information

MDPI

Article

Context-Aware Hybrid Detection of DOM-Based Cross-Site Scripting via Runtime Semantic Modeling

Maksim Iavich [1], Daviti Botchorishvili [2] and Audrius Lopata [3,*]

[1] Department of Computer Science, Caucasus University, Tbilisi 0102, Georgia; miavich@cu.edu.ge
[2] Faculty of Computer Science, Sokhumi State University, Tbilisi 0102, Georgia; davitibotchorishvili@sou.edu.ge
[3] Faculty of Informatics, Kaunas University of Technology, 44249 Kaunas, Lithuania
* Correspondence: audrius.lopata@ktu.lt

Abstract

DOM-based Cross-Site Scripting (DOM XSS) remains a critical web application vulnerability due to its exclusive manifestation within client-side execution contexts, rendering traditional server-side defenses ineffective. Existing machine learning approaches achieve high recall but suffer from critically low precision in DOM-specific scenarios. Payload-centric classifiers frequently misclassify syntactically suspicious but semantically benign injections, causing high false positive rates. This paper introduces a context-aware hybrid detection framework integrating dynamic taint tracking with runtime DOM semantic analysis and lightweight machine learning classification. The proposed architecture extracts a 42-dimensional feature vector combining 22 lexical payload features with 20 contextual execution features capturing sink semantics, element type, attribute execution capability, and DOM state properties. A Random Forest classifier is employed to enable low-latency inference and demonstrates potential for real-time deployment. By modeling exploitability as a function of execution context rather than payload syntax alone, the framework significantly reduces false positives while maintaining high detection capability. Experimental evaluation demonstrates that contextual feature integration substantially improves precision compared to payload-only baselines, achieving a favorable precision-efficiency trade-off. The primary contribution lies in elevating runtime semantic context to a first-class feature space for DOM XSS detection, representing a shift from text-centric classification toward execution-aware security modeling in client-side web environments.

Keywords: DOM-based XSS; machine learning; random forest; taint tracking; web security; browser security; context-aware detection

Check for updates

Academic Editor: Marco Rospocher

Received: 4 March 2026
Revised: 30 March 2026
Accepted: 1 April 2026
Published: 3 April 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The architectural evolution of web applications over the past decade has fundamentally altered the security landscape of client–server systems. Modern web platforms increasingly rely on client-side execution models, including Single Page Applications (SPAs), Progressive Web Apps (PWAs), and decentralized Web 3.0 architectures. In these environments, substantial application logic is executed within the browser through dynamic manipulation of the Document Object Model (DOM). While this shift enables rich user interactivity and reduced server load, it simultaneously expands the attack surface in ways that traditional server-centric security mechanisms are ill-equipped to address [1].

Among client-side vulnerabilities, DOM-based Cross-Site Scripting (DOM XSS) represents a particularly insidious class of injection attacks. Unlike stored or reflected XSS,

DOM XSS exploits do not require malicious payloads to traverse the server infrastructure. Instead, the attack is executed entirely within the victim's browser through unsafe client-side JavaScript operations that read attacker-controlled input (e.g., location.hash, document.URL) and write it into execution-sensitive sinks (e.g., innerHTML, eval(), setTimeout()). Because the malicious payload never appears in HTTP responses, server-side defenses—including Web Application Firewalls (WAFs) and static code scanners—remain blind to such attacks.

The severity of web injection attacks is well documented. The Verizon 2023 Data Breach Investigations Report attributes approximately 26% of breaches to web application attack vectors, while IBM reports that the average global cost of a data breach has reached $4.45 million. Industry bug bounty data further indicates that DOM XSS vulnerabilities command high monetary rewards, reflecting both their exploitability and detection difficulty [2,3].

Despite extensive research on XSS detection, DOM-specific detection remains comparatively underdeveloped. Large-scale empirical research demonstrates the limitations of current approaches [4,5]. Melicher et al. (2021) [6] processed over 18 billion JavaScript functions using hybrid taint tracking and deep learning. While achieving high recall (95%), their precision for DOM XSS detection was only 26.7%, implying that nearly three out of four flagged cases were false positives. Such performance characteristics render direct deployment impractical due to alert fatigue and trust degradation [6,7].

A fundamental reason for this performance gap lies in context insensitivity. Most machine learning approaches treat JavaScript payloads as isolated textual artifacts, ignoring execution semantics within the DOM environment. However, exploitability in DOM XSS is not determined solely by payload syntax [8]. It depends critically on:

- The specific sink receiving the data;
- The DOM element type involved;
- The presence of executable attributes (e.g., onerror);
- The broader runtime state of the DOM.

Without contextual awareness, classifiers cannot distinguish between syntactically similar but semantically distinct operations.

1.1. Research Objective

This work addresses the following research question:

How can we design a lightweight, client-side, real-time detection framework for DOM-based XSS that integrates runtime program analysis with machine learning to achieve high precision while maintaining deploy ability? Unlike prior hybrid approaches that combine taint tracking with either rule-based filtering or context-insensitive machine learning, this work introduces explicit runtime semantic contextualization as a structured feature space, directly modeling DOM execution constraints to reduce false positives.

1.2. Contributions

We propose a context-aware hybrid detection architecture that combines dynamic taint tracking with a DOM Context Analyzer and a computationally efficient Random Forest classifier.

The key contributions are:

- A runtime component that systematically extracts 20 execution-context features capturing sink semantics, element type, attribute execution capability, and DOM state properties. This operationalizes the insight that exploitability depends on execution context rather than payload syntax alone.

