iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.79689

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

Taintaru: DOM-based XSS Detection using Taint Tracking

Shriya Bhatia[1], Pooja Tupe[2]
University of Mumbai, Mumbai, India

Abstract: ***A DOM-based Cross-Site Scripting (DOM-XSS) attack is one of the most dangerous and common client-side security problems in today's web applications. DOM-based XSS attacks are a serious risk to single-page applications, particularly those using React, Vue.js, Angular, and Svelte. Current detection methods fall short in three key ways: they consume too many system resources, need direct access to application source code, and can't handle obfuscated attack payloads effectively. Even worse, they miss the unique rendering behaviors of different JavaScript frameworks. This paper reviews the existing solutions, approximately thirty publications spanning 2005 to 2025, and introduces a Chrome Manifest V3 browser extension that tackles the gaps in the existing solutions. The approach will use service worker-based taint tracking to catch DOM-XSS vulnerabilities without slowing down applications. The system will include a unified taint abstraction layer that works across multiple frameworks and employs machine learning to decode obfuscated payloads. Taking hints from React Fiber's reconciliation process, we've framed a conceptual idea of a delta re-analysis mechanism that will track taint propagation from network requests all the way to DOM manipulation—without touching the application code itself.***

Keywords: DOM-based XSS, Taint Tracking, Dynamic Taint Analysis, Cross-Site Scripting Detection, Web Security, JavaScript Security, Taint Propagation, Machine Learning for Security

I. INTRODUCTION

Cross-Site Scripting (XSS) was initially identified by CERT/CC in the early 2000s; it remains among the most prevalent web vulnerabilities monitored by OWASP [35]. Of its three principal variants, DOM-XSS presents the greatest detection challenge: malicious execution is confined entirely to the browser, with attacker-controlled sources such as *location.hash* or *window.name* supplying unsanitized data to dangerous sinks such as *innerHTML* or *eval()*, bypassing server-side filters and web application firewalls. Empirical studies confirm the scale of the problem: Lekies et al. [5] identified 6,167 unique vulnerabilities across 9.6% of the Alexa top 5,000 sites, and W. Melicher et al. [9] confirmed approximately 3.6% vulnerabilities of the top 10,000. This review consolidates the evolution of detection methodology, evaluates fifteen tools in a structured comparison, and situates the proposed *Taintaru* system within the research landscape.

II. BACKGROUND

A. Cross-Site Scripting: Taxonomy and Threat Model

According to OWASP categorization, there are three typical kinds of cross-site scripting:

(1) Reflected XSS: Attackers embed malicious script in a request (for example, through URL parameters). The server reflects it in the response, and the script then executes in the victim's browser without being stored.

(2) Stored XSS: The script is permanently stored in the backend (for example, database or message store) of the application, and then it is presented to every user who accesses the source of the compromised content.

(3) DOM-based XSS: The problem is caused completely by the way the DOM in the browser is handled; the server response may be harmless, but the client-side JavaScript first reads the data from the attacker-controllable source and then writes it to the dangerous sink.

Figure 1 shows how a client-side XSS attack occurs through tainted data flow. A malicious script is injected via the URL (source), passed through the application without proper validation (propagation), and finally executed in the browser using functions like document.write() (sink). This leads to execution of harmful code, such as stealing user cookies.

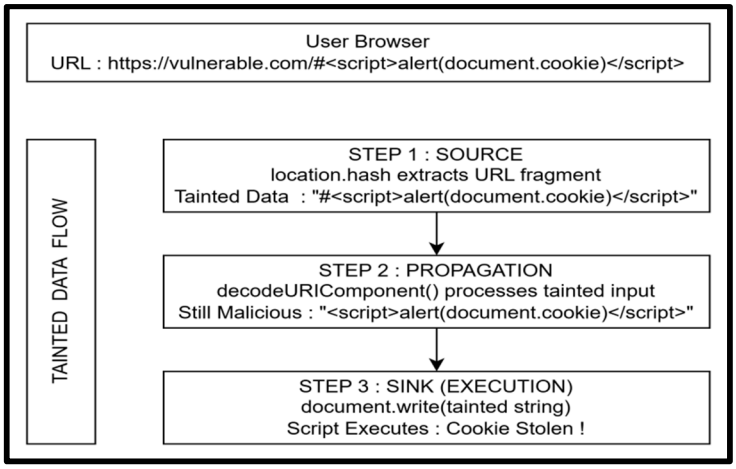


Figure 1: DOM-based XSS attack flow

Table 1 describes the differences between the three types of XSS across various aspects.

Aspect	DOM-based XSS	Reflected XSS	Stored XSS
Execution Context	Client-side only	Server-client roundtrip	Server-client roundtrip
Source	location.hash, postMessage(), URLSearchParams	GET/POST query parameter	Database field (comments, profiles)
Propagation	JS functions (decodeURIComponent() -> variables)	Server-side echo (no sanitization)	Server query -> response rendering
Sink	innerHTML, eval(), document.write()	HTML response (Results: <script>..)	HTML injection(class= ‘comment’> <script>..)
Server Awareness	None	Full request visible	Payload stored persistently
Impact Scope	Single victim (current tab)	Single victim (direct link)	Multiple victims (all page viewers)
Detection Difficulty	High (client-side, framework sinks)	Medium(server logs)	Low (DB content scanning)
Example Payload	#<script>alert(‘DOM XSS’)</script>	?search= <script>alert (‘Reflected’) </script>	<script>alert (‘Stored’) </script> in comment
Defense Layer	Client-side taint tracking	Server sanitization	DB sanitization + output encoding

Table 1: XSS Attack Types Comparison

B. Sources, Sinks, and Taint Propagation

A source is any browser API from which an attacker may inject controlled data (e.g., *document.URL, location.hash, localStorage*). A sink is an API capable of introducing executable content into the DOM (e.g., *innerHTML, eval(), document.write()*). Taint tracking formalizes surveillance of marked values between source and sink through four operations: introduction (labelling tainted

source data), propagation (transferring labels through operations), checking (inspecting labels at sinks), and sanitization (removing labels from validated data) [1]. TT-XSS [7] catalogued 10 sources, 9 sinks, and 21 transfer functions; PanoptiChrome [18] extended instrumentation to 675 browser APIs.

Table 2 summarizes primary sinks ordered by the level of severity.

Sink	Type	Risk Level	Example Attack
eval()	JavaScript	Critical	eval(location.hash)
Function()	JavaScript	Critical	new Function(location.search)
innerHTML	HTML	High	div.innerHTML= location.hash
outerHTML	HTML	High	element.outerHTML = postMessage()
document.write	HTML	Medium	document.write(decodeURL(hash))
setTimeout/setInterval	String -> Code	Medium	setTimeout(urlParams,0)

Table 2: DOM XSS Sink Categories

C. Detection Paradigms

Static analysis involves code or ASTs checking for taint flow paths without executing them, which is fast but prone to false positives due to dynamic JavaScript code [20]. Dynamic taint analysis involves marking data and tracing it at runtime to offer resistance against obfuscations with some overhead [9], [18]. Policy-based prevention involves checking contracts on DOM access at the browser or compiler level with zero overhead but requires wide-scale usage [10], [11], [19]. Graybox database-layer analysis involves instrumenting database protocols to detect context-dependent XSS that is not detectable by HTTP-layer analysis tools [26]. Machine learning techniques involve training classifiers on payload data; ML-taint hybrids deliver the best precision-recall tradeoff results [14].

III. LITERATURE REVIEW

A. Foundational Work on Taint-Based XSS Detection

1) Dynamic Taint Analysis for Exploit Detection [1]

The dynamic taint analysis at the binary level using TaintCheck, developed by Newsome and Song in their research published in 2005, had already identified user inputs as tainted and propagated them through registers and memory. The detection of automatic exploits had been achieved by checking for tainted data at instruction pointers and format string arguments. The fundamental operations introduced, propagated, checked, and sanitized by the research had been the intellectual basis for all the JavaScript taint tools reviewed in this paper.

2) Cross-Site Scripting Prevention with Dynamic Data Tainting [2]

In the domain of browser-embedded taint tracking, Vogt et al. were the first to propose the concept of taint tracking through their modification of the Firefox JavaScript engine. By using dynamic tracking through their modified JavaScript engine, as well as a static analysis, Vogt et al. identified 8.58% of more than one million web pages as having the potential for containing harmful information flows. This work provided a basis for the use of a modified JavaScript engine, taint-labelled data, and sink interception, which is used by most of the existing DOM-based XSS detection techniques.

3) 25 Million Flows Later [5]

The first large-scale empirical study on DOM-XSS using the taint-based approach by Lekies, Stock, and Johns used a taint-aware version of Chromium to crawl the top 5,000 websites according to Alexa, discovering 6,167 unique DOM-XSS vulnerabilities on 480 domains (9.6% of crawled websites). This proves that the taint-based detection approach is feasible at scale without relying on manual auditing, while providing an estimate of the real-world prevalence of DOM-XSS.

B. Dynamic Taint Tracking Systems

1) TT-XSS [7]

The first taint tracking framework for DOM-XSS, proposed by Wang et al., uses PhantomJS/WebKit, 10 sources, 9 sinks, and 21 transfer functions for JavaScript string operations. In addition, the automated exploit verification module is incorporated. The proposed framework, titled TT-XSS, successfully identified 17 out of 55 vulnerabilities and verified 5 of them using Google's Firing Range, outperforming AWVS 10.0, which could not verify any of the attacks. The only drawback of this framework is the use of the deprecated PhantomJS.

2) DOMsday [9]

In another study, Melicher et al. employed a modified Chromium V8 browser, which employed a byte-precise taint-tracking technique, and crawled the top 10,000 sites from Alexa, which resulted in 4.14 million data flows. In their Method B, they injected data from taint-tracked source origins, not from URL ends, and were able to discover 83% more confirmed vulnerabilities than other conventional techniques, resulting in 3,219 confirmed vulnerabilities. In addition, 82% of the identified vulnerabilities came from advertising and analytics iframes, and static analysis tools only identified 10% of what dynamic analysis tools identified.

3) Targeted Exploit Generation [13]

Bensalim et al. proposed Method C, which utilizes taint metadata to change only the tainted character range in the URL, while the rest of the application logic remains unchanged. Method C, which used the top 100,000 Tranco sites, reached a 45.82% exploit validation rate, which is almost twice the exploit validation rate of Method A and Method B, and validated 846 flows that Method A and Method B could not exploit, including flows through JSON-encoded parameters.

4) PanoptiChrome [18]

The first complete dynamic taint analysis framework was proposed by Kanyal and Sarangi for modern Chromium (version 117), extending V8's Ignition interpreter by approximately 7,000 lines to analyze both explicit and implicit information flows among 675 browser APIs. On 20,000 sites in the Tranco set, it detected 208 unknown fingerprinting APIs not detected by the state-of-the-art approach.

The co-location threshold filter improved false positives from 10.84% to 3.26%, but its approximately 16% page-load overhead and requirement for a maintained browser fork are limitations.

5) Augur [15]

Augur also addressed the key issue of taint propagation in JavaScript's async execution model using a unique taint tree ID per async function execution, storing its taint stack on suspension, and restoring it on resumption. Based on NodeProf/GraalVM, it correctly processes nested async/await calls, Promise resolutions, and .then() chains with a median overhead of 1.77x, outperforming Ichnaea's 5.92x on 17 out of 20 benchmarks. The drawback is that GraalVM is not used in web browsers, so DOM contexts cannot be addressed.

C. Static and Hybrid Analysis Approaches

1) Splendor [17]

Splendor was the first to propose a static model for the detection of stored XSS attacks in PHP applications using the DAL abstraction, which was completely ignored by the previous tools, such as RIPS. Fuzzy token matching is a technique that is used to extract database operation triples from the code property graphs. Two-phase taint analysis is a technique that stitches the write paths and read paths that share the same table-column pair. Splendor evaluated over 4,000 GitHub PHP projects and identified 17 zero-day CVEs, all of which were in DAL-based projects and were previously missed by RIPS and Black Widow.

2) Fluffy [16]

Fluffy leverages a combination of static taint analysis using CodeQL and unexpectedness classification by machine learning. The authors also evaluated four instantiations of ML, where Novelty Detection reported 0.95–0.98 F1 scores for command injection attacks using fewer than 10 seed names. Fluffy, a combination of static taint analysis and unexpectedness classification, successfully identified 117 (89.3%) of 131 confirmed CVEs for JavaScript, including CVE-2022-24785, a vulnerability in the moment.js library.

3) Joern Taint [20]

Effendi et al. proposed an incrementally updatable and scalable data dependence analysis system using Joern that supports Java, Python, and JavaScript. The over-approximated, unannotated external method behavior enables the analysis to proceed even in the absence of complete annotation coverage. The annotations provide precision at query time without requiring graph reconstruction. On Securibench Micro using Java, Joern achieved 0.871 F1 in 1.74 sec compared to CodeQL's 0.759 in 79.57 sec, using 0.29 GB of memory compared to CodeQL's 1.26 GB of memory.

D. Policy-Based and Prevention Approaches

1) API Hardening [11]

The Google project on API hardening improved JavaScript and TypeScript compilers to include compile-time restrictions on XSS-vulnerable DOM sink APIs, replacing them with type-safe alternatives (SafeHtml, SafeUrl, and SafeScript) that can only be created via specific channels. After its application to 2+ billion lines of Google code in 2018, there was a 90% reduction in DOM XSS bug bounty reports in major products in the following year. This approach directly influenced the W3C Trusted Types standard but requires monorepo adoption and cannot apply to third-party code.

2) PoliDOM [10]

PoliDOM extended the Chromium web browser's Blink renderer to include declarative per-element DOM Security Policies delivered over HTTP response headers. A DOM monitoring module catches all requests to modify the DOM below the JavaScript engine layer after de-obfuscation by the browser. This makes the system immune to encoding attacks. PoliDOM successfully blocked 55 out of 60 OWASP XSS test vectors in under 1 ms per operation with very few false positives, although five test vectors were able to evade the system by using string encoding in the URL.

3) TrustyMon [19]

TrustyMon adapted the W3C Trusted Types browser API for signature-based monitoring, not sanitization. During the extraction, Trusted Types callbacks catch all strings going to sinks, parse them to an AST, and compute signatures through preorder traversal, recording node types and built-in identifier values. At runtime, unknown injections cause violation reports. TrustyMon detected 100% of 19 CVE-mapped vulnerabilities and 2,181 injected attacks with just 27 ms of extra page load time without modifying the browser or server.

4) XSnare [12]

XSnare is a Firefox extension that stores a local database of application-specific XSS signatures derived from CVE data, which filters raw HTTP response strings prior to HTML parsing to evade parser rearrangement. Framework-specific probes detect the underlying CMS, load relevant signatures, and sanitize content using DOMPurify or character encoding. XSnare detected and patched 71 of 81 (93.4%) of the most recent 100 WordPress XSS CVEs, using 59 signatures, while overall CVE coverage reached 94.2% for four CMS platforms.

E. Graybox Database-Layer Detection

1) Steinhauser and Tůma [26]

Steinhauser and Tůma have identified that black-box scanners only address one of the four XSS variants, which are reflected/stored and context-insensitive/context-sensitive. The gray-box technique instruments the communication protocol of MariaDB/MySQL to capture the database fetch operations and inject malicious code, followed by a recursive parser of the browser context to determine the precise nesting context of HTML, CSS, JavaScript, and/or URI. The authors have tested their technique on eight open-source applications, and all of them have been successfully exploited with XSS. The three commercial scanners have identified four of the eight applications to be completely clean.

F. Neural and Learning-Based Taint Analysis

1) NEUTAINT [21]

NEUTAINT adopted end-to-end neural program embeddings and gradient-based saliency analysis, replacing the traditional rule-based taint propagation. A fully connected neural network, trained on 2,000 input/output pairs, served as a proxy for source-to-sink mapping. The result of this effort was 68% accuracy for hot-byte detection, compared to 58% for libdft, with 40 times lower

overhead and a 2.07% false-positive rate. Crucially, it was the only tool that completed libjpeg analysis within a reasonable timeframe, as all other tools took more than 24 hours, owing to taint explosion during JPEG decompression.

2) AirTaint [22]

AirTaint improved taint analysis from individual instructions to basic blocks, creating a concise taint behavior specification per block through hybrid emulation, followed by translation to x86 assembly using static binary rewriting. This resulted in the elimination of runtime context-switching overhead, achieving an average speedup of 509x over libdft and 216x over TaintRabbit while achieving identical taint results for 50 million verification tests. AirTaint successfully identified 14 CVEs from nine real-world application packages, verifying the effectiveness of granularity reduction.

3) Conflux [23]

Conflux addressed both under-tainting due to lack of control flow information and over-tainting due to standard control flow semantics in two novel ways: binding scopes that constrain propagation to branch edges on all paths of execution and a loop-relative stability heuristic that blocks taint explosions on branch-dependent loops. As a plugin to Phosphor, Conflux scored the best F1 on 43 out of 48 encoding benchmark test cases with just 2 false positives compared to 28 for the second-best method.

G. Machine Learning and Ensemble Approaches

1) Hybrid ML + Taint [14]

Melicher et al. used a DNN pre-filter, which predicted vulnerability using 32 million labeled JavaScript functions from a taint-tracking Chromium crawl. The DNN filtered out 97.5% of the functions while maintaining 94.5% recall of confirmed vulnerabilities, providing a 3.43x reduction in taint-tracking overhead. This research proved that, without taint confirmation, ML is not enough, with only 0.4% precision at 95% recall.

2) Ensemble ML / AlgoXSSF [33]

Nagarjun and Ahamad developed a well-balanced dataset of 154,626 labeled samples and compared six ensemble techniques, where the highest accuracy of 99.89% was obtained by the Histogram-based Gradient Boosting technique. All the samples were represented as fixed-size Unicode integer vectors and standardized using the StandardScaler. Although the model performs well in benchmark scenarios, it uses simple character-based features without the use of AST information, and the model is not tested in real-world scenarios.

3) XSS Detection via ML Algorithms [34]

Harshavardhan et al. performed a comparative evaluation of k-Nearest Neighbors, Logistic Regression, Random Forest, and Support Vector Machines for detecting XSS attacks on a mixed data set of actual attack samples and benign web data. Random Forest showed superior robustness to various attack vectors through ensemble learning, and SVMs performed best in detecting complex non-linear patterns in high-dimensional feature spaces. The results suggested that the choice of a classification algorithm should depend on the context and highlighted two important open problems: adversarial robustness and streaming.

H. Advanced Taint Analysis Foundations

1) Multi-Language DTA on GraalVM [24]

Kreindl, Bonetta, and Mössenböck proposed a three-layered GraalVM platform, which includes a language-independent core, language-specific extensions, and analysis-specific applications. The language-independent core manages shadow memory and taint labels using Truffle instrumentation. The proposed system has demonstrated cross-language taint propagation from JavaScript strings to a C function using LLVM IR. The base instrumentation cost is negligible, i.e., around 0.3% when the callback is empty. The major issues to be addressed are the granularity of labels for language types and the possibility of merging labels using compiler optimizations.

2) Galette [25]

Galette fixed the incompatibility of JVM taint tracking with recent Java versions (8, 11, 17, and 21) by hybridizing shadowing and mirroring, placing their runtime in java.base via jlink for JPMS compatibility. Galette generated 0 semantic failures and 0 propagation failures over 3,451 synthetic test programs. With a 7.4× overhead, Galette performs better than Phosphor (8.4×) and MirrorTaint (5,593×), setting a new baseline for compatible and efficient JVM taint analysis.

3) AutoLeak [27]

AutoLeak adopted a model for the browser DOM as a labeled directed multigraph and used a calculation of state differences to exhaustively enumerate cross-site leak methods without a predefined list of candidates. In a test of 151,776 cases for Chrome, Firefox, and Safari browsers, AutoLeak found five new XS-Leak classes and leaked on 20 of the 24 major Tranco top-50 websites. Although AutoLeak focuses on the exploitation of information side channels rather than direct exploitation of DOM XSS, the DOM graph difference technique provides a complementary method for identifying illegal structural changes.

IV. PROPOSED SYSTEM

Taintaru is a browser extension-based runtime taint-tracking framework designed specifically for modern JavaScript platforms without modifying the browser or server infrastructure. Taintaru instruments seven attacker-controllable source types: URL, hash, cookie, referrer, postMessage, localStorage, and sessionStorage. Eight attacker-controllable sink types: innerHTML, eval, document.write, setTimeout, src attributes, and href attributes are also instrumented in Taintaru, in addition to framework-specific escape hatch APIs: React's dangerouslySetInnerHTML, Angular's bypassSecurityTrustHtml, and Vue's v-html. Bridging instrumentation includes component lifecycle hooks and virtual DOM reconciliation in React, Vue, Angular, and Svelte—making Taintaru the first detection framework to specifically address these framework-specific patterns. The design goals include 98% detection recall, 2% false positives, and page-load overheads below 2%, significantly improving over PanoptiChrome's 16% page-load overhead.

V. COMPARATIVE ANALYSIS

Table 3 summarizes key characteristics of the fifteen reviewed tools alongside the proposed Taintaru system across five evaluation dimensions.

Tool / Reference	Year	Domain	Key Contribution	Primary Limitation
Newsome & Song — TaintCheck [1]	2005	Binary DTA	Foundational taint introduction /propagation/checking model; basis for all JS taint systems	Binary-level only; no direct JS applicability
Vogt et al. [2]	2007	Browser DTA	First Firefox engine modification for JS taint; detected harmful flows on 8.58% of 1M pages	Static co-analysis limited; Firefox-only engine patch
Lekies et al. [5]	2013	Empirical DOM-XSS	6,167 DOM-XSS vulnerabilities on Alexa top 5,000 (9.6%) via automated taint-aware Chromium	Research prototype; no persistent XSS coverage
TT-XSS [7]	2017	DOM-XSS Detection	First DOM-XSS framework: 10 sources, 9 sinks, 21 transfer functions, and auto-exploit verification	Relies on deprecated PhantomJS; no second-order input support
DOMsday [9]	2018	DOM-XSS Empirical	Byte-precise V8 tracking; method B yields 83% more confirmed vulns; 3,219 vulns in Alexa top 10K	High crawl overhead; homepage-only shallow coverage
PoliDOM [10]	2019	DOM-XSS Prevention	Declarative per-element DOM Security Policies in Blink; 91.7% OWASP coverage; <1 ms overhead/op	Requires a Chromium fork and manual policy authoring
API Hardening [11]	2020	Compile-Time Prevention	Compiler-enforced sink bans across 2B+ LOC, ~90% bug bounty reduction, and Trusted Types precursor	Requires org-wide monorepo adoption; no third-party script coverage
Bensalim et al. [13]	2021	Exploit Generation	Method C: 45.82% exploit rate (vs. 24.43% Method A); handles JSON-encoded URL params; 7,199 exploits	URL sources only; no cookie / localStorage /postMessage

Tool / Reference	Year	Domain	Key Contribution	Primary Limitation
Hybrid ML+Taint [14]	2021	ML + Dynamic Hybrid	DNN pre-filter: 97.5% function reduction; 94.5% recall retained; 3.43× overhead reduction	Standalone ML precision 0.4%; extreme class imbalance
Augur [15]	2022	Async JS DTA	Full ES7 async/await + Promise support via taint-tree; 1.77× median overhead; built on NodeProf	GraalVM is absent from browsers; the DOM context is unsupported
Steinhauser & Tůma [26]	2020	Graybox / Stored XSS	DB-layer interception finds all 4 XSS combos; exploits in all 8 tested apps vs. 0 by 3 commercial scanners.	Single-threaded; JS-embedded HTML XSS missed
Fluffy [16]	2023	Static + ML Hybrid	CodeQL + ML unexpectedness detection: 117/131 CVEs; discovers moment.js CVE-2022-24785	Path-traversal F1 low (0.29–0.76); Codex rate-limited
AutoLeak [27]	2023	XS-Leaks / DOM Side Channels	DOM modeled as LDMG; 5 novel XS-Leak classes; 20/24 Tranco top-50 sites affected	Targets side channels, not direct DOM-XSS exploitation
PanoptiChrome [18]	2024	In-Browser DTA	First modern Chromium-117 DTA; 675 APIs; 208 new fingerprinting APIs; GC-safe via Ephemeron refs	~16% page-load overhead; requires maintained fork
TrustyMon [19]	2025	DOM-XSS Detection	Trusted Types as AST-signature monitoring, 100% CVE detection, 27 ms overhead, no browser modification	Chromium-only; dynamic AST obfuscation may bypass
Taintaru (Proposed)	2026	DOM-XSS Detection	First framework-aware browser-extension taint tracker: React, Vue, Angular, Svelte; 7 sources, 8 sinks; <2% overhead target	Framework versions require maintenance; empirical validation is pending.

Table 3: Comparative Summary of DOM-XSS Detection Tools and the Proposed Taintaru System

A. *Cross-Cutting Observations*

1) *Detection rate versus overhead:* The best recall is achieved by dynamic taint tracking at the engine level (DOMsday [9] and PanoptiChrome [18]). This method, however, is associated with significant performance overhead and requires maintaining browser forks. Policy-based approaches (TrustyMon [19]: 27 ms; PoliDOM [10]: <1 ms/op) have negligible performance overhead, but browser patching or compiler integration is necessary. The hybrid ML+Taint method [14] reflects the best trade-off with respect to performance (94.5% recall, 3.43× overhead reduction).
2) *Framework coverage gap:* However, none of the fifteen tools reviewed provides instrumentation for React, Angular, Vue, or Svelte framework-specific DOM manipulation patterns or the encoding of escape hatch APIs in source sink specifications. This is where Taintaru aims to fill the gap.
3) *Stored and context-sensitive XSS:* The gray-box database layer approach [26] is the only technique reviewed for the combination of stored and context-sensitive XSS. None of the reviewed tools utilizes a database layer.

VI. RESEARCH GAPS

1) *Performance overhead:* The improvements of NEUTAINT 40× and AirTaint 509× for libdft [21], [22] have not been achieved for browser JavaScript. PanoptiChrome's page load overhead of 16% [18] persists as a barrier.
2) *Framework coverage:* None of the reviewed tools' instrumentation targets React's, Angular's, Vue's, or Svelte's lifecycle patterns capture framework-specific escape hatch APIs in source-sink specifications.

3) *Adversarial payloads:* The obfuscations generated using an LLM-boosted generation approach [32] have a structural complexity 28.1% higher than those generated using other tools, making them harder to detect using static signature-based ML and character-level ML detection techniques.
4) *Asynchronous SPA taint propagation:* Augur's taint tree approach [15] remains unimplemented in the context of single-page web application browsers where taint analysis traverses event loops, state stores, and Promise chains.
5) *Stored and context-sensitive XSS:* The gray-box approach to taint analysis targeting the database layer remains unimplemented in conjunction with client-side taint analysis; no tool covers all four XSS variant combinations.
6) *Evaluation benchmarks:* Currently, there is no benchmark that covers DOM XSS patterns in modern JavaScript frameworks with ground truth and asynchronous flows. While Securibench-micro.js [20] provides a foundation, it is not complete.
7) *Context-sensitive sanitization Verification:* None of the reviewed tools verify context matches sanitization, a major cause of DOM XSS in carefully crafted code [8].

VII. FUTURE RESEARCH DIRECTIONS

1) *Basic-block abstraction for JavaScript taint:* AirTaint's granularity reduction mechanism [22] should be adapted for JavaScript, i.e., defining function-body or async-task units as analysis primitives with precomputed taint behavior to achieve sub-2% page load overhead in production browsers.
2) *Framework-aware lifecycle modeling:* Taint analysis needs to be extended to encode framework-specific lifecycle events as taint boundaries, in addition to AutoLeak's DOM graph difference method [27] for detecting unauthorized structural changes during virtual DOM reconciliation.
3) *Neural taint analysis for JavaScript:* NEUTAINT's neural embedding mechanism should be trained on execution traces from taint-aware browsers [9], [18], which enables gradient-based saliency analysis as a lightweight browser-deployable taint engine.
4) *Integrated graybox and client-side detection:* Integrating database-layer interception mechanisms [26] with client-side taint tracking can offer end-to-end visibility from database write to DOM sink, addressing all four XSS variant combinations in a unified stack.
5) *Standardized DOM-XSS benchmark:* A benchmark for the community should include framework-specific vulnerabilities, flow-level ground truth, obfuscation types, and asynchronous SPA patterns, following securibench-micro.js [20].
6) *Adversarially robust detection:* LLM adversarial training [32] with TrustyMon AST value signatures [19] and NEUTAINT saliency analysis [21] would create detectors resilient to structural and semantic obfuscation.
7) *Conflux-inspired control-flow-aware sanitization:* Expanding Conflux's binding scope formalism [23] for JavaScript would allow context-sensitive verification of sanitization. A flow would be valid if a suitable sanitization function exists within the binding scope for every source-to-sink path with the appropriate injection context.

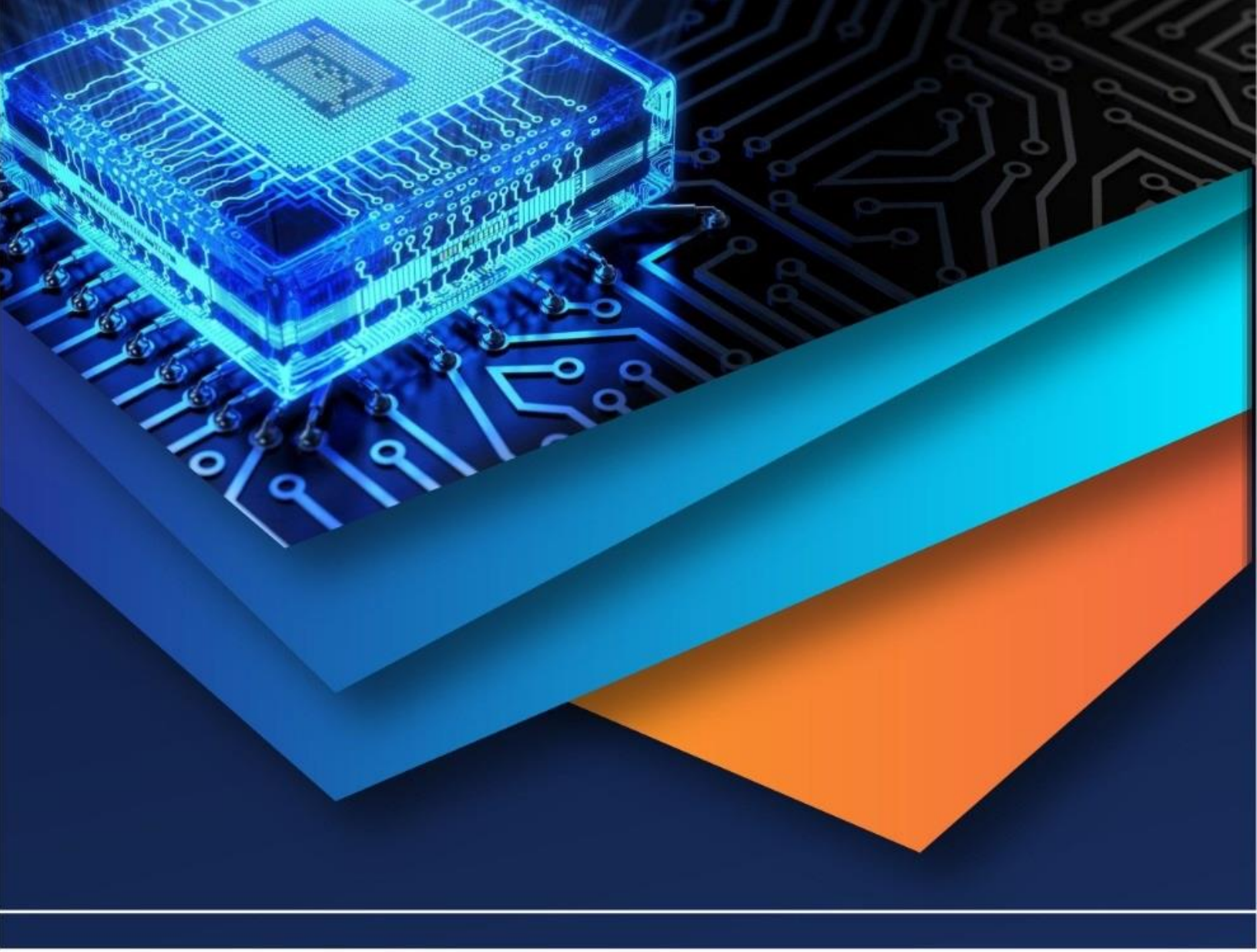
VIII. CONCLUSIONS

In addition, this review has sought to synthesize over thirty publications, from binary-level dynamic taint analysis to neural and machine learning-based approaches, in order to achieve a comparative evaluation of fifteen tools against the proposed Taintaru system. The major conclusion drawn is that none of the paradigms is individually effective; dynamic engine-level tracking is high-recall, high-overhead, policy-based approaches are low-cost but need universal acceptance; static approaches are scalable but circumvented by dynamic JavaScript; and the gray box database-based approach is effective against stored and context-dependent XSS. The combination of ML and taint is the best current balance for operation. Seven major gaps in the current state of the art are identified in terms of performance, framework coverage, adversarial tolerance, asynchronous propagation, integration of stored XSS, standardization of benchmarks, and context sensitivity. The proposed Taintaru system addresses some of these gaps through framework-aware instrumentation of browser extensions, and empirical evaluation is the first priority.

REFERENCES

[1] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in Proc. 12th Annu. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, Feb. 2005.
[2] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in Proc. 14th Annu. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, Feb. 2007.
[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in Proc. IEEE Symp. Security and Privacy (S&P), Oakland, CA, USA, May 2006.
[4] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in Proc. 17th Annu. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, Mar. 2010.

[5] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Berlin, Germany, Nov. 2013, pp. 1193–1204.
[6] I. Parameshwaran et al., "DexterJS: Robust testing platform for DOM-based XSS vulnerabilities," in Proc. 10th Joint Meeting on Foundations of Software Engineering (FSE), Bergamo, Italy, Sep. 2015.
[7] R. Wang, G. Xu, X. Zeng, X. Li, and Z. Feng, "TT-XSS: A novel taint tracking based dynamic detection framework for DOM cross-site scripting," J. Parallel Distrib. Comput., 2017, doi: 10.1016/j.jpdc.2017.07.006.
[8] J. Weinberger, P. Saxena et al., "A systematic analysis of XSS sanitization in web application frameworks," in Proc. 16th European Symp. Research in Computer Security (ESORICS), Leuven, Belgium, Sep. 2011, pp. 150–171, doi: 10.1007/978-3-642-23822-2_9.
[9] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting," in Proc. 25th Annu. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, Feb. 2018.
[10] J. Iqbal, R. Kaur, and N. Stakhanova, "PoliDOM: Mitigation of DOM-XSS by detection and prevention of unauthorized DOM tampering," in Proc. 14th Int. Conf. Availability, Reliability and Security (ARES), Canterbury, UK, Aug. 2019.
[11] P. Wang, J. Bangert, and C. Kern, "If it's not secure, it should not compile: Preventing DOM-based XSS in large-scale web development with API hardening," in Proc. IEEE Symp. Security and Privacy (S&P), San Francisco, CA, USA, May 2020.
[12] J. C. Pazos, J.-S. Légaré, I. Beschastnikh, and W. Aiello, "Precise XSS detection and mitigation with client-side templates (XSnare)," in Proc. 9th ACM Conf. Data and Application Security and Privacy (CODASPY), Dallas, TX, USA, Mar. 2019.
[13] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking about my generation: Targeted DOM-based XSS exploit generation using dynamic data flow analysis," in Proc. 14th European Workshop on Systems Security (EuroSec), Online, Apr. 2021.
[14] W. Melicher, C. Fung, L. Bauer, and L. Jia, "Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning," in Proc. Web Conf. (WWW), Ljubljana, Slovenia, Apr. 2021.
[15] M. W. Aldrich, A. Turcotte, M. Blanco, and F. Tip, "Augur: Dynamic taint analysis for asynchronous JavaScript," in Proc. 37th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), Michigan, USA, Oct. 2022.
[16] Y. W. Chow, M. Schäfer, and M. Pradel, "Beware of the unexpected: Bimodal taint analysis," in Proc. 32nd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), Seattle, WA, USA, Jul. 2023.
[17] H. Su, F. Li, L. Xu et al., "Splendor: Static detection of stored XSS in modern web applications," in Proc. 32nd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), Seattle, WA, USA, Jul. 2023.
[18] R. Kanyal and S. R. Sarangi, "PanoptiChrome: A modern in-browser taint analysis framework," in Proc. ACM Web Conf. (WWW), Singapore, May 2024.
[19] S. Park, J. Kim, S. Keum, H. Lee, and S. Son, "TrustyMon: Practical detection of DOM-based XSS attacks using trusted types," in Proc. ACM Asia Conf. Computer and Communications Security (ASIA CCS), Taipei, Taiwan, Apr. 2025.
[20] S. D. B. Effendi, X. Pinho, A. M. Dreyer, and F. Yamaguchi, "Scalable language agnostic taint tracking using explicit data dependencies," in Proc. 14th ACM SIGPLAN Int. Workshop on the State Of the Art in Program Analysis (SOAP), 2025.
[21] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "NEUTAINT: Efficient dynamic taint analysis with neural networks," in Proc. IEEE Symp. Security and Privacy (S&P), San Francisco, CA, USA, May 2020.
[22] Q. Sang, Y. Wang, Y. Liu, X. Jia, T. Bao, and P. Su, "AirTaint: Making dynamic taint analysis faster and easier," in Proc. IEEE Symp. Security and Privacy (S&P), San Francisco, CA, USA, May 2024.
[23] K. Hough and J. Bell, "A practical approach for dynamic taint tracking with control-flow relationships," ACM Trans. Softw. Eng. Methodol. (TOSEM), vol. 31, no. 2, 2021.
[24] J. Kreindl, D. Bonetta, and H. Mössenböck, "Towards efficient, multi-language dynamic taint analysis," in Proc. 16th ACM SIGPLAN Int. Conf. Managed Programming Languages and Runtimes (MPLR), Athens, Greece, Oct. 2019.
[25] K. Hough and J. Bell, "Dynamic taint tracking for modern Java virtual machines," Proc. ACM Softw. Eng., vol. 2, 2025.
[26] A. Steinhauser and P. Tůma, "Database traffic interception for graybox detection of stored and context-sensitive XSS," Charles University, Prague, Tech. Rep, 2020.
[27] D. T. Noß, L. Knittel, C. Mainka, M. Niemietz, and J. Schwenk, "Finding all cross-site needles in the DOM stack: A comprehensive methodology for the automatic XSS detection in complex web applications," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Copenhagen, Denmark, Nov. 2023.
[28] J. Kaur, U. Garg, and G. Bathla, "Detection of cross-site scripting (XSS) attacks using machine learning techniques: A review," Artif. Intell. Rev., 2023, doi: 10.1007/s10462-023-10433-3.
[29] I. K. Thajeel et al., "Machine and deep learning-based XSS detection approaches: A systematic literature review," J. King Saud Univ. – Comput. Inf. Sci., 2023, doi: 10.1016/j.jksuci.2023.101628.
[30] Z. Liu et al., "MFXSS: An effective XSS vulnerability detection method based on multi-features of JavaScript programs," Comput. Secur., 2023, doi: 10.1016/j.cose.2022.102890.
[31] M. Al-Kasassbeh et al., "An efficient artificial intelligence approach for early detection of cross-site scripting attacks," Results Eng., 2024, doi: 10.1016/j.rineng.2024.102134.
[32] A. R. Ibrahimzada et al., "Leveraging large language models to strengthen machine learning-based cross-site scripting detection," arXiv:2504.21045, Apr. 2025.
[33] P. M. D. Nagarjun and S. S. Ahamad, "Ensemble methods to detect XSS attacks," Int. J. Adv. Comput. Sci. Appl. (IJACSA), vol. 11, no. 5, 2020.
[34] G. Harshavardhan et al., "XSS attack detection using machine learning algorithms," Int. J. Sci. Res. Eng. Manage. (IJSREM), vol. 7, no. 12, Dec. 2023, doi: 10.55041/IJSREM27487.
[35] OWASP Foundation, "OWASP Top 10 Web Application Security Risks," 2021. [Online]. Available: https://owasp.org/Top10/



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY