	Extensions	MIT	✓	✗	✗	✗
Fakeium	Analyzer	Any	MIT	✓	✓	✓	✓

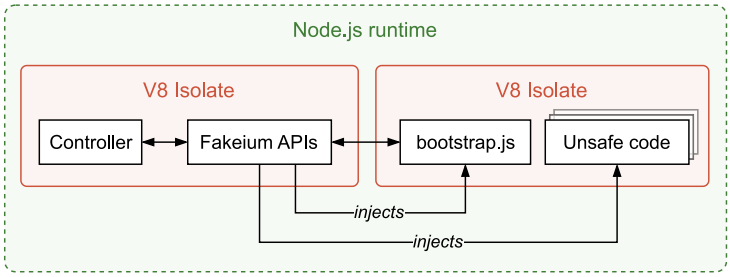


Fig. 1. Overview of Fakeium architecture.

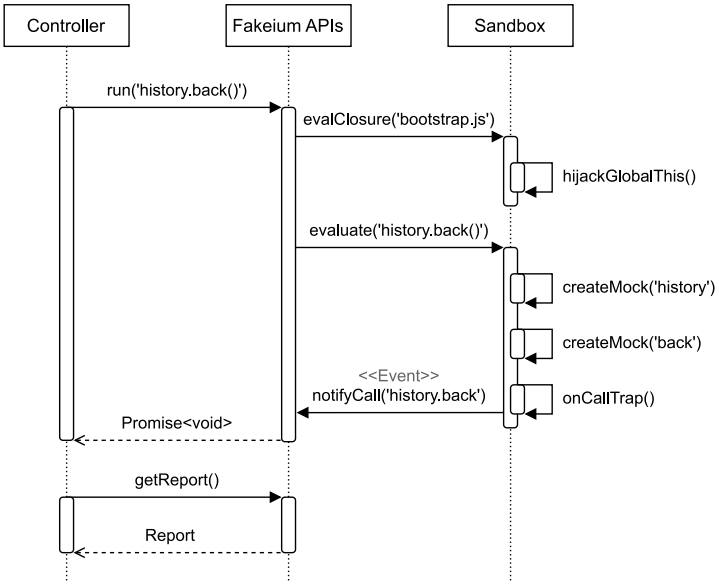


Fig. 2. Sequence diagram of an example program execution.

approaches relying on JavaScript-based membranes [41], forking new processes [30,31], or running Chromium with sandboxing disabled [14, 33,34]. Any missing API that the analyzed code expects is mocked at runtime using ECMAScript `Proxy` objects [42].

Fakeium is implemented in 1.8k lines of code, excluding comments and whitespace, linted with ESLint [43] and validated by 40+ unit tests. It is built using pkgroll [44], resulting in a final unpacked size of 50 kiB.

2.2. Software functionalities

Mocks. A key feature of Fakeium is its ability to dynamically and recursively mock (create dummies of) objects and properties missing from the global scope as they are accessed by the analyzed code (e.g., `chrome.cookies.getAll`). This approach eliminates the need to implement all existing JavaScript APIs or anticipate what variables a

```
import { Reference } from "fakeium";

// 1. Serializable value hook
// The value (second argument) is deep-copied into the
// sandbox.
fakeium.hook("document", {
  nodeType: 9, // Node.DOCUMENT_NODE
  readyState: "complete"
});
fakeium.hook("require", undefined);

// 2. Function hook
// The sandbox exposes a binding that, when invoked,
// executes the provided function outside the sandbox
// and returns a deep-copy of the result.
fakeium.hook("bridge", () => {
  console.log("Hi from outside the sandbox!");
  return { success: true }; // Copied to the sandbox
});

// 3. Reference hook
// In this example, it redirects accesses of the "chrome"
// object to "browser", performed inside the sandbox.
fakeium.hook("chrome", new Reference("browser"));
```

Listing 2 Examples of hooks supported by Fakeium.

program will require. Mocks ensure deterministic execution as they do not rely on external resources (except for `Math.random`, which can still be hooked). This design runs orders of magnitude faster than traditional dynamic sandboxes, trading some accuracy for speed and scalability.

When the bootstrap script runs in an isolate, it wraps the global scope in a `Proxy` [42], defining handlers for the `get`, `set`, `has`, `construct` and `apply` traps to log events and create new mocks of undefined properties on-the-fly. Native objects like `JSON` are wrapped in a mock on first access, proxying calls to the original object using the `Reflect` object [45]. Undefined properties are filled with *full mocks*, which are as generic as possible since the expected value is unknown. These mocks have an infinite number of properties, are instantiable, iterable, invocable, and thenable.

To maximize code path exploration, mocks execute function arguments that are discovered when an `apply` trap is triggered, like the `cb` variable in `window.addEventListener("load", cb)`. Callbacks are scheduled for invocation immediately after they are discovered. We use tainting through non-enumerable `Symbol` properties to prevent re-entrancy. If a mock encounters a function that is already tainted, it will not execute it again. The main purpose of this forced execution approach is not to guarantee a perfect traversal of all code paths, but rather to maximize runtime signals by interacting with as much of the analyzed code as possible.

Hooks. When default mocking is insufficient (e.g., a program expects specific values or network payloads), Fakeium's hook system offers extensive sandbox customization. Developers can use hooks to modify variables from the sandbox's global scope, as shown in Listing 2. Depending on the type of hooked value, they have slightly different behaviors. Serializable values like numbers, strings or simple objects are deep-copied into the sandbox using the structured clone algorithm [46]. Functions are exposed as callable bindings within the sandbox and executed *outside* of it. They can accept arguments from the sandbox and return values to it (even promises), provided these are serializable. Lastly, references redirect `get`, `set` and `apply` traps to other variables inside the sandbox. Fakeium uses these by default to alias objects like `window` to `globalThis`. More examples on how to write hooks can be found on Fakeium's project page and in Section 3.

Module resolution. Fakeium supports ES modules via a user-defined module resolver callback. This function receives a fully resolved URL instance [47] and should return a `Buffer` containing the module's source code, or `null` if the module is not found. The default origin is *"file:///"*, but it can be set to any other compliant value such as *"https://example.com"* or *"chrome-extension://"*.

Event logging. Fakeium logs variable reads, writes, and function calls that occur during execution. These events are captured by the `Proxy` traps in mocks and immediately transmitted to the main isolate, ensuring that they are recorded even if the program crashes or throws an uncaught error. Communication with the main isolate is conducted through a callback that is passed as an argument to the bootstrap closure when the isolate is initialized. This ensures that the analyzed source cannot obtain a reference this object. Each event includes the source location where it originated (filename, line, column) obtained using the `Error.captureStackTrace` internal V8 function.

Object tainting. Internally, Fakeium uses tainting with `Symbol`-keyed properties to track object metadata. These include marking mock objects to avoid mocking them twice, assigning unique IDs for logging, tracking visited callbacks to prevent infinite recursion and re-entrancy, and identifying stub functions from hooks. These properties are hidden from the analyzed program via `Proxy` traps to avoid altering its behavior.

3. Illustrative examples

To demonstrate the practical application and customization of Fakeium, we analyze a real-world malware sample from an August 2024 campaign targeting npm users [7]. The malware mimicked "noblox.js", a popular Roblox API library, having its malicious payload in the "postinstall.js" script, which we obtained from Socket [48]. The script is heavily obfuscated and crashes when trying to call `require()`, which is `undefined` by default in Fakeium. Static analysis of this obfuscated script yields no useful signals regarding its behavior.

To overcome the immediate crash and analyze the script, we use the hooking capabilities of our tool to redefine `globalThis.require`. This allows us to provide controlled, fake implementations for the modules the malware attempts to load, such as "child_process", "crypto", "fs", "node-fetch", "os", and "path". These *ad hoc* mocks prevent the sandbox from crashing while still isolating the malware from actual system resources. For instance, we return fixed, tainted values for calls like `os.userInfo().username` and `crypto.randomBytes()` to trace their usage and implement minimal functionality for the `Buffer` class, which the malware also expects. Listing 3 shows a simplified version of these hooks.

With these relatively minor customizations, Fakeium successfully runs the obfuscated sample and reveals its behavior. The analysis report shows several stages:

1. The "postinstall.js" script performs deobfuscation routines, involving multiple calls to `parseInt` and `setInterval` to unpack the main malicious payload.
2. It uses the mocked "node-fetch" module to download an executable from a GitHub repository[1] and save it to disk with a random filename.
3. It adds several keys to the Windows Registry using `execSync()` to bypass User Account Control (UAC) and execute the downloaded binary with elevated privileges (Listing 4).
4. The malware sends a message to a Discord channel with the content *"someone got trolled, Uac Bypassed, Quasar Ran"*, presumably to notify the operators of a successful compromise.

This example demonstrates Fakeium's extensive customization options and ease of use when setting up custom environments, giving analysts fine-grained control to analyze sophisticated samples that require tailored responses to uncover hidden logic. It also shows its

[1] Hosted at https://github.com/aspdasdksa2/callback/raw/main/cmd.exe.

```
fakeium.hook("require", specifier => {
  if (specifier === "crypto") {
    return { randomBytes: () => "fixedValue" };
  }
  if (specifier === "os") {
    return {
      userInfo: () => {
        return { username: "testUsername" };
      }
    };
  }
  if (specifier === "child_process") {
    return {
      execSync: cmd => {
        console.log("execSync()", cmd);
      }
    };
  }
  // [...]
});

fakeium.hook("Buffer.from", (value, encoding) => {
  return {
    toString: () => {
      return Buffer.from(value, encoding).toString();
    }
  };
});
```

Listing 3 Hooks used to analyze the malware example (simplified).

```
reg add "HKCU\Software\Classes\
    AppX82a6gwre4fdg3bt635tn5ctqjf8msdd2\Shell\open\
    command" /f
reg add "HKCU\Software\Classes\
    AppX82a6gwre4fdg3bt635tn5ctqjf8msdd2\Shell\open\
    command" /ve /t REG_SZ /d "C:\WindowsApi\fixedValue.
    exe" /f
reg add "HKCU\Software\Classes\ms-settings\Shell\Open\
    command" /f
# [...]
reg add "HKCU\Software\Classes\ms-settings\Shell\Open\
    command" /v DelegateExecute /t REG_SZ /d "" /f
start "" "C:\Windows\System32\fodhelper.exe"
```

Listing 4 UAC bypass via FodHelper.exe commands (simplified).

ability to dissect recent, heavily obfuscated malware samples and reveal complex, multi-stage behaviors that would be hidden from static analysis alone.

4. Impact

Fakeium greatly improves the analysis of JavaScript in limited-resource environments, particularly for obfuscated code, by complementing static analysis with signals that can only be observed at runtime. Its strength lies in its ability to analyze behaviors at scale without the need for complex elicitation techniques.

For academic researchers, this opens the door to large-scale measurement studies that were previously impractical due to the high overhead of traditional dynamic analysis. Furthermore, it serves as a rapid, cost and time-effective triage tool for industry production environments. For instance, it can be integrated into threat intelligence platforms to detect malware, or into CI/CD pipelines to catch unexpected behaviors at runtime before deployment. By providing a quick initial assessment, our tool reduces computing costs and time by identifying sources that require more intensive analysis.

Fakeium is designed for broad adoption, distributed under the MIT License and available as an npm package. Its customizability, ease of integration, and low resource requirements make it ideal for both academic research and industry.

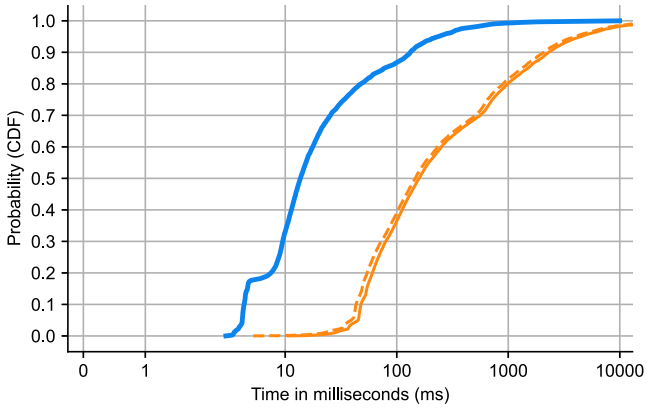


Fig. 3. Wall time distributions for static and dynamic analysis pipelines. Solid orange line for static analysis (whole duration), dashed orange line for static analysis (only AST parsing), **bold solid blue** line for Fakeium.

4.1. Evaluation

To support our claims, we evaluate Fakeium on various datasets of JavaScript programs. Although Fakeium uses dynamic execution, its output is similar to that of static analysis in that it aims to reveal all possible signals in the code, albeit obtained at runtime. Therefore, rather than comparing it to traditional dynamic analysis, which typically provides a trace of a single code path and would not serve as a suitable baseline, we compare it to static analysis to evaluate its relative overhead and the additional signals it uncovers.

Datasets and methodology. We crawled two datasets representative of the current web landscape in July 2024: the top 1,000 most popular Chrome browser extensions from the Chrome Web Store [49], and the latest version of all 4,135 libraries served by Cloudflare's cdnjs [50]. We also included several versions of a custom script A obfuscated with varying intensity using javascript-obfuscator [51].

For the static analysis pipeline, we use esbuild [52] to bundle entrypoints identified in the manifest files [53] and HTML documents of extensions, followed by AST generation and traversal with Babel [8] to extract API calls and string literals. For dynamic analysis, the sources were run directly in Fakeium with a 10-second execution timeout and a 256 MiB heap limit.

After discarding sources that caused esbuild to throw an error, we end up with 941 extensions (comprising 6,056 entrypoints) and 4,006 cdnjs libraries. We note that Fakeium crashed due to uncaught errors in 38% of extension entrypoints and 17% of libraries. This is expected due to the mocking nature of the tool and does not prevent it from still generating a report. We consider this behavior an acceptable trade-off to achieve completeness and speed in signal discovery.

Performance. Fakeium exhibits remarkable efficiency, making it suitable for large-scale analysis. On average, dynamic analysis took just 70 ms per source, with an average heap usage of 1.5 MiB. This is orders of magnitude faster and uses far less memory than the AST parsing phase of the static analysis pipeline, which averaged 1,201 ms and 67.5 MiB heap per source. Figs. 3 and 4 show the results of our time and memory usage measurements, respectively. This minimal overhead allows for the integration of Fakeium alongside static tools and is consistent with our experience using it with considerably larger datasets [17].

Signals. Fakeium consistently uncovers critical signals that static analysis misses. It identified hidden API calls (detected by Fakeium but

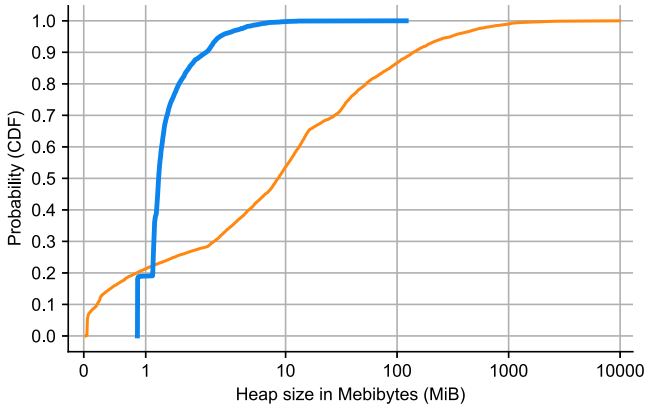


Fig. 4. Heap memory size distributions for static and dynamic analysis pipelines. Orange line for static analysis, **bold blue** line for Fakeium.

Table 2
Comparison of the number of API calls and strings detected in obfuscated sources.

Level	Static		Fakeium	
	Calls	Strings	Calls	Strings
None	10/10	10/10	10/10	10/10
Default	10/10	10/10	10/10	10/10
Low	10/10	10/10	10/10	10/10
Medium	5/10	2/10	10/10	10/10
High	2/10	0/10	10/10	10/10
Custom	2/10	0/10	10/10	10/10

absent in static analysis) in 57% of sources with functions calls, with an average of 4 ± 10 such calls per source. Dynamic analysis often reveals security and privacy-relevant strings. Fakeium successfully extracted 205,919 unique hidden literals from 41% of the dataset sources that were not detected by the static analysis pipeline. These included relevant items such as 18 hardcoded IPv4 ranges, 32 UUIDs, 339 URLs (including 175 distinct origins and 84 Google Analytics measurement IDs), and 422 JSON documents.

Obfuscation resistance. Fakeium is highly effective against common JavaScript obfuscation techniques. As shown in Table 2, Fakeium consistently identified all ten API calls and string literals in our obfuscated scripts. In contrast, the performance of static analysis degraded significantly with increased obfuscation, failing to find any in the high and custom presets.

4.2. Limitations and future work

Despite not experiencing this during our evaluation, we acknowledge that some programs may exploit Fakeium's mocking nature to detect that they are running in a sandbox. While we recognize that no sandbox is entirely immune to detection, we propose two potential mitigations to evasion techniques. First, an attempt by a program to detect its environment (e.g., by using unusual variable accesses) is a strong indicator of suspicious behavior. Therefore, we can use Fakeium to detect these signals and flag the program for further analysis. Second, we propose developing better mocks by integrating tools like jsdom [54] to more accurately simulate a native browser, making detection more difficult. This strategy should also allow us to reduce the number of crashes that we identified during our evaluation. By providing a more complete DOM environment, we expect fewer uncaught errors caused by API functionalities that are missing from our default generic mocks.

5. Conclusions

Dynamic analysis of JavaScript applications at scale is a significant challenge due to the inherent time and computational costs, specially when dealing with obfuscation or highly dynamic code. In this paper, we presented Fakeium, a lightweight sandbox based on the V8 engine for the efficient execution of untrusted JavaScript code. Fakeium is a practical, scalable alternative to traditional dynamic analysis when the latter is unfeasible due to resource constraints.

Our experiments demonstrate that Fakeium has a minimal overhead, averaging just 70 ms and 1.5 MiB of heap per run. This efficiency makes it an ideal fit to integrate into existing static analysis pipelines, providing dynamic insights that supplement their findings without significant overhead. Furthermore, its high degree of customizability through hooks allows developers to create tailored environments for complex instrumentation and analysis scenarios. Its dynamic nature makes it resilient against obfuscation, enabling it to extract meaningful behavioral signals from sources that would otherwise be opaque to static methods. Overall, Fakeium is a valuable and accessible tool for advancing the study of web security and privacy.

CRediT authorship contribution statement

José Miguel Moreno: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Conceptualization. **Narseo Vallina-Rodriguez:** Writing – review & editing. **Juan Tapiador:** Writing – review & editing, Supervision, Resources, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Juan Tapiador reports financial support was provided by Agencia Estatal de Investigación (Spain). Jose Miguel Moreno reports financial support was provided by Spanish Ministry of Science and Innovation. Narseo Vallina-Rodriguez reports financial support was provided by Agencia Estatal de Investigación (Spain). Narseo Vallina-Rodriguez reports financial support was provided by ESF, Investing in Your Future. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by the Spanish AEI grant CYCAD (PID2022-140126OB-I00). José Miguel Moreno is funded by the Spanish Ministry of Science and Innovation with a FPI Predoctoral Grant (PRE2020-094224). Narseo Vallina-Rodriguez has been appointed as 2019 Ramon *y* Cajal fellow (RYC2020-030316-I) funded by MICIU/AEI/10.13039/501100011033 and the ESF Investing in your future.

Appendix A. Base obfuscation script

Listing 5 provides the source code of the script used as a base to generate the obfuscated sources dataset.

Data availability

A link to the GitHub repository containing the source code of Fakeium is included in the manuscript. The datasets used for evaluation of the tool are available upon reasonable request.

```
// Regular function calls
firstFn("ARG_1");
secondFn("ARG_2");

// Async function calls
(async () => {
  thirdFn("ARG_3");
  await fourthFn("ARG_4");
  fifthFn("ARG_5");
})();

// Callbacks
window.addEventListener("load", () => {
  sixthFn("ARG_6");
});
setTimeout(() => {
  seventhFn("ARG_7");
  eighthFn("ARG_8");
}, 1000);

// More complex logic
(() => {
  for (let i=0; i<Number.MAX_VALUE; i++) {
    if (i === 20) {
      ninthFn("ARG_9");
      break;
    }
  }
  tenthFn("ARG_10");
})();
```

Listing 5 Base script used for obfuscation experiments.

References

[1] Palant W. More malicious extensions in chrome web store. 2023, URL https://palant.info/2023/05/31/more-malicious-extensions-in-chrome-web-store/.

[2] Titterington A. Dangerous browser extensions. 2023, URL https://www.kaspersky.com/blog/dangerous-browser-extensions-2023/50059/.

[3] ReasonLabs Research Team. The cashback extension killer. 2023, URL https://reasonlabs.com/research/the-cashback-extension-killer.

[4] Toulas B. Malware force-installs chrome extensions on 300,000 browsers, patches DLLs. 2024, URL https://www.bleepingcomputer.com/news/security/malware-force-installs-chrome-extensions-on-300-000-browsers-patches-dlls/.

[5] Unit 42. Hacking employers and seeking employment: Two job-related campaigns bear hallmarks of north Korean threat actors. 2023, URL https://unit42.paloaltonetworks.com/two-campaigns-by-north-korea-bad-actors-target-job-hunters/.

[6] Check Point Research Team. Check point CloudGuard spectral detects malicious crypto-mining packages on NPM. 2024, URL https://blog.checkpoint.com/2023/02/14/check-point-cloudguard-spectral-detects-malicious-crypto-mining-packages-on-npm-the-leading-registry-for-javascript-open-source-packages/.

[7] Gelb Y. Year-long campaign of malicious npm packages targeting roblox users. 2024, URL https://checkmarx.com/blog/year-long-campaign-of-malicious-npm-packages-targeting-roblox-users/.

[8] Ng B, Zhu H, Jùnliàng H, Smyth L, Ribaudo N, Sauleau S, et al. Babel. 2024, URL https://babel.dev/.

[9] Andreasen E, Gong L, Møller A, Pradel M, Selakovic M, Sen K, et al. A survey of dynamic analysis and test generation for JavaScript. ACM Comput Surv 2017;50(5). http://dx.doi.org/10.1145/3106739.

[10] Rauschmayer A. Exploring JavaScript. ES2024 ed.. 2024, Ch. 28. URL https://exploringjs.com/.

[11] Kapravelos A, Grier C, Chachra N, Kruegel C, Vigna G, Paxson V. Hulk: Eliciting malicious behavior in browser extensions. In: Proceedings of the 23rd USENIX conference on security symposium. USENIX Association; 2014, p. 641–54, URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kapravelos.

[12] Li B, Vadrevu P, Lee KH, Perdisci R, Liu J, Rahbarinia B, et al. JSgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser JavaScript executions. In: Proceedings of the 2018 network and distributed system security (NDSS) symposium, vol. 122, 2018, p. 132. http://dx.doi.org/10.14722/ndss.2018.23319.

[13] Yang Z, Allen J, Landen M, Perdisci R, Lee W. TRIDENT: Towards detecting and mitigating web-based social engineering attacks. In: proceedings of the 32nd USENIX security symposium (USENIX security 23). USENIX Association; 2023, p. 6701–18, URL https://www.usenix.org/conference/usenixsecurity23/presentation/yang-zheng.

[14] Xie Q, Murali MVK, Pearce P, Li F. Arcanum: Detecting and evaluating the privacy risks of browser extensions on web pages and web content. In: Proceedings of the 33rd USENIX security symposium. USENIX Association; 2024, p. 4607–24, URL https://www.usenix.org/conference/usenixsecurity24/presentation/xie-qinge.

[15] Android developers. UI/Application exerciser monkey. 2023, URL https://developer.android.com/studio/test/other-testing-tools/monkey.

[16] The Chromium Authors. V8 JavaScript engine. 2024, URL https://v8.dev.

[17] Moreno JM, Vallina-Rodriguez N, Tapiador J. Did I vet you before? Assessing the chrome web store vetting process through browser extension similarity. 2024, http://dx.doi.org/10.48550/arXiv.2406.00374, ArXiv Preprint ArXiv.2406.00374.

[18] Sun K, Ryu S. Analysis of JavaScript programs: Challenges and research trends. ACM Comput Surv 2017;50(4). http://dx.doi.org/10.1145/3106741.

[19] Jensen SH, Møller A, Thiemann P. Type analysis for JavaScript. In: International static analysis symposium. Springer; 2009, p. 238–55. http://dx.doi.org/10.1007/978-3-642-03237-0_17.

[20] Jensen SH, Møller A, Thiemann P. Interprocedural analysis with lazy propagation. In: International static analysis symposium. Springer; 2010, p. 320–39. http://dx.doi.org/10.1007/978-3-642-15769-1_20.

[21] Jensen SH, Madsen M, Møller A. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. Association for Computing Machinery; 2011, p. 59–69. http://dx.doi.org/10.1145/2025113.2025125.

[22] Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R. Saving the world wide web from vulnerable JavaScript. In: Proceedings of the 2011 international symposium on software testing and analysis. Association for Computing Machinery; 2011, p. 177–87. http://dx.doi.org/10.1145/2001420.2001442.

[23] Kashyap V, Dewey K, Kuefner EA, Wagner J, Gibbons K, Sarracino J, et al. JSAI: a static analysis platform for JavaScript. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. Association for Computing Machinery; 2014, p. 121–32. http://dx.doi.org/10.1145/2635868.2635904.

[24] Nielsen BB, Hassanshahi B, Gauthier F. Nodest: feedback-driven static analysis of node.js applications. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. Association for Computing Machinery; 2019, p. 455–65. http://dx.doi.org/10.1145/3338906.3338933.

[25] Koishybayev I, Kapravelos A. Mininode: Reducing the attack surface of node.js applications. In: Proceedings of the 23rd international symposium on research in attacks, intrusions and defenses. USENIX Association; 2020, p. 121–34, URL https://www.usenix.org/conference/raid2020/presentation/koishybayev.

[26] Fass A, Backes M, Stock B. JStap: a static pre-filter for malicious JavaScript detection. In: Proceedings of the 35th annual computer security applications conference. Association for Computing Machinery; 2019, p. 257–69. http://dx.doi.org/10.1145/3359789.3359813.

[27] Ren K, Qiang W, Wu Y, Zhou Y, Zou D, Jin H. JSRevealer: A robust malicious JavaScript detector against obfuscation. In: Proceedings of the 53rd annual IEEE/iFIP international conference on dependable systems and networks. 2023, p. 339–51. http://dx.doi.org/10.1109/DSN58367.2023.00041.

[28] Moog M, Demmel M, Backes M, Fass A. Statically detecting JavaScript obfuscation and minification techniques in the wild. In: Proceedings of the 51st annual IEEE/iFIP international conference on dependable systems and networks. 2021, p. 569–80. http://dx.doi.org/10.1109/DSN48987.2021.00065.

[29] Kolbitsch C, Livshits B, Zorn B, Seifert C. Rozzle: De-cloaking internet malware. In: Proceedings of the 2012 IEEE symposium on security and privacy. 2012, p. 443–57. http://dx.doi.org/10.1109/SP.2012.48.

[30] Ingram L, Walfish M. Treehouse: Javascript sandboxes to help web developers help themselves. In: Proceedings of the 2012 USENIX annual technical conference. USENIX Association; 2012, p. 153–64, , URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/ingram.

[31] Vasilakis N, Karel B, Roessler N, Dautenhahn N, DeHon A, Smith JM. BreakApp: Automated, flexible application compartmentalization. In: Proceedings of the 2018 network and distributed system security (NDSS) symposium. 2018, http://dx.doi.org/10.14722/ndss.2018.23131.

[32] Kim K, Kim IL, Kim CH, Kwon Y, Zheng Y, Zhang X, et al. J-Force: Forced execution on JavaScript. In: Proceedings of the 26th international conference on world wide web. International World Wide Web Conferences Steering Committee; 2017, p. 897–906. http://dx.doi.org/10.1145/3038912.3052674.

[33] Jueckstock J, Kapravelos A. VisibleV8: In-browser monitoring of JavaScript in the wild. In: Proceedings of the 2019 internet measurement conference. Association for Computing Machinery; 2019, p. 393–405. http://dx.doi.org/10.1145/3355369.3355599.

[34] Pantelaios N, Kapravelos A. FV8: A forced execution JavaScript engine for detecting evasive techniques. In: Proceedings of the 33rd international symposium on research in attacks, intrusions and defenses. USENIX Association; 2024, p. 3747–64, URL https://www.usenix.org/conference/usenixsecurity24/presentation/pantelaios.

[35] Ahmadpanah MM, Hedin D, Balliu M, Olsson LE, Sabelfeld A. SandTrap: Securing JavaScript-driven trigger-action platforms. In: Proceedings of the 30th USENIX security symposium. 2021, p. 2899–916, URL https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadpanah.

[36] Curtsinger C, Livshits B, Zorn B, Seifert C. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In: Proceedings of the 20th USENIX security symposium. USENIX Association; 2011, URL https://www.usenix.org/conference/usenix-security-11/zozzle-fast-and-precise-browser-javascript-malware-detection.

[37] Vasilakis N, Staicu C-A, Ntousakis G, Kallas K, Karel B, DeHon A, et al. Preventing dynamic library compromise on node.js via RWX-based privilege reduction. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. Association for Computing Machinery; 2021, p. 1821–38. http://dx.doi.org/10.1145/3460120.3484535.

[38] Sarker S, Jueckstock J, Kapravelos A. Hiding in plain site: Detecting JavaScript obfuscation through concealed browser API usage. In: Proceedings of the 2020 ACM internet measurement conference. Association for Computing Machinery; 2020, p. 648–61. http://dx.doi.org/10.1145/3419394.3423616.

[39] Laverdet M, Open-source contributors. isolated-vm – secure & isolated JS environments for nodejs. 2024, URL https://github.com/laverdet/isolated-vm.

[40] Gruber J. Embedded builtins – V8. 2018, URL https://v8.dev/blog/embedded-builtins.

[41] Cutsem TV. Membranes in JavaScript. 2012, URL https://tvcutsem.github.io/js-membranes.

[42] MDN contributors. Proxy. 2024, URL https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

[43] Zakas NC, Djermanovic M, Open-source contributors. ESLint – find and fix problems in your JavaScript code. 2024, URL https://eslint.org/.

[44] Osame H, Open-source contributors. pkgroll – Zero-config package bundler for Node.js + TypeScript. 2024, URL https://github.com/privatenumber/pkgroll.

[45] MDN contributors. Reflect. 2024, URL https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Reflect.

[46] MDN contributors. The structured clone algorithm. 2024, URL https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Structured_clone_algorithm.

[47] MDN contributors. URL. 2024, URL https://developer.mozilla.org/docs/Web/API/URL.

[48] Socket Inc. noblox.js-async / postinstall.js. 2024, URL https://socket.dev/npm/package/noblox.js-async/files/4.6.9/postinstall.js.

[49] Google Chrome Developers. Chrome web store. 2024, URL https://chromewebstore.google.com/.

[50] Kirkman R, Davis T, Cowley M, Open-source contributors. cdnjs – the #1 free and open source CDN built to make life easier for developers. 2024, URL https://cdnjs.com/.

[51] Kachalov T, Open-source contributors. JavaScript obfuscator tool. 2024, URL https://obfuscator.io/.

[52] Wallace E, Open-source contributors. esbuild – An extremely fast bundler for the web. 2024, URL https://esbuild.github.io/.

[53] Google Chrome Developers. Manifest file format. 2012, URL https://developer.chrome.com/docs/extensions/reference/manifest.

[54] Denicola D, Open-source contributors. jsdom – A JavaScript implementation of various web standards, for use with Node.js. 2025, URL https://github.com/jsdom/jsdom.