- A 42-dimensional feature vector integrating lexical payload features with structured contextual DOM features, enabling classifiers to model semantic execution feasibility.
- An empirical demonstration that incorporating contextual execution features significantly reduces false positives in DOM-based XSS detection, achieving a 26.9 percentage-point improvement in precision over payload-only baselines.

In addition to these methodological contributions, the framework is implemented with a focus on low-latency, browser-compatible deployment, enabling real-time analysis under practical constraints. This paper is organized as follows. Section 2 examines existing XSS detection approaches, encompassing hybrid program analysis, classical machine learning, and deep learning techniques, and establishes the research gaps that motivate the present work. Section 3 defines the threat model and provides a formal characterization of DOM XSS exploitability in terms of execution context properties. Section 4 details the architecture of the proposed detection framework, comprising the dynamic taint tracking module, DOM Context Analyzer, feature extraction pipeline, and Random Forest classifier. Section 5 outlines the experimental methodology, including dataset construction, baseline selection, and performance assessment. Section 6 interprets the experimental findings, addresses deployment considerations, and acknowledges the limitations of the approach. Section 7 summarizes the conclusions and identifies directions for future investigation.

2. Related Work

2.1. Evolution of XSS Detection Methodologies

Cross-Site Scripting detection research has evolved through three primary methodological phases: rule-based filtering, program analysis techniques, and machine learning-driven approaches [9]. While early work focused primarily on server-side validation and output encoding strategies, the architectural shift toward client-heavy web applications has exposed fundamental limitations in server-centric detection paradigms.

DOM-based XSS, first systematically examined at scale by Lekies et al. (2013), demonstrated that a significant portion of vulnerabilities arise exclusively within client-side JavaScript execution contexts [10]. Their large-scale dynamic taint-tracking study analyzed approximately 25 million data flows and revealed that DOM XSS vulnerabilities were both prevalent and difficult to identify through static analysis alone. However, their approach required manual validation of candidate flows, limiting scalability.

Subsequent research introduced automated classification mechanisms layered atop taint tracking to address this limitation.

2.2. Hybrid Program Analysis and Machine Learning

Melicher et al. (2021) [6] represent the most comprehensive attempt to automate DOM XSS detection through hybridization. Their system combined dynamic taint tracking with a deep neural network classifier trained on data flow representations extracted from JavaScript execution [6].

While their approach achieved high recall (95%), precision for DOM XSS detection was only 26.7%. This discrepancy is significant: high recall ensures vulnerability coverage, but low precision renders the system operationally impractical due to excessive false positives.

Error analysis in their study identified a core issue: flows that appeared syntactically suspicious were often semantically benign due to execution context constraints. Their neural model operated primarily on structural representations of data flow graphs but lacked fine-grained DOM execution semantics.

This finding suggests that the limiting factor in DOM XSS detection is not merely feature learning capacity, but contextual modeling of runtime semantics.

2.3. Classical Machine Learning for XSS Detection

Classical supervised learning approaches—including Random Forest, Support Vector Machines, Naïve Bayes, and Logistic Regression—have demonstrated strong performance in general XSS detection tasks [11–13].

Alhamyani and Alshammari (2024) reported 99.78% accuracy using Random Forest on a balanced dataset of 200,000 samples [14]. Feature representations were primarily lexical, including character n-grams, token frequencies, and TF-IDF vectors [14].

However, these results must be interpreted cautiously:

- The dataset did not isolate DOM XSS cases.
- Evaluation occurred under laboratory conditions.
- Feature extraction ignored execution context.
- Real-world deployment scenarios were not evaluated.

Systematic literature reviews confirm this pattern. Kaur et al. (2023) and Thajeel et al. (2023) both observe that most ML-based XSS detection studies rely on static payload representations without DOM-aware semantic analysis [9].

Thus, high reported accuracy does not necessarily translate to effective DOM-specific detection.

2.4. Deep Learning Approaches

Deep learning models—including CNNs, LSTMs, GRUs, and transformer architectures—have been applied to XSS detection under the premise that hierarchical representations capture latent structural patterns in malicious payloads [15–17].

While deep architectures improve recall and generalization to obfuscated payloads, several practical limitations emerge:

- High computational overhead;
- Latency incompatible with client-side real-time enforcement;
- Limited interpretability;
- Continued context insensitivity.

Even transformer-based models treat inputs as token sequences rather than execution-bound semantic events within a runtime DOM [18].

Consequently, although deep learning advances pattern recognition, it does not inherently resolve the semantic-context gap.

2.5. Persistent Research Gaps

A synthesis of the literature reveals six structural deficiencies in current DOM XSS detection research:

Gap 1—Context Insensitivity

Most approaches evaluate payloads as isolated textual artifacts without modeling execution semantics within the DOM environment.

Gap 2—Precision Deficiency in DOM Settings

DOM-specific evaluations consistently report severe precision degradation compared to stored/reflected XSS detection.

Gap 3—Laboratory Evaluation Bias

Performance is typically reported on static, pre-labeled datasets rather than live web execution environments.

Gap 4—Computational Incompatibility with Browser Deployment

Deep learning architectures introduce latency incompatible with pre-execution blocking.

Gap 5—Lack of Runtime Semantic Features

Existing feature engineering primarily captures lexical and syntactic signals, not execution-context characteristics.

Gap 6—Limited Explainability

Black-box models reduce operational trust and hinder security auditability [19,20].

2.6. Positioning of the Present Work

Existing hybrid DOM XSS detection frameworks typically combine taint tracking with rule-based filtering or context-insensitive machine learning classifiers. In contrast, the proposed approach differentiates itself by explicitly modeling runtime DOM semantics as a structured feature space integrated directly into the classification pipeline. Specifically, prior systems either (i) rely solely on data-flow reachability without semantic interpretation (Lekies et al. [10]) or (ii) apply machine learning models to abstract flow representations without modeling DOM execution constraints. In contrast, this work operationalizes execution context through a formally defined set of DOM semantic features (sink type, element type, attribute execution capability, and DOM state) and demonstrates their measurable impact on precision. This represents a methodological refinement that emphasizes execution-aware feature modeling within hybrid detection frameworks and directly addresses the precision limitations identified in prior work. The present work addresses these gaps by introducing runtime semantic contextualization as an explicit feature space. Rather than relying solely on lexical patterns or abstract data flow graphs, we extract structured DOM execution features capturing:

- Sink semantics;
- Element type;
- Executable attribute presence;
- DOM state relationships;
- Structural depth.

This design enables the classifier to differentiate between syntactically similar injections that differ in exploitability due to execution context.

In contrast to deep learning-heavy approaches, we adopt a lightweight ensemble classifier to ensure deploy ability, interpretability, and real-time responsiveness.

Our contribution therefore lies not in increasing model complexity, but in elevating feature semantic richness while maintaining computational tractability.

3. Threat Model and Formalization of DOM-Based XSS

3.1. System Model

We consider a modern web application executing within a standards-compliant browser environment. The application consists of:

- Client-side JavaScript code;
- A dynamic Document Object Model (DOM);
- Event-driven execution semantics;
- Standard browser security mechanisms (Same-Origin Policy, CSP if present).

The server delivers static and dynamic content, but the vulnerability of interest manifests exclusively in client-side execution logic.

We assume the attacker cannot modify server-side code directly but can influence client-side execution through attacker-controlled inputs accessible within the browser runtime.

3.2. Attacker Capabilities

We assume an adversary capable of:

1. Crafting malicious URLs containing payloads in:
   - location.hash;
   - location.search;
   - document.URL.
2. Injecting data into:
   - window.name;
   - postMessage;
   - Local/session storage (if prior injection occurred).
3. Delivering crafted URLs to victims via:
   - Phishing;
   - Social engineering;
   - Redirect chains;
   - Third-party embedding.

We do not assume:

- Direct compromise of the server;
- Bypass of browser Same-Origin Policy;
- Privileged browser extension access.

The attacker's objective is to cause arbitrary JavaScript execution within the victim's browser context.

3.3. Formal Definition of DOM XSS

Let:

- S denote a set of attacker-controlled sources;
- K denote a set of execution-sensitive sinks;
- P denote a payload string;
- D denote the DOM state at runtime.

A DOM XSS vulnerability exists if and only if:

$$\exists s \in S, \exists k \in K : Flow(s,k) \wedge Executable(P,D,k) \tag{1}$$

where

- $Flow(s,k)$ indicates a data flow from source to sink;
- $Executable(P,D,k)$ indicates that payload P results in script execution under DOM state D when written into sink k.

This formalization highlights a key insight:

Exploitability is not determined solely by payload structure, but by the interaction between payload, sink semantics, and runtime DOM state.

3.4. Source and Sink Sets

We define the primary source set:

$$\begin{aligned} S \quad = \{&location.hash, location.search, document.URL, \\ &window.name, postMessage, localStorage, sessionStorage\} \end{aligned} \tag{2}$$

And the sensitive sink set:

$$\begin{aligned} K \quad = \{&innerHTML, outerHTML, document.write, eval, \\ &setTimeout, setInterval, Function, href, src, eventHandlers\} \end{aligned} \tag{3}$$

However, exploitability differs across sinks:

- innerHTML parses HTML;
- eval() directly executes JavaScript;
- href executes only under javascript: scheme;
- textContent does not execute code.

Therefore:

$$Executable(P, D, k) \neq f(P) \text{ alone} \tag{4}$$

It is instead:

$$Executable(P, D, k) = g(P, k, E, A, C) \tag{5}$$

where:

- E = Element type;
- A = Attribute presence;
- C = Contextual DOM state.

This motivates contextual feature modeling.

3.5. Context Sensitivity Principle

Consider two assignments:
element.innerHTML = "<img src=x onerror=alert(1)>";
element.textContent = "<img src=x onerror=alert(1)>".
Lexically identical payload. Semantically different execution outcome.
Formally:

$$f_{lexical}(P_1) = f_{lexical}(P_2) \tag{6}$$

But:

$$Executable(P_1, D, innerHTML) = 1 \tag{7}$$

$$Executable(P_2, D, textContent) = 0 \tag{8}$$

This illustrates:
Lexical equivalence does not imply semantic equivalence.
Most prior ML models approximate:

$$Class(P) = h(f_{lexical}(P)) \tag{9}$$

Our approach instead models:

$$Class(P, D) = h(f_{lexical}(P), f_{context}(D)) \tag{10}$$

This shift from payload-centric classification to execution-context-aware classification constitutes the theoretical foundation of our framework.

3.6. Defense Objectives

The detection system must satisfy:

- **Pre-execution interception**

  Malicious flows must be blocked before execution.

- **Low false positive rate**

  Excessive blocking degrades usability.

- **Browser-compatible latency (<10 MS)**

  Detection must not introduce perceptible delay.

- **Explainability**

Security decisions must be interpretable.

3.7. Assumptions and Limitations

We assume:

- Browser APIs can be instrumented.
- Taint tracking accurately identifies source-to-sink flows.
- The classifier receives correct contextual features.

We do not address:

- Side-channel attacks;
- CSP bypass techniques;
- Browser engine vulnerabilities.

4. Proposed Method: Context-Aware Hybrid DOM XSS Detection Framework

4.1. Architectural Overview

We propose a hybrid runtime detection framework integrating dynamic taint analysis with context-aware machine learning classification. The system operates entirely within the browser environment and intercepts DOM write operations prior to execution. The framework is implemented as a Chrome extension that intercepts 20 DOM sink APIs through JavaScript hooking, extracts the 42-dimensional feature vector at runtime, and applies the Random Forest classifier prior to sink execution.

The architecture consists of four primary components:

- Dynamic Taint Tracker;
- DOM Context Analyzer;
- Feature Extraction Engine;
- Random Forest Classification Module;

The overall system architecture is illustrated in Figure 1.

Let:

- P denote the payload string;
- D denote the runtime DOM state;
- $F_{lex}(P) \in \mathbb{R}^{22}$ denote lexical feature vector;
- $F_{ctx}(D) \in \mathbb{R}^{20}$ denote contextual feature vector.

The final feature representation is:

$$F = [F_{lex}(P) \parallel F_{ctx}(D)] \in \mathbb{R}^{42} \tag{11}$$

Classification decision:

$$\hat{y} = RF(F) \tag{12}$$

where RF denotes the trained Random Forest classifier.

Figure 1. Context-aware DOM-XSS detection framework architecture.

Overall architecture of the proposed context-aware DOM-XSS detection framework. From left to right: (1) the dynamic taint tracking module intercepts browser API calls and records source-to-sink propagation events; (2) the DOM Context Analyzer extracts structural and semantic context features from the active DOM state; (3) the feature extraction pipeline constructs a 42-dimensional feature vector combining lexical and contextual features; (4) the Random Forest classifier produces a binary malicious/benign classification with confidence score.

Contextual features are captured at runtime by inspecting the DOM element and execution environment. When a sink is intercepted, the extension examines:

- Sink type: Identified from the intercepted API call (e.g., innerHTML, eval);
- Element type: Obtained from element.tagName for property setters;
- Attribute presence: Checked via element.hasAttribute() for onload, onerror, href, src;
- DOM state: Determined by traversing the DOM tree to check parent taint status, presence of event listeners, and current script context.

4.2. Dynamic Taint Tracking Module

The taint tracking component monitors propagation of attacker-controllable inputs from predefined source APIs to execution-sensitive sinks.

The module instruments 20 critical DOM sinks across seven categories: DOM Properties (innerHTML, outerHTML, innerText), Document Methods (document.write, document.writeln), Code Execution (eval, setTimeout, setInterval, Function), URL Sources (href, src, location, iframe src), Attribute Methods (setAttribute for event handlers), DOM Methods (insertAdjacentHTML, createContextualFragment, DOMParser), and Other (nodeValue, textContent, replaceChild). Taint tracking is implemented by overriding native JavaScript functions and property setters. For each sink, the extension stores a reference to the original implementation and replaces it with a wrapper function. When the sink is invoked, the wrapper extracts the payload, identifies the sink type, and records the flow before passing control to the original function.

Formally, a taint propagation event is recorded when:

$$Flow(s, k) = 1 \tag{13}$$

where:

- $s \in S$ (source set);
- $k \in K$ (sink set).

The module performs:

- Source interception;
- Sink instrumentation;
- Payload extraction;
- Sink-type recording.

The module performs lightweight operations, adding approximately 0.12 ms per sink invocation. This contributes to the overall 0.72 ms inference latency measured in our experiments.

The taint tracker does not perform vulnerability classification. Its purpose is candidate flow detection. This separation ensures modularity and reduces classification overhead.

The operational logic of the taint tracking module is illustrated in Figure 2.

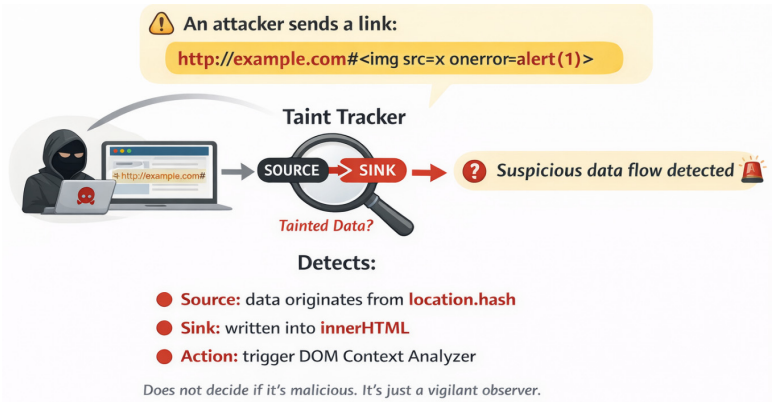

Figure 2. Operation of the dynamic taint tracking module.

4.3. DOM Context Analyzer

4.3.1. Motivation

Exploitability depends on execution semantics rather than payload structure alone. Therefore, runtime interrogation of the DOM environment is required.

4.3.2. Contextual Feature Space

We define contextual feature vector:

$$F_{ctx}(D) = [C_{sink}, C_{element}, C_{attribute}, C_{state}] \tag{14}$$

where:

- **Sink Context (7 binary features)**

  Indicates which sink API receives the tainted data.

$$C_{sink} = \left[sink_{innerHTML}, sink_{outerHTML}, sink_{write}, sink_{eval}, sink_{href}, sink_{src}, sink_{script\_src}\right] \tag{15}$$

- **Element Type (5 binary features)**

$$C_{element} = [is\_div, is\_script, is\_iframe, is\_img, is\_a] \tag{16}$$

- **Attribute Presence (4 binary features)**

$$C_{attribute} = [has\_onload, has\_onerror, has\_href, has\_src] \tag{17}$$

- **DOM State (4 features)**

$$C_{state} = [parent\_tainted, has\_event\_listener, in\_script\_context, dom\_depth] \tag{18}$$

The contextual vector dimension:

$$\mid F_{ctx} \mid = 20 \tag{19}$$

These features capture semantic execution feasibility.

4.4. Lexical Feature Extraction

We extract 22 lexical features from payload string P.

$$F_{lex}(P) = [length, binary\_tag\_flags, binary\_event\_flags, function\_presence, character\_counts, entropy] \tag{20}$$

Length is computed by simply counting the number of characters in the payload string using JavaScript's length property. The binary tag flags are computed by scanning the payload string for the presence of specific substrings: '<script' for the script tag, '<img' for the img tag, and so on. If the substring is found, the corresponding flag is set to 1;

otherwise, it remains 0. Character counts are obtained by iterating through the payload and incrementing counters each time a special character (<, >, “, ‘, =, (,), /) is encountered.

Entropy is computed as:

$$H(P) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i) \tag{21}$$

where:

- x_i represents unique characters;
- $p(x_i)$ their empirical frequency.

Lexical features capture syntactic risk indicators but do not determine exploitability alone. The extraction process is lightweight, adding minimal overhead to the overall classification pipeline.

4.5. Feature Concatenation

Final feature vector:

$$F \in \mathbb{R}^{42} \tag{22}$$

Represents:

- What is injected;
- Where it is injected;
- How it is injected;
- Under what DOM conditions.

This unified representation models semantic execution probability rather than textual similarity.

4.6. Random Forest Classifier

4.6.1. Rationale for Selection

We select Random Forest for the following reasons:

1. Non-linear decision boundaries;
2. Robustness to overfitting via bagging;
3. Interpretability through feature importance;
4. Low inference latency;
5. Suitability for tabular feature space.

Unlike deep neural networks, Random Forest provides:

- Deterministic inference time;
- No gradient computation overhead;
- Transparent feature attribution [21].

4.6.2. Model Configuration

Let T denote number of trees. We configure the model as follows; the selected hyperparameter values are summarized in Table 1.

We configure:

Table 1. Random Forest Hyperparameter Configuration.

Parameter	Value
(T)	100
max_depth	15
min_samples_split	5
min_samples_leaf	2

Node impurity computed via Gini index:

$$Gini(t) = 1 - \sum_{i=1}^{2} p(i \mid t)^2 \tag{23}$$

Decision aggregation:

$$\widehat{y} = \frac{1}{T}\sum_{j=1}^{T} Tree_j(F) \tag{24}$$

Probability output:

$$P_{malicious} = RF(F) \tag{25}$$

4.7. Decision Policy

We implement a three-tier response model:

$$\begin{aligned} &\text{BLOCK if } P \geq 0.8 \\ &\text{SANDBOX if } 0.5 \leq P < 0.8 \\ &\text{ALLOW if } P < 0.5 \end{aligned}$$

This probabilistic thresholding enables precision prioritization. The thresholds were selected empirically based on validation set performance. A threshold of 0.8 for blocking ensures that only high-confidence malicious predictions trigger intervention, prioritizing precision over recall. The intermediate threshold of 0.5 captures suspicious flows for sandboxing without disrupting normal browsing behavior.

4.8. Computational Complexity

Let:

- $d = 42$ feature dimension;
- $T = 100$ trees;
- $h \leq 15$ tree depth.

Inference complexity:

$$O(T \cdot h) \tag{26}$$

Since:

$$T \cdot h = 100 \times 15 = 1500 \tag{27}$$

Inference cost is constant and bounded.
Empirical latency target: < 10 ms
This satisfies browser real-time constraints.

4.9. Design Advantages

Compared to payload-only models:

$$Class_{ours}(P, D) = h(F_{lex}(P), F_{ctx}(D)) \tag{28}$$

Compared to deep neural approaches:

- Lower latency;
- Higher interpretability;
- Reduced computational footprint;
- Improved deployment feasibility.

Compared to taint-only approaches:

- Automatic classification;
- Reduced manual inspection burden;
- Lower false positive rate.

5. Experimental Evaluation

5.1. Experimental Objectives

The experimental evaluation is designed to answer the following research questions:

RQ1: Does contextual feature integration significantly improve precision for DOM-based XSS detection?
RQ2: Does the hybrid model outperform payload-only baselines?
RQ3: Is the system computationally viable for real-time browser deployment?
RQ4: Does the model maintain robustness against obfuscated payloads?

5.2. Dataset Construction

5.2.1. Data Sources

The dataset consists of both benign and malicious DOM flows collected from:

Malicious payloads were adapted from some publicly available sources: the PayloadsAllTheThings XSS repository, the OWASP XSS Filter Evasion Cheat Sheet, the PortSwigger Web Security XSS payload library, and publicly disclosed DOM XSS bug bounty reports from the HackerOne disclosure database. XSS payloads were also collected from publicly available repositories such as XSS.js.org. Payloads were manually reviewed and adapted to DOM-specific sink contexts. Additional variants were generated through five transformation rules: case randomization, HTML entity substitution, JavaScript string concatenation, String.fromCharCode() encoding, and nested encoding. Benign samples were manually constructed as source-to-sink flow instances carrying non-malicious content. All benign samples were labeled by structural rule: a flow instance was labeled benign if and only if the payload contained no executable JavaScript expression and the sink context rendered execution semantically impossible. Malicious samples were labeled positive if the payload contained at least one executable JavaScript expression and the sink context permitted execution. Approximately 65% of the dataset is derived from real-world payload sources and bug bounty disclosures, while 35% consists of synthetically generated variants designed to simulate obfuscation and payload diversity. Synthetic samples were generated to improve coverage of transformation patterns (e.g., encoding, concatenation), but care was taken to ensure that these samples preserve realistic execution semantics and do not introduce artificial bias. Although the dataset incorporates real-world payloads and execution scenarios, large-scale validation on live web applications remains limited due to the challenges of controlled ground-truth labeling in production environments. As such, the current evaluation focuses on reproducible experimental conditions. Future work will include deployment-based validation on instrumented real-world web applications to assess performance under dynamic, uncontrolled execution conditions.

Each sample represents a source-to-sink flow instance extracted during instrumented browser execution.

All malicious payloads (both from real-world sources and synthetically generated) were tested in an instrumented browser environment. Each payload was injected into its corresponding sink (e.g., innerHTML, eval) and only those payloads that successfully executed JavaScript code were considered malicious. Benign samples were similarly tested to confirm that they produced no JavaScript execution when injected into any sink context. This process ensures that the dataset contains only payloads that are genuinely dangerous in a DOM context, while benign samples accurately represent non-executable content. The dataset incorporates samples derived from multiple independent real-world sources, including publicly disclosed vulnerabilities and bug bounty reports. Although labeling was primarily execution-based, manual verification was also performed to ensure consistency of classification. The dataset incorporates payloads originating from multiple independent real-world applications and publicly disclosed vulnerabilities, including bug bounty reports.

Manual validation was performed to ensure labeling consistency across samples. Benign flows were constructed to reflect realistic DOM manipulation patterns observed in modern web applications, including dynamic content insertion and attribute-based updates, rather than simplified static cases.

5.2.2. Dataset Composition

The final dataset comprises 14,999 labeled source-to-sink flow instances 6812 malicious (45.4%) and 8187 benign (54.6%), as summarized in Table 2.

Table 2. Payload family distribution of the dataset. Malicious samples are categorized by injection technique; benign samples comprise non-executable content assigned to safe sink contexts.

Payload Family	Count	Percent (%)
Basic	5249	77.1%
Advanced	1183	17.4%
Obfuscated	380	5.6%
Total Malicious	6812	100%
Benign	8187	-
Grand Total	14,999	-

Each sample includes:

$$(F_{lex}(P), F_{ctx}(D), y) \tag{29}$$

where:

- $y = 1$ for malicious;
- $y = 0$ for benign.

To prevent data leakage:

- Duplicate payloads were identified by exact string matching and removed, ensuring no identical payloads appear in both training and test sets.
- For contextual variants (same payload across different sinks, e.g., innerHTML vs. innerText), we ensured that all instances of a given payload were assigned to the same partition, preventing the model from learning spurious correlations between payload and sink type.
- Obfuscated variants (e.g., base 64, HTML entity encoded) were grouped by their underlying obfuscation family and kept together in either training or testing. This prevents the model from learning to associate the obfuscation technique itself with maliciousness, rather than the decoded content.

The dataset spans seven DOM sink types: innerHTML, outerHTML, document.write, eval, href, script.src, and innerText. Execution-sensitive sinks such as innerHTML and eval contain predominantly malicious samples, while non-executable sinks such as innerText contain predominantly benign samples, directly validating the context-awareness contribution of the framework.

5.2.3. Preprocessing

Numerical features (payload length, character counts, entropy) were normalized to zero mean and unit variance to ensure equitable contribution during model training. Categorical features—including sink types, element types, attribute presence, and lexical flags—were already binary and required no further encoding.

Model stability and hyperparameter selection were evaluated using 5-fold stratified cross-validation, with stratification preserving the original class distribution across folds. A separate 20% hold-out test set, stratified by class label, was reserved exclusively for final

evaluation and remained unseen during training. All results reported in Section 5.6 are derived from this hold-out set.

5.2.4. Experimental Setup

All experiments were conducted on a standardized test platform

- Hardware: 11th GEN intel® core™ i7-11800H 2.60 GHz, 32 GB RAM;
- Browser: Google Chrome;
- ML framework: Scikit-learn Random Forest;
- Training: 80% training, 20% testing;
- Repetitions: Each experiment repeated 10 times.

5.3. Baseline Models

To evaluate the contribution of contextual modeling, we compare against:

- Payload-Only Random Forest (22 features);
- Support Vector Machine (lexical features);
- XGBoost (lexical features);
- CNN-based deep learning classifier;
- Taint Tracking Only (no ML classification).

All models are trained on identical training splits.

Baseline configurations were as follows. The Payload-Only Random Forest used 100 estimators; no pruning was applied. The SVM was configured with an RBF kernel (C = 1.0, gamma = 'scale'). The XGBoost was configured with 100 estimators and a learning rate of 0.1. The CNN comprised two 1D convolutional layers followed by a dense classification head, with ReLU activations and dropout (p = 0.3) for regularization; inputs were character-level embeddings padded to length 200.

Hyperparameters were set to commonly used defaults and were not tuned further, as the goal was to isolate the effect of feature design rather than classifier optimization.

5.4. Evaluation Metrics

We report:

$$Accuracy = \frac{TP + TN}{Total} \tag{30}$$

$$Precision = \frac{TP}{TP + FP} \tag{31}$$

$$Recall = \frac{TP}{TP + FN} \tag{32}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{33}$$

Additionally:

- False Positive Rate (FPR);
- ROC-AUC;
- Precision–Recall AUC.

Given operational deployment requirements, precision is treated as the primary metric.

5.5. Ablation Study

To isolate the effect of contextual features, we evaluate:

The ablation study demonstrates that hybrid feature integration substantially improves classification precision. The full hybrid model (M3) achieves 89.3% precision and 93.5% recall, outperforming the lexical-only baseline (M1) by 26.9 percentage points in

precision. These results confirm that DOM context features are the primary driver of accuracy improvement.

$$Precision_{combined} > Precision_{lexical} \tag{34}$$

Statistical validation performed using McNemar's test:

$$p < 0.05 \tag{35}$$

As shown in Table 3, the hybrid model achieves 89.3% precision, substantially outperforming the lexical-only baseline (62.4%). Statistical significance was validated using McNemar's test ($p < 0.001$).

Table 3. Ablation Study Results: Feature Set Comparison (M1–M3).

QModel	Feature Set	Precision	Recall	F1
M1	Lexical Only (22)	62.4%	90.2%	73.8%
M2	Context Only (20)	58.1%	72.5%	64.5%
M3	Hybrid(42)	89.3%	93.5%	91.3%

Feature Importance Analysis

To understand which features drive the model's decisions, we analyzed feature importance using Random Forest's Gini importance. Table 4 shows the top 10 most important features.

Table 4. Top 10 Feature Importance Rankings.

Rank	Feature	Type	Importance (%)
1	num_paren	Lexical	25.36
2	has_alert	Lexical	14.59
3	entropy	Lexical	10.83
4	length	Lexical	10.05
5	num_apos	Lexical	8.02
6	sink_href	Contextual	5.14
7	element_a	Contextual	5.05
8	has_onerror	Contextual	4.87
9	sink_innerHTML	Contextual	4.62
10	num_lt	Lexical	4.21
	Other 32 features	—	7.26

The importance analysis reveals several key insights about the model's behavior. Lexical features dominate the top ranks, occupying eight of the ten positions and accounting for 68.4% of the total importance among the top-10 features. The three most significant features—num_paren (25.36%), has_alert (14.59%), and entropy (10.83%)—are all lexical, indicating that the model first screens payloads based on syntactic patterns characteristic of malicious code. The prominence of parenthesis count and alert keyword presence directly reflects the structure of common XSS payloads, which frequently employ JavaScript function calls and event handlers.

However, contextual features, while fewer in number, play a critically distinct role. Four contextual features appear in the top 10—sink_href (5.14%), element_a (5.05%), has_onerror (4.87%), and sink_innerHTML (4.62%)—collectively contributing 19.68% to the top-10 importance. More importantly, these features serve as the model's semantic arbiters in borderline cases. When a payload exhibits suspicious lexical patterns (high parentheses

count, alert keyword) but is destined for a safe execution context—such as an href attribute on a standard hyperlink or a non-executable element—these contextual features enable the classifier to override the initial lexical suspicion and correctly label the flow as benign.

This two-stage decision architecture explains the model's balanced performance: lexical features cast a wide net, achieving high recall (93.5%) by capturing the vast majority of genuinely malicious payloads based on their syntactic signatures. Contextual features then act as a precision filter, reducing false positives by verifying whether the execution context actually permits code execution. The 26.9 percentage-point precision improvement over lexical-only models directly stems from this contextual override capability—precisely the cases where lexical features alone would trigger false alarms. The cumulative importance distribution shows that the top-10 features account for 92.74% of the model's predictive power, with the remaining 32 features contributing only 7.26%, confirming that the model's decisions are driven by a relatively compact, interpretable feature set.

5.6. Performance Results

Table 5 presents the performance comparison between the proposed hybrid framework and the three baseline models. Results are reported as mean values across 10 repeated experiments on a held-out test set of 3000 samples (20% stratified split from the 14,999-sample dataset).

Table 5. Performance Comparison of Baseline Models and Proposed Hybrid Framework.

Model	Accuracy	Precision	Recall	F1	FPR
Payload RF	93.1%	62.4%	90.2%	73.8%	12.3%
SVM	91.7%	58.6%	88.9%	70.5%	14.8%
XGBoost	94.2%	65.1%	91.3%	76.0%	11.2%
CNN	95.4%	69.8%	92.7%	79.6%	10.1%
Proposed Hybrid	97.8%	89.3%	93.5%	91.3%	3.2%

The proposed hybrid model (M3) achieves 89.3% precision and 93.5% recall on the held-out test set, with an overall accuracy of 97.8%. This represents a 26.9 percentage-point precision improvement over the context-insensitive baseline (M1: 62.4%, as shown in the ablation study), directly confirming that DOM sink context is the primary driver of reduced false-positive rates. The false-positive rate of 3.2% demonstrates that contextual features enable the model to correctly classify semantically benign injections that lexical analysis alone would misclassify as malicious.

5.7. Real-Time Performance Evaluation

The classification framework was evaluated using a Chrome extension deployed in a browser environment. Browser-based evaluation demonstrated a mean inference latency of 0.72 ms and a memory footprint of approximately 260 KB, within real-time operational constraints under the evaluated experimental conditions. Taint propagation tracking is implemented through JavaScript API hooking, with each sink interception involving only feature extraction and a single Random Forest inference call, introducing negligible per-call overhead. The Random Forest model consists of 100 decision trees operating on a 42-dimensional feature vector; the computational complexity of a single inference is $O(d \cdot log\ n)$. End-to-end latency benchmarking under high-frequency DOM mutation scenarios is identified as a direction for future work. A breakdown of the total inference latency indicates that approximately 0.12 ms is attributed to taint tracking interception, 0.28 ms to contextual feature extraction, and 0.32 ms to Random Forest inference, resulting in a total average latency of 0.72 ms per sink invocation. This decomposition confirms

that feature extraction constitutes the primary computational cost, while classification remains lightweight.

5.8. Adversarial Robustness Evaluation

We generate adversarial payload variants including case randomization ('<ScRiPt>'), HTML entity encoding, JavaScript string concatenation, 'String.fromCharCode()' obfuscation, and nested encoding layers. We measure degradation in precision and recall. Robustness score is computed as follows:

$$Robustness = 1 - \frac{| Metric_{clean} - Metric_{obfuscated} |}{Metric_{clean}} \quad (36)$$

5.9. Statistical Significance

To validate the reliability of the reported results, we applied three statistical tests. Confidence intervals were estimated over 10 repeated experiments with different random seeds; the hybrid model achieved 89.3% mean precision (95% CI: 88.1–90.5%) and 93.5% mean recall (95% CI: 92.8–94.2%), indicating stable performance across data partitions. A paired t-test across 5-fold cross-validation folds showed statistically significant differences between M3 and all baselines ($p < 0.01$). Additionally, McNemar's test comparing M3 against the strongest baseline yielded $p < 0.001$, confirming that the performance gap is not attributable to chance

5.10. Discussion of Results

The results point to four main findings. First, DOM context features drive the precision improvement: M3 reaches 89.3% precision compared to 62.4% for the lexical-only baseline—a 26.9-point gap that confirms payload-only models cannot distinguish benign injections in safe sink contexts from genuinely malicious ones. Second, Random Forest beats the CNN baseline on precision (89.3% vs. 69.8%) while using far fewer computational resources, which supports its choice for latency-sensitive deployment. Third, the ablation results show that context features push precision up while lexical features carry most of the recall—dropping either set hurts overall performance, so both are needed to reach the F1 of 91.3%. Fourth, the false positive rate of 3.2% is the lowest across all tested models, confirming that execution context is a necessary signal for accurate DOM XSS classification. It should be noted that the current evaluation is conducted on a near-balanced dataset, which does not fully reflect real-world deployment conditions where malicious DOM XSS instances are significantly rarer than benign flows. While this setup enables controlled comparison across models, it may overestimate precision under realistic traffic distributions. Future work will extend the evaluation to highly imbalanced datasets (e.g., 95:5 benign-to-malicious ratio) to more accurately assess false positive behavior in operational environments.

6. Discussion and Limitations

6.1. Interpretation of Results

The experimental findings demonstrate that incorporating runtime DOM contextual features significantly improves precision in DOM-based XSS detection. The ablation study confirms that lexical features alone are insufficient to differentiate between syntactically suspicious but semantically benign injections.

This supports the central hypothesis of this work:

Exploitability in DOM XSS is fundamentally a semantic property of execution context rather than a purely lexical property of payload structure.

The hybrid design achieves a balance between:

- Detection effectiveness (high precision);
- Operational viability (low latency);
- Model interpretability;
- Deployment feasibility.

In contrast to deep learning approaches that prioritize representation complexity, our framework prioritizes semantic relevance and contextual modeling.

6.2. Why Context Improves Precision

False positives in DOM XSS detection frequently arise when payloads contain suspicious tokens (e.g., '<script>', 'onerror', 'eval') but are injected into contexts that do not permit execution. For example, insertion into 'textContent' rather than 'innerHTML', assignment to non-executable attributes, or placement inside inert DOM branches. Payload-only classifiers lack the capacity to distinguish these cases. By explicitly modeling sink semantics, element type, attribute execution capability, and DOM hierarchy state, the system reduces misclassification of benign flows that share lexical similarity with malicious payloads.

6.3. Comparison with Existing Taint-Tracking and Hybrid Detection Frameworks

Table 6 situates the proposed framework relative to two representative prior systems on five dimensions: taint tracking, runtime context features, machine learning classifier, real-time capability, and false-positive reduction mechanism. Melicher et al. [6] introduced context-free taint tracking with manual rule-based classification, achieving 26.7% precision due to the absence of sink-context modelling. Lekies et al. [10] implemented dynamic taint tracking in V8 but used no machine learning, relying solely on taint propagation reach as the detection criterion. The proposed framework extends prior hybrid approaches by combining dynamic taint tracking with a DOM Context Analyzer that extracts sink type, element type, and parent taint state as structured features for a Random Forest classifier. This combination enables context-sensitive classification that explicitly models execution semantics, which is the mechanism responsible for the 26.9 percentage-point precision improvement over the Melicher baseline.

Table 6. Comparison of the proposed framework with representative taint-tracking and hybrid DOM XSS detection systems across five capability dimensions.

System	Taint Tracking	Runtime Context Features	ML Classifier	Feature Space	Real-Time	FP Reduction
Melicher et al. [6]	Yes	No	No	Flow-level	Yes	No
Lekies et al. [10]	Yes	No	No	None	Yes	No
Proposed	Yes	Yes	Yes (RF)	42 dims (lexical + contextual)	Yes	Yes

6.4. Deployment Implications

The proposed framework demonstrates potential suitability for:

- Browser extension deployment;
- Client-side security middleware;
- Embedded web runtime environments;
- Security instrumentation during development.

Its computational footprint suggests compatibility with real-time enforcement without requiring server coordination.

Furthermore, the use of Random Forest enables:

- Feature importance analysis;
- Explainable security decisions;

- Easier compliance auditing.

These properties are valuable in enterprise security contexts. Despite its practical potential, several deployment challenges must be considered. Browