Dynamic Security Analysis of JavaScript: Are We There Yet?

Stefano Calzavara
Università Ca' Foscari Venezia
Venice, Italy
stefano.calzavara@unive.it

Samuele Casarin
Università Ca' Foscari Venezia
Venice, Italy
Scuola IMT Alti Studi Lucca
Lucca, Italy
samuele.casarin@unive.it

Riccardo Focardi
Università Ca' Foscari Venezia
Venice, Italy
focardi@unive.it

Abstract

In this paper, we systematically evaluate the effectiveness of existing tools for the dynamic security analysis of client-side JavaScript, focusing in particular on information flow control. Each tool is evaluated in terms of: (*i*) *compatibility*, i.e., the ability to process and analyze existing scripts without breaking; (*ii*) *transparency*, i.e., the ability to preserve the original script semantics when security enforcement is not necessary; (*iii*) *coverage*, i.e., the effectiveness in terms of number of detected information flows; (*iv*) *performance*, i.e., the computational overhead introduced by the analysis. Our investigation shows that most of the existing analysis tools are incompatible with the modern Web and the compatibility issues affecting them are not easily fixed. Moreover, transparency issues abound and make us question analysis correctness. This is also confirmed by our coverage evaluation, showing that some tools are unable to detect any information flow on real-world websites, while the remaining tools report significantly different outputs. Finally, we observe that the computational overhead of analysis tools may be significant and can exceed 30x. In the end, out of all the evaluated tools, just one of them (Project Foxhound) is effective enough for practical adoption at scale.

CCS Concepts

• **Security and privacy → Web application security**; *Software security engineering*.

Keywords

JavaScript, Information flow control, Web measurements

ACM Reference Format:
Stefano Calzavara, Samuele Casarin, and Riccardo Focardi. 2025. Dynamic Security Analysis of JavaScript: Are We There Yet?. In *Proceedings of the ACM Web Conference 2025 (WWW '25), April 28-May 2, 2025, Sydney, NSW, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3696410.3714614

1 Introduction

JavaScript is the most popular programming language in the world, according to recent data from GitHub [25] and Stack Overflow [47]. Although originally designed as a simple scripting language to carry out easy tasks and make web pages more enjoyable to navigate, it evolved into a production-level programming language that is now used to develop security-sensitive applications, thus motivating a significant interest by the research community. The community proposed different approaches to analyze JavaScript: static analysis [28, 31, 33, 40], dynamic analysis [12] and hybrid solutions [44, 51]. Yet, to the best of our knowledge, we still lack standard and widely accepted analysis tools that can be used to perform realistic security assessments of the JavaScript code available on the modern Web. For example, Steffens et al. implemented their own taint tracking tool for JavaScript as part of a study on client-side XSS [48], while Ahmad et al. developed a new taint tracking engine to investigate information flows involving the web storage [11]. Even industrial vendors like Samsung and SAP are developing new tools for JavaScript analysis [30, 32]. This picture suggests that existing efforts are difficult to reuse or suffer from important issues, but what is actually lacking remains unclear. The goal of this paper is shedding light on the current state of the art to understand key strengths and weaknesses of existing tools.

Contributions. In this work, we systematically evaluate the effectiveness of existing tools for information flow control of JavaScript programs. Information flow control is a powerful and general technique for security assessments, being able to uniformly capture both confidentiality and integrity policies [42]. We focus on dynamic and hybrid approaches, because they are better suited to deal with the high amount of dynamic features of JavaScript, such as string-to-code transformation functions [13]. Moreover, we only focus on the traditional use case of JavaScript as a client-side scripting language running in web browsers, which received a significant deal of attention from the web security community, see, e.g., research on web tracking [10] and XSS detection [48].

After a preliminary investigation of the state of the art to identify a set of candidate analysis tools to evaluate (Section 2), we assess them on popular live websites to measure:

- Their *compatibility* with the modern Web, i.e., their ability to process and analyze existing scripts from real-world websites without breaking (Section 3).
- Their *transparency* guarantees, i.e., their ability to preserve the execution behavior of the original scripts when they do not violate an intended information flow policy (Section 4).
- Their *coverage*, i.e., their effectiveness in terms of number of detected information flows (Section 5).
- Their *performance* in terms of running times, which determines their practicality and amenability for large-scale web security measurements (Section 6).

Our analysis shows that, despite significant efforts from the research community, most analysis tools are unavailable, difficult to run or do not support modern browser automation frameworks. Even for those tools which we are able to run, the results are largely unsatisfactory, showing that just a single tool (Project Foxhound [32]) is effective enough for practical adoption. To support reproducibility, we share all our code and data [9, 17].

2 Methodology

We here discuss the methodology used in our evaluation of publicly available dynamic analysis tools for JavaScript.

2.1 Tool Selection

To make our investigation systematic, we started our tool selection from a survey of dynamic analysis techniques for JavaScript by Andreasen et al. [12] and we used Google Scholar to identify more recent research that references any of the papers from the survey. To better define the scope of our evaluation and avoid improper comparisons, we restricted the focus to information flow analyzers rather than generic dynamic analysis tools such as Jalangi [45].

This process led to a preliminary list of 18 tools to evaluate, which we further reduced based on two additional criteria required by our experimental study. First, we require *tool availability*, i.e., we only focus on analysis tools which are publicly available according to the research papers where they have been presented. To obtain a tool, we first look at the location of the tool's binary or source code reported in the paper. If this attempt is unsuccessful, e.g., due to broken links, we request the implementation via e-mail to all authors. In case we receive no reply within a week, we solicit the authors again and wait another week for a reply. If we eventually get the implementation, we spend up to eight hours to make it run in our testing environment and deem it as functional or not. Moreover, we require support for *browser automation*. Since our analysis is based on a large-scale measurement, we discard those tools which are not supported by standard browser automation frameworks. In particular, we discard those tools which are integrated into a web browser that cannot be controlled using a modern version of Selenium [8], Puppeteer [7] or Playwright [6]. The full list of tools is shown in Appendix A.

After considering for evaluation 18 tools presented in the literature, we were left with just eight tools that we managed to download and put into operation. The evaluated tools are:

(1) Project Foxhound [32]: a modified version of Mozilla Firefox supporting dynamic taint tracking;
(2) PanoptiChrome [29]: a modified version of Chromium supporting information flow analysis;
(3) JSFlow [27]: an experimental JavaScript engine with information flow control capabilities, which we evaluate through its deployment within the Chromium browser used for research on browser fingerprinting [46];
(4) JEST [20]: a dynamic information flow analyzer for JavaScript based on monitor inlining and value boxing;
(5) IF-Transpiler [44]: a hybrid approach for information flow control of JavaScript operating in two stages, transpilation and monitor inlining;
(6) GIFC [39]: a dynamic information flow analyzer built on top of Linvail [19], a dynamic analysis tool for JavaScript with fine-grained tracking of runtime values by means of metaprogramming;
(7) LinvailTaint [18]: a dynamic taint tracker based on Linvail;
(8) JalangiTT [11]: a dynamic taint analysis tool inspired to Ichnaea [30] (that is not publicly available) and based on Jalangi [45], a generic dynamic analysis framework based on code-level instrumentation.

The significant gap between the number of considered tools and the number of evaluated tools already leads to a first insight of our research: *most analysis tools for JavaScript are unavailable, difficult to automate or do not work on modern setups.*

The 18 analysis tools that we considered are based on different architectures: ten are based on browser modifications, two are implemented as browser extensions and the remaining six are based on code rewriting, i.e., the code is instrumented for analysis purposes. Out of the eight tools that we managed to run, five are based on code rewriting and three are based on browser modifications. This leads to the second insight of our preliminary evaluation: *analysis tools based on code rewriting are easier to setup and maintain than tools based on other architectures, e.g., browser modifications.*

2.2 Web Measurement

We assess the different criteria by running the tools on a list of popular real-world websites, which offer a common playground for research on web security measurements. In particular, we build our analysis on the Top 10k domains included in the Tranco ranking [38] generated on 27 September 2024. Recall that different tools are based on different architectures. Tools based on browser modifications can be used off-the-shelf to access the websites under analysis, while tools based on code rewriting might require additional work for browser integration. If the browser integration layer is provided by the authors of the tool, we reuse it as it is, otherwise we implement our own integration layer based on a web proxy that instruments both the inline and the external scripts by calling the tool.

Before starting our web measurement, we performed a *pre-analysis* step to identify tools suffering from obvious flaws and filter out those which are too slow for any reasonable practical adoption, thus constituting a bottleneck for our investigation. In particular, we let each tool access the homepages of the top 1,000 domains of Tranco that are accessible using a standard web browser (Google Chrome), setting a timeout of five minutes per page. Since most websites should take around five seconds to fully render [22], this means accepting an overhead of around 60x on average for the analysis phase. Our pre-analysis showed that JSFlow triggered the timeout for all the websites under analysis, while all the other tools managed to complete the analysis on more than 50% of the websites (sometimes many more). Based on this, we removed JSFlow from our evaluation pipeline. We also dropped GIFC from our evaluation, because it is based on a legacy version of Linvail which we did not manage to configure correctly. Still, we observe that our evaluation includes LinvailTaint, which is based on a more recent version of the same analysis framework. We eventually evaluated the remaining six tools on the 6,921 domains of Tranco which are accessible using a standard web browser.

3 Compatibility Evaluation

JavaScript is a complex and living language, undergoing yearly revisions. This means that existing analysis tools for JavaScript may support just part of the language or become outdated as soon as JavaScript is extended with new features. A key question for existing analysis tools is whether they are *compatible* with the modern Web, in the sense that they can potentially be used to analyze production scripts running on live websites. We define compatibility as the ability of an analysis tool to process and analyze existing scripts from real-world websites without breaking. For example, compatibility violations might occur when a tool is unable to parse a script because it uses some unsupported JavaScript features or cannot perform the analysis after parsing, e.g., because the script instrumentation process fails.

3.1 Measurement Methodology

Since we define compatibility as the ability of an analysis tool to (i) process and (ii) analyze existing scripts from real-world websites without breaking, we correspondingly structure our analysis in two parts, discriminating between *syntactic compatibility* and *eventual compatibility* as defined below. We perform our analysis on all the scripts available in the homepages of the analyzed websites. In total, we found 148,029 scripts included or inlined within the homepage of the websites under analysis, reducing to 110,533 scripts after deduplication based on the MD5 hash of their code.

3.1.1 *Syntactic Compatibility.*

To understand whether an analysis tool processes existing scripts correctly, we base our investigation on JavaScript parsing. In particular, we observe that JavaScript is based on a living standard called ECMAScript (ES), which is updated on a yearly basis since 2015 to introduce new language features and refine some of the limitations of the original JavaScript design.

Since ES espouses backward compatibility, each new edition of the JavaScript language is defined as a superset of the previous edition. We can then leverage the following methodology to determine whether a tool t is *syntactically compatible* with a script s, i.e., it can parse it correctly. We first try to parse the script's code with parsers for different versions of the language, ordered from ES5 (the standard before 2015) through the latest ES version at the time of writing. The first parser correctly recognizing the script provides the resulting version number V_s. We finally compare V_s with the ES version V_t supported by the tool t and we say that t is syntactically compatible with s if and only if $V_s \leq V_t$.

In our analysis, we performed a manual investigation to identify the ES versions supported by the different tools from trusted sources. When the tool is a patched version of a popular browser, we relied on information provided by MDN's `browser-compat-data` [5], a widely-used project offering data about compatibility among browsers. For the other tools, we took into account the supported version of ES declared by the authors in the research article. If this information is unavailable, we reconstructed the supported ES version by inspecting the source code of the tool's parser and identifying the recognized syntactic constructs.

3.1.2 *Eventual Compatibility.*

Although syntactic compatibility is helpful, because it tells us whether existing tools can possibly be used off-the-shelf on the modern Web, one might argue that parsing issues are easily solved by means of *transpilers* like Babel [1], which can compile existing code to different ES versions to support backward compatibility. We then used Babel to transpile all the scripts creating compatibility issues down to ES5, which is supported by all the tools under investigation. After this step, we log all the internal errors of the tool t produced when analyzing the script s (transpiled, if needed) and we stipulate that t is *eventually compatible* with s if and only if no fatal error is encountered by the tool during the transpilation (when necessary) and analysis phases. Specifically, we consider the following types of errors as fatal: transpilation errors, instrumentation errors (for tools based on code rewriting), parsing errors after transpilation, and crashes during execution.

A challenge we encountered with transpilation was the support for native modules. Introduced in ES6, this feature allows JavaScript programs to be split into isolated components, which are then loaded using `import` statements. Unfortunately, unlike traditional scripts, ES5 does not provide an easy way to simulate such semantics. To address this problem, we first use Babel to transform native modules into CommonJS [4], where `import` statements are replaced with function calls. The function in question, known as `require`, is not defined in browser environments. Hence, we leverage browserify [2] to bundle all CommonJS modules into a single ES5 script, linking them through an implementation of the `require` function. Finally, we include this bundle in the website, replacing the original program based on native modules.

3.1.3 *Compatibility.*

Note that syntactic compatibility and eventual compatibility are independent concepts, i.e., a tool may satisfy one property, both properties, or neither. We just say that a tool t is *compatible* with a script s when t is both syntactically and eventually compatible with s, i.e., the parsing and analysis phases operate correctly without transpilation.

3.2 Empirical Results

Figure 1 reports for each tool the percentage of scripts where the tool is syntactically compatible, eventually compatible and compatible. The first observation we make is that four out of six evaluated tools have a syntactic compatibility rate of 69%, i.e., they cannot parse about one third of the production scripts running in live websites, because their supported ES version is too old (ES5). Unfortunately, this cannot be generally fixed by transpilation to an older ES version. The eventual compatibility rate of JEST and IF-Transpiler is 32% and 16% respectively, meaning that less than one third of the scripts can be analyzed with these tools, despite transpilation to ES5 using a state-of-the-art tool like Babel. LinvailTaint can take some advantage from transpilation: its compatibility rate is 47%, but its eventual compatibility rate is 63%, meaning that the use of transpilation allows it to recover around 16% of the scripts under analysis. The root cause of the eventual compatibility issues faced by JEST, IF-Transpiler and LinvailTaint are instrumentation errors: although transpilation and parsing succeed, the tools are unable to correctly instrument the scripts for analysis. JalangiTT is the tool reaping most benefits from transpilation, because its compatibility rate is 67%, but its eventual compatibility rate is 96% (+29%). Eventual compatibility issues affecting JalangiTT are primarily crashes, meaning that the analysis starts but unexpectedly terminates with a fatal error. To sum up, *many existing analysis*

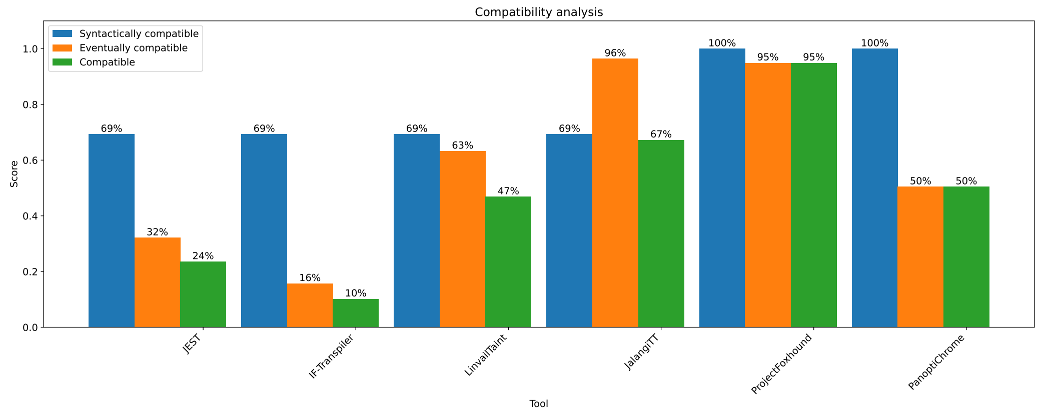

Figure 1: Compatibility analysis of the different tools.

tools for JavaScript support only legacy ES versions and are unable to correctly parse scripts on modern websites; while transpilation to ES5 can mitigate this issue for tools that already have a relatively high compatibility rate, it is far from a perfect solution in practice.

Project Foxhound is the most robust solution for the modern Web, outperforming all the tools seen so far in terms of compatibility, because it is able to successfully analyze 95% of the collected scripts out of the box. This can be explained by the fact that Project Foxhound is the most recent tool and is implemented on top of a commercial browser like Mozilla Firefox, thus being more prepared to face the complexity of JavaScript. However, this is not a general rule for modified browsers. Although PanoptiChrome is built on a modern Chromium version, it achieves only a 50% compatibility rate in practice, mainly because the tool becomes unresponsive when analyzing complex websites. The authors of PanoptiChrome did not encounter this issue, likely because they operated the tool manually without an automation framework, terminating it after a fixed timeout [29], whereas our approach relies on Playwright to detect when a page has finished loading, which throws an exception whenever it loses control of PanoptiChrome. This shows that *even recent tools that have been designed to be compatible with the modern Web can still face troubles when analyzing complex websites.*

To provide additional insights on compatibility issues, we investigate which JavaScript features are most often used by existing scripts in the wild. This information is useful, because it allows tool designers to understand which JavaScript features are most important to prioritize when developing new analysis tools. The top ten language features are shown in Table 1: block scoping and arrow functions are used on more than 20% of the analyzed scripts, hence it is crucial to support them in analysis tools. Remarkably, most of the most popular language features have been introduced in ES2015. This teaches us that *any realistic JavaScript core should include a significant number of ES2015 features. Features introduced in newer versions of ES are not nearly as widespread.*

4 Transparency Evaluation

Compatibility is a necessary precondition to run any analysis tool, but it does not suffice to ensure that the tool operates as intended. Ideally, an analysis tool should never modify the behavior of the analyzed code unless this is required to enforce security. In line with existing work [35], we say that a tool is *transparent* on a website if and only if it preserves the website's original behavior when the website already respects the intended security policy.

Table 1: Top ten JavaScript features observed in the wild

Construct	ES Version	Frequency
Block scoping	ES2015	25%
Arrow functions	ES2015	24%
Destructuring	ES2015	16%
Template literals	ES2015	12%
Spread	ES2015	12%
Classes	ES2015	8%
Object spread	ES2018	8%
For-of	ES2015	7%
Computed properties	ES2015	6%
Async functions	ES2017	6%

4.1 Measurement Methodology

Transparency is harder to measure than compatibility, because the behavior of live websites is unknown and we cannot determine what is their "intended" security policy. In our measurement, we restrict the focus to the most liberal policy which does not enforce any security requirement, i.e., all the information flows are allowed. This policy should never modify the website's behavior and thus identifies a minimal baseline for transparency.

To evaluate transparency on a website w, we access the homepage of w with both a standard browser b and an analysis tool t. We then wait enough time to allow both b and t to render the HTML document, which we detect based on the load event; if we are unable to detect the event within two minutes in any of the two runs, we discard the website because it did not complete loading. We collect relevant events within the two runs to build corresponding *execution traces* of the website w, which characterize the website behavior. Finally, we compare the execution trace observed by b

against the execution trace observed by t. If there is a mismatch between the two traces, we detect a transparency violation for t.

4.1.1 Defining and Comparing Execution Traces. Choosing the relevant events abstracting an execution trace is a difficult task, because there is a delicate trade-off to consider. On the one hand, we would like to abstract the execution into as many events as possible to precisely characterize the website behavior and detect most transparency violations. On the other hand, observing too many events might make the analysis brittle and lead to the detection of transparency violations even when they do not actually take place. To avoid over-reporting, our approach is prudent and abstracts execution traces as sets of *uncaught exceptions*, which are a strong signal that there was something wrong with the program run.

In particular, let E_b and E_t stand for the sets of uncaught exceptions detected in the standard browser b and in the analysis tool t when visiting a given website: we detect a transparency violation if and only if $E_t \nsubseteq E_b$, i.e., when the tool observes an uncaught exception that was not observed in a standard web browser. We do not check the weaker requirement $E_t \neq E_b$, because the JavaScript execution model is asynchronous and even the presence of the load event is not a bullet-proof guarantee that the HTML document was entirely rendered, hence set equality is difficult to achieve. However, since the tool execution is normally slower than the browser execution, the tool execution is expected to be a prefix of the browser execution, i.e., for a transparent tool we expect $E_t \subseteq E_b$.

4.1.2 Dealing with Non-Determinism. The behavior of websites might be non-deterministic for different reasons. For example, websites might serve different content depending on the requesting user agent. This means that a standard browser b and an analysis tool t might observe different execution traces not because transparency was broken, but because the website served different content to them. Moreover, even different executions on the same browser or tool might exhibit different behavior for generic reasons, e.g., different scripts may be loaded at runtime due to highly dynamic components like advertisement scripts [41].

To mitigate the impact of non-determinism, we rely on a record and replay technique proposed in prior research [37]. We use the Web Page Replay tool from Google's Catapult project [3] to record all the requests and responses fired from the browser b and deterministically reproduce them in the tool t. Although Web Page Replay is a robust and useful tool, it is not perfect. To further mitigate non-determinism, we perform five executions of the recorded trace and we identify the execution trace to use in our transparency analysis based on majority voting. If there is no execution trace represented at least three times in our five executions, we conclude that Web Page Replay is not operating correctly on the website under analysis and we exclude it from our transparency assessment.

4.1.3 Avoiding Measurement Artifacts. A subtle aspect that we have to deal with is that our own measurement approach might break transparency. Since our methodology abstracts execution traces in terms of events that can only be observed inside the browser, i.e., exceptions, we cannot rely on external components like network proxies to ensure that transparency is not accidentally broken. In principle, any new script injected in the website by our measurement approach might negatively interfere with the website logic or

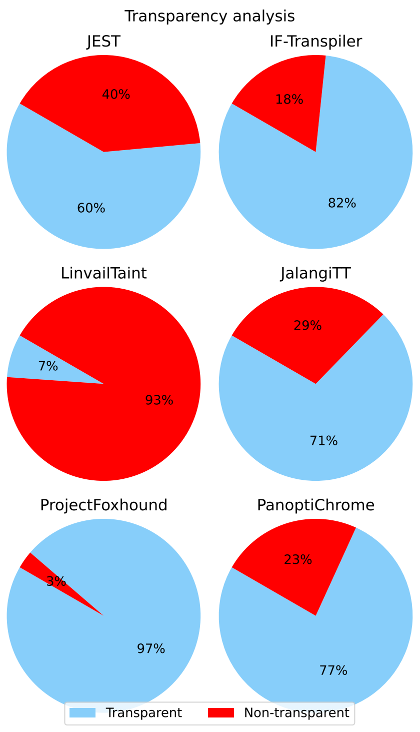

Figure 2: Transparency analysis of the different tools.

with the analysis tool under scrutiny. To reduce such risks, we use a very lightweight instrumentation approach which is designed to minimize room for transparency violations. In particular, we just register an event handler via the `addEventListener` function in order to capture uncaught errors raised by the website scripts.

4.2 Empirical Results

For each tool, we measure transparency just on those websites where the tool satisfies eventual compatibility for at least one script, i.e., the tool correctly parses and analyzes some scripts of the website possibly after transpilation. Otherwise, the tool would not perform any analysis at all, which would make it artificially easier to preserve transparency and bias our measurement. Before presenting results, we report that our strategy for addressing non-determinism through majority voting is definitely successful: in almost all websites, we were able to identify a predominant execution trace, with just 13 cases in total where this was not feasible.

Figure 2 reports the number of websites where each tool was able to operate transparently. The figure leads to several interesting findings, motivating the importance of a fine-grained analysis distinguishing compatibility and transparency as done in our evaluation. We observed that JEST and IF-Transpiler are incompatible with many scripts, with eventual compatibility rates of only 32% and 16%, respectively. However, their transparency rates are higher, at 60% for JEST and 82% for IF-Transpiler, indicating that they do not trigger new exceptions on most websites during analysis. In other words, once these tools are able to perform their analysis, we have relatively high assurance about their respect of the original website semantics. Conversely, LinvailTaint enjoyed a higher eventual compatibility rate of 63%, but suffers from a very low transparency rate equal to 7%. This means that in virtually all the analyzed websites we are able to detect new exceptions introduced by the presence of the tool, which might modify the website semantics and make the analysis unreliable. JalangiTT and Project Foxhound, which showed the best figures in terms of eventual compatibility, also preserve transparency in most websites. This is particularly apparent for Project Foxhound, showing a transparency rate of 97%, as opposed to the 71% of JalangiTT. Lastly, in addition to having a lower eventual compatibity rate than Project Foxhound, PanoptiChrome exhibits a lower transparency rate of 77%. Our evaluation shows that *transparent information flow control through code rewriting is possible but difficult to implement correctly, because even the enforcement of an empty security policy might trigger new exceptions in live websites. Existing solutions based on browser modifications offer higher transparency guarantees than tools based on code rewriting.* For space reasons, we discuss the main types of observed transparency violations in Appendix B.

5 Coverage Evaluation

Coverage estimates how many information flows are successfully detected by an analysis tool. It is reasonable that existing analysis tools have limitations leading to both false positives and false negatives. Unfortunately, most analysis tools do not undergo a comprehensive coverage evaluation, meaning that their effectiveness on real-world JavaScript code is unknown.

5.1 Measurement Methodology

To measure the effectiveness of different information flow monitors, we would need a *ground truth* of JavaScript programs annotated with their correct set of information flows. In prior work Sayed et al. crafted such a benchmark, which was used to assess the effectiveness of IF-Transpiler [44]. The same benchmark was also used in an empirical evaluation of different information flow monitors for JavaScript performed by Scull Pupo et al. [39]. Unfortunately, this existing benchmark only includes 28 small test cases covering a tiny subset of JavaScript features and none of the standard library functions, hence it does not capture the complexity of real-world JavaScript. Moreover, 23 of the 28 test cases involve the monitoring of implicit information flows, hence they cannot assess the effectiveness of taint trackers designed to detect just explicit flows.

In line with the rest of the paper, we aim to evaluate existing tools against real-world websites to mitigate the bias associated with using a synthetic and limited benchmark. However, as discussed above, there is currently no ground truth of information flows for real-world websites. Without an accurate manual scrutiny of websites, it is impossible to determine whether a tool is missing some information flows (false negatives) or detecting flows that do not occur in practice (false positives). We then use a comparative approach to collect flows from the same websites with different tools and the same security policy, comparing their outputs for agreement. Additionally, we apply a simple heuristic to detect explicit flows based on syntactic matches.

5.1.1 Policy Specification. Differently from transparency, coverage evaluation requires tools to be configured with a non-trivial information flow policy that logs relevant information flows. This task is challenging because different tools use different policy languages, support different sources and sinks, and track different types of flows. Some tools do not even support customizable policies and run with a hard-coded policy. Therefore, we look for a policy that is specifiable in all the considered tools, while being simple enough to reduce development burden and minimize room for inconsistencies. We then consider a common policy that reports information flows originating from client-side storage, i.e., `document.cookie` and `localStorage.getItem`, and reaching the network sink `fetch(`*target*`)`, where *target* is the target domain of the network request. This policy is supported by all the tools, because communication of client-side identifiers over the network is a typical concern of web privacy studies [24]. Note that the considered tools also support other popular network sinks, such as XMLHttpRequest, yet not all of them log the target domain.

Project Foxhound and JalangiTT already hard-code an extended version of this policy. Therefore, we simply filter out the information flows involving irrelevant sources or sinks. IF-Transpiler and LinvailTaint do not offer a straightforward way to define sources and sinks. IF-Transpiler requires manual insertion of statements in the instrumented code to upgrade and check the security level of variables at source and sink locations. LinvailTaint instead reserves two variables, one tagged as source and the other one as sink of the analysis. We initialize the analysis by installing wrappers around source and sink functions that execute tool-specific instructions for upgrading and checking the security levels of return values and arguments, respectively. We could not configure the policy in the case of JEST due to a number of generic problems and the lack of example policies in the JEST repository that could guide our attempts. Our experience aligns with that of other independent researchers who did not include JEST in their experimental evaluation [39].

Since each tool represents and reports flows using its own format, we parse and uniformly model information flows as triples of the form (src, snk, w). *src* and *snk* identify respectively the source and the sink of the flow, while w determines the website where the flow has been identified. The latter information is necessary to define the flow identity, as two flows with the same features but found on different websites should be considered distinct. Note that this representation does not consider whether a flow is explicit or implicit. Generally, ignoring this distinction could result in inaccurate measurements. In fact, taint trackers capture fewer information flows than general tools for information flow analysis, since the former find only explicit flows, while the latter detect both explicit and implicit flows. However, neither of the tools capable of capturing implicit flows, namely IF-Transpiler and PanoptiChrome, provide sufficient information to distinguish between the two types of flows. Therefore, we log all flows detected by these tools indiscriminately and draw our conclusions with awareness of this limitation.

5.1.2 Defining Agreement. We compare each flow identified by a tool t with two independent sets of observations. First, we check whether some other tool t' found the same flow: if this is the case, the flow is likely a true positive. The intuition here is that false positives are introduced by over-approximations and analysis errors which are unlikely to be the same for different tools. We refer to this approach as *cooperative agreement.* Moreover, we check whether

Table 2: Number of information flows detected by the tools and estimate of true positives according to cooperative agreement ($\mathrm{TP_c}$) and syntactic agreement ($\mathrm{TP_s}$)

	Flows	$\mathrm{TP_c}$	$\mathrm{TP_s}$	TP	TP%
IF-Transpiler	0	0	0	0	–
LinvailTaint	35	5	24	24	69%
JalangiTT	67	10	33	33	49%
Project Foxhound	919	28	862	862	94%
PanoptiChrome	128	13	95	95	74%

the flow was identified by a simple heuristic to detect explicit flows based on syntactic matches, hence we call this strategy *syntactic agreement*. In particular, we look for a correspondence between the content of client-side storage and the payload of network requests by using longest substring matching: if the longest common substring between the value of a client-side storage item and the payload (path + query string) of a network request has length at least 8, it is likely that some value was read from the client-side storage and sent over the network, hence the reported flow is likely a true positive. To make this heuristic approach more precise, we remove timestamps from network requests before matching: in fact, timestamps in network requests are often generated dynamically rather than retrieved from client-side storage, which would lead to spurious syntactic matches. By using this automated technique, we are able to detect (likely) true positives at scale even in absence of a ground truth. We complement this automated analysis with some manual investigation to refine our findings.

5.2 Empirical Results

Table 2 presents the findings of our coverage evaluation. The first column reports the number of information flows detected by each tool. $\mathrm{TP_c}$ and $\mathrm{TP_s}$ are the estimated number of true positives according to cooperative and syntactic agreement, respectively. TP indicates the number of flows meeting at least one of the agreement heuristics and TP% normalizes TP to the total number of flows.

The first observation we make is that syntactic agreement outperforms cooperative agreement in estimating true positives. On one side, the small number of true positives based on cooperative agreement suggests that each tool produces outcomes which are poorly shared by other tools. On the other side, the large number of true positives based on syntactic agreement confirms that our simple matching criteria are sufficient to capture apparent data transfers. Project Foxhound is the most effective tool in terms of detected information flows (919), with the second best-performing tool being PanoptiChrome with just 128 detected flows. Moreover, most of the information flows detected by Project Foxhound are marked as true positives by our heuristics (94%). After manually reviewing a random sample of 10 potential false positives, it turns out that they are actually true positives missed by our criteria for syntactic agreement. More precisely, most of these cases occur when storage items are transformed (e.g., encoded in Base64) before reaching a network sink, or when communicated values are actually timestamps or short strings from client-side storage. This further confirms that Project Foxhound far surpasses the other tools in terms of coverage, proving to be a dependable tool when it comes to searching for information flows on the modern Web.

IF-Transpiler finds no information flows at all in the wild. Since this finding is surprising, we used the small information flow benchmark by Sayed et al. [44] to confirm that the tool is configured correctly and is able to identify the information flows in the benchmark. We also tested variants of the benchmark with our set of sources and sinks, confirming that the tool is operating correctly in a small synthetic environment. To better understand why IF-Transpiler is basically unusable in the wild, we restricted our focus to a sample of those websites where it enjoyed eventual compatibility and transparency. This way, we exclude compatibility issues and transparency violations as the reasons leading to information flows being undetected. What we observed is that, for all such websites, also the other tools did not find any flows, except one detected by Project Foxhound. This suggests that the websites where IF-Transpiler operate correctly are so simple that they really have no information flows, while on the other websites we may attribute ineffective coverage to analysis errors.

LinvailTaint and JalangiTT are the only language-based tools capable of finding information flows, identifying 35 and 67 explicit flows, respectively. Our agreement heuristics estimate 69% true positives for LinvailTaint compared to 49% for JalangiTT, indicating that the former may be more precise than the latter. However, both these percentages suggest a potential high number of false positives, especially for JalangiTT, where just one in two detected flows is likely a true positive. To unveil the reason behind this potential prevalence of false reports, we manually validated a random sample of 10 flows for both tools, aiming to sort these cases as either confirmed false positives or not. We found that only 2 in 10 instances from LinvailTaint were confirmed as false positives, with the remaining cases being misclassified by our matching heuristics. In fact, we identified values from client-side storage reaching a network sink after being decoded, primarily from Base64. In contrast, we confirmed 8 in 10 instances from JalangiTT to be false positives, leading us to conclude that this tool operates with lower precision.

In the end, our analysis shows that *existing information flow analyzers for JavaScript show a low agreement rate and may suffer from potential false positives, despite their use of dynamic analysis.*

6 Performance Evaluation

Contrary to the other three requirements, a satisfactory performance in terms of running times is not a stringent requirement of analysis tools, because different users have different computational resources and desiderata. Still, performance is particularly important for large-scale analyses of the Web, such as those carried out in traditional web security measurements.

6.1 Measurement Methodology

Comparing the performance of different tools is surprisingly subtle, because different tools suffer from different failures. For example, if a tool is not eventually compatible with most of the scripts on a website, its performance figures cannot be trusted because most of the JavaScript code does not undergo the analysis phase. We then measure the analysis time of a tool t just over those websites where (i) t is eventually compatible with all the scripts on the website and

Table 3: Performance statistics

Tool	Overhead
LinvailTaint	23.9x
JalangiTT	8.6x
Project Foxhound	1.4x
PanoptiChrome	36.7x

(*ii*) t is transparent there. This way, we know that all the scripts on the websites have been analyzed without breakage. Computing the average analysis time of t on such websites does not directly enable a comparison with other tools though. For example, if a tool enjoys transparency just on very simple websites making limited use of JavaScript, its average analysis time would be biased towards low numbers just because such websites are easy to analyze. We then evaluate performance in terms of the analysis overhead introduced over a traditional web browser on the same websites. In particular, let W_t be the set of websites that t can analyze based on our inclusion criterion above. If $T_b(w)$ is the time required to render the website w in a standard web browser and $T_t(w)$ is the time required to analyze the same website using the tool t, we compute the overhead of t as the average overhead across all websites in W_t, i.e., $\text{avg}_{w \in W_t}(T_t(w)/T_b(w))$. Note that we do not measure the performance of JEST and IF-Transpiler because we have been unable to detect any information flows with those tools in the wild, meaning that their practical adoption is infeasible.

6.2 Empirical Results

Table 3 reports the outcome of our performance evaluation. Once again, Project Foxhound outperforms its competitors, with an average overhead of just 1.4x with respect to a standard web browser. This result can be justified by the fact that all taint tracking operations are performed natively within the browser. Nonetheless, the other modified browser we consider, PanoptiChrome, suffers from a pronounced overhead of 36.7x. This is mainly due to the authors' recommendation to disable certain optimizations in the JavaScript engine, including the JIT compiler, which would otherwise prevent the tool from running. Additionally, the tool's ability to detect implicit flows, as well as explicit ones, could further contribute to slow down the execution. This shows that *information flow tracking can be efficiently performed through browser modifications, yet detection of implicit flows may significantly slow down the analysis and make it infeasible for web measurements.*

Regarding language-based tools, we observe that LinvailTaint and JalangiTT suffer from a significant overhead of around 23.9x and 8.6x, respectively. After inspecting the source code of these two tools, we suspect that this extra time is necessary to make the analysis precise, i.e., these tools reimplement several native operations in JavaScript with the goal of exposing their internals to the analysis. It is worth noting that the authors of Linvail, which is used to create LinvailTaint, report that their library experiences an overhead one order of magnitude higher than that of Jalangi, which JalangiTT relies on [19]. However, our findings reveal the overheads between the derived tools to be much closer. This might be attributed to the fact that their benchmark comprises CPU-intensive programs, a common choice for performance evaluation purposes. In turn, this study uses real-world web pages as its benchmark, where JavaScript primarily facilitates interactivity rather than engaging in heavy computations. Since our automated analysis does not emulate user interaction, we may expect that some portions of JavaScript code are not actually executed at all. In the end, we note that *analysis tools based on code rewriting incur major performance overheads, which may be attributed to the need of reimplementing native functions in JavaScript to make them visible to the analysis.*

7 Related Work

There exist a few surveys on the dynamic analysis of JavaScript, some of which focusing also on security applications and information flow control [12, 16, 50]. A major difference of this work with respect to traditional surveys is that we do not just review existing papers, but we experimentally measure the effectiveness of different tools on live websites based on systematic criteria. The goal of our work is informing the community about the actual effectiveness of existing tools and identifying room for improvement for future analysis tools. It is worth noticing that prior work on information flow control for JavaScript already compared the effectiveness of different tools for information flow control. For example, the authors of GIFC compared their tool against competitors on a small benchmark of synthetic JavaScript programs designed to test the features of different information flow analyzers [39]. This evaluation on a synthetic benchmark draws overly optimistic conclusions and does not shed light about the effectiveness of existing tools on real-world programs, such as the scripts that we collect from prominent websites in the Tranco list. Moreover, our analysis uses more sophisticated and systematic evaluation criteria formulated in terms of compatibility and transparency, which elucidate the main shortcomings of existing tools when they face the Web. Finally, we mention that our research methodology draws inspiration from traditional web measurement studies [23]. Prior work in the field performed measurements of JavaScript programs to analyze web tracking [11], fingerprinting [46] and XSS vulnerabilities [48].

8 Conclusion

Dynamic security analysis tools for JavaScript do not fare well against the Web. The only effective solution given the current state of the art is Project Foxhound, which is a tool based on browser modifications, rather than code rewriting. Unfortunately, our experience with prior work suggests that tools based on browser modifications do not age well, because they are difficult to maintain and easily become outdated. Indeed, out of ten tools based on browser modifications considered for evaluation, we just managed to run three, two of which are very recent. This motivates additional research on how to combine the best of the two worlds, i.e., how to design a JavaScript analysis tool which is effective and efficient like browser-based solutions, while being as simple to maintain as solutions based on code rewriting. We encourage the community to investigate this challenging problem and propose new solutions.

Acknowledgments. This research was supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] 2024. Babel. https://babeljs.io/
[2] 2024. Browserify. https://browserify.org/
[3] 2024. Catapult. https://chromium.googlesource.com/catapult/
[4] 2024. CommonJS. https://wiki.commonjs.org/wiki/CommonJS
[5] 2024. mdn/browser-compat-data. https://github.com/mdn/browser-compat-data
[6] 2024. Playwright. https://playwright.dev/
[7] 2024. Puppeteer. https://pptr.dev/
[8] 2024. Selenium. https://www.selenium.dev/
[9] 2025. DynSecAnJS. https://github.com/eleumasc/DynSecAnJS
[10] Zubair Ahmad, Stefano Calzavara, Samuele Casarin, and Ben Stock. 2024. Information flow control for comparative privacy analyses. *Int. J. Inf. Sec.* 23, 5 (2024), 3199–3216. doi:10.1007/S10207-024-00886-0
[11] Zubair Ahmad, Samuele Casarin, and Stefano Calzavara. 2024. An Empirical Analysis of Web Storage and Its Applications to Web Tracking. *ACM Trans. Web* 18, 1 (2024), 7:1–7:28.
[12] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.* 50, 5 (2017), 66:1–66:36.
[13] Aslan Askarov and Andrei Sabelfeld. 2009. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. IEEE, 43–59.
[14] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *POPL*, John Field and Michael Hicks (Eds.). ACM, 165–178.
[15] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *POST (Lecture Notes in Computer Science, Vol. 8414)*, Martín Abadi and Steve Kremer (Eds.). Springer, 159–178.
[16] Nataliia Bielova. 2013. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebraic Methods Program.* 82, 8 (2013), 243–262.
[17] Stefano Calzavara, Samuele Casarin, and Riccardo Focardi. 2025. Dynamic Security Analysis of Javascript: Are We There Yet?: Dataset. https://doi.org/10.5281/zenodo.14774184
[18] Laurent Christophe. 2023. LinvailTaint. https://github.com/lachrist/aran/blob/664f0a304b555bcb106f24e72734ad8c88dac429/graveyard/test/live/linvail-taint.js
[19] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *SANER*. IEEE Computer Society, 260–270.
[20] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *CCS*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 629–643.
[21] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for javascript. In *PLDI*, Michael Hind and Amer Diwan (Eds.). ACM, 50–62.
[22] Sourojit Das. 2024. How fast should a Website Load in 2024? | BrowserStack. https://www.browserstack.com/guide/how-fast-should-a-website-load
[23] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. 2022. Reproducibility and Replicability of Web Measurement Studies. In *WWW*, Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (Eds.). ACM, 533–544.
[24] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1388–1401. doi:10.1145/2976749.2978313
[25] GitHub. 2022. The Top Programming Languages. https://octoverse.github.com/2022/top-programming-languages.
[26] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 748–759.
[27] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC*, Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.). ACM, 1663–1671.
[28] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *SAS (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 238–255.
[29] Rahul Kanyal and Smruti R. Sarangi. 2024. PanoptiChrome: A Modern In-browser Taint Analysis Framework. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*, Tat-Seng Chua, Chong-Wah Ngo, Ravi Kumar, Hady W. Lauw, and Roy Ka-Wei Lee (Eds.). ACM, 1914–1922. doi:10.1145/3589334.3645699
[30] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Trans. Software Eng.* 46, 12 (2020), 1364–1379.
[31] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *FSE*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 121–132.
[32] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. 2022. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In *EuroS&P*. IEEE, 236–250.
[33] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 96.
[34] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *CCS*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 1193–1204.
[35] Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.* 4, 1-2 (2005), 2–16.
[36] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *NDSS*. The Internet Society.
[37] Marius Musch and Martin Johns. 2021. U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild. In *USENIX Security*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2935–2950.
[38] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *NDSS*. The Internet Society.
[39] Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. 2018. Practical Information Flow Control for Web Applications. In *RV (Lecture Notes in Computer Science, Vol. 11237)*, Christian Colombo and Martin Leucker (Eds.). Springer, 372–388.
[40] IBM Research. 2006. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net [Accessed 18-04-2024].
[41] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. 2022. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2047–2064. https://www.usenix.org/conference/usenixsecurity22/presentation/roth
[42] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19.
[43] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*. The Internet Society.
[44] Bassam Sayed, Issa Traoré, and Amany Abdelhalim. 2018. If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript. *Comput. Secur.* 75 (2018), 92–117.
[45] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 488–498.
[46] Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld. 2021. EssentialFP: Exposing the Essence of Browser Fingerprinting. In *EuroS&P*. IEEE, 32–48.
[47] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023.
[48] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*. The Internet Society.
[49] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 655–670.
[50] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 50, 4 (2017), 59:1–59:34.
[51] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *ISSTA*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 49–59.
[52] Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *ISSTA*, Mauro Pezzè and Mark Harman (Eds.). ACM, 336–346.

A Tool Selection Details

Table 4 reports all the tools that we initially considered in our research. We mark with a gray background the tools which have been excluded from further investigation based on the criteria discussed in Section 2.1.

Table 4: Dynamic analysis tools considered in this work. Rows with a gray background include tools which have been excluded from further investigation for different reasons (see the Notes column)

Analysis tool	Type	Explicit flows	Implicit flows	Architecture	Notes
FLAX [43]	Dynamic	✓	×	Modified browser	Unavailable implementation
PersistentClientsideXSS [34, 48, 49]	Dynamic	✓	×	Modified browser	Broken browser automation
ChromiumTaintTracking [36]	Dynamic	✓	×	Modified browser	Broken browser automation
JSA [51]	Hybrid	✓	×	Modified browser	Unavailable implementation
JSBAF [52]	Hybrid	✓	×	Modified browser	Unavailable implementation
Project Foxhound [32]	Dynamic	✓	×	Modified browser	-
PanoptiChrome [29]	Dynamic	✓	✓	Modified browser	-
FlowFox [26]	Dynamic	✓	✓	Modified browser	Lack of browser automation
ZaphodFacets [14]	Dynamic	✓	✓	Browser extension	Failed to run on modern setup
ifc4bc [15]	Hybrid	✓	✓	Modified browser	Failed to run on modern setup
SIF [21]	Hybrid	✓	✓	Browser extension	Unavailable implementation
JSFlow [27]	Dynamic	✓	✓	Modified browser	Implementation from [46]
JEST [20]	Dynamic	✓	✓	Code rewriting	-
IF-Transpiler [44]	Hybrid	✓	✓	Code rewriting	-
GIFC [39]	Dynamic	✓	✓	Code rewriting	Built on top of Linvail [19]
LinvailTaint [18]	Dynamic	✓	×	Code rewriting	Built on top of Linvail [19]
Ichnaea [30]	Dynamic	✓	×	Code rewriting	Unavailable implementation
JalangiTT [11]	Dynamic	✓	×	Code rewriting	Built on top of Jalangi [45]

B Transparency Issues

To better understand why transparency was broken, we measure the frequency of the different types of exceptions occurring during the analysis and exposing transparency violations in the different tools (see Figure 3). TypeError is the most common type of exception in nearly all the tools. A manual inspection reveals that these cases mostly involve accessing properties of `undefined` or `null`. Specifically, for language-based tools, we observe that some identifiers involved in these errors are related to the analysis code. This suggests that the analysis performed by these tools is not robust and can easily alter the execution upon reaching invalid states. Moreover, we identify equivalent error messages with different formats between runs on Firefox and Project Foxhound. We conservatively mark these differences as false positives; despite our efforts to align the execution of the tool with that of a regular browser, we were unable to eliminate them completely. Finally, the majority of SyntaxError instances thrown by IF-Transpiler indicate that the tool is generating syntactically incorrect analysis code.

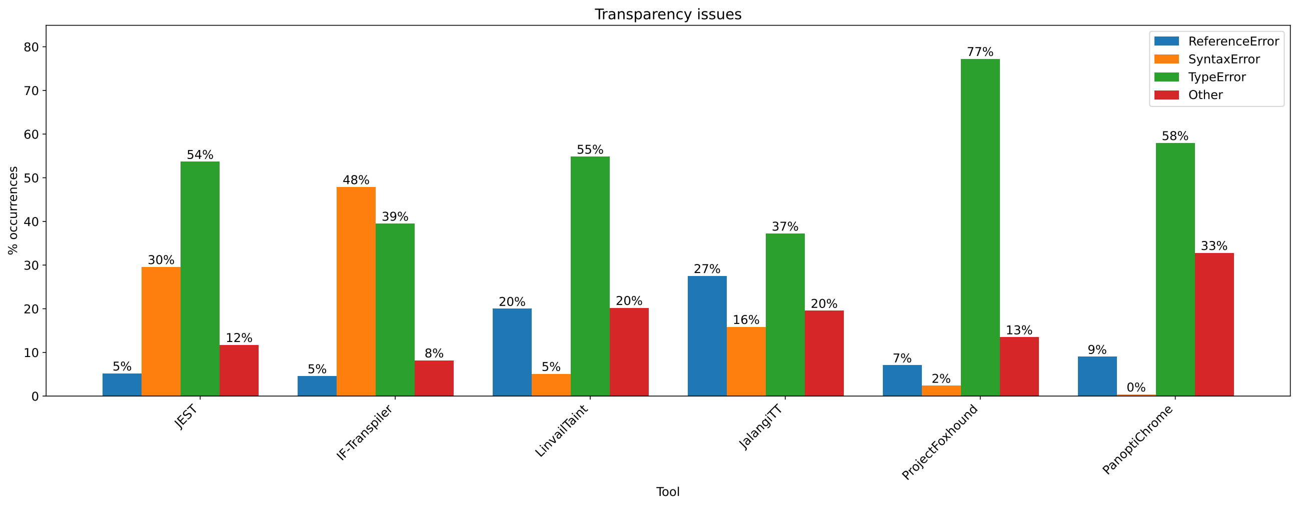

Figure 3: Classification of observed transparency issues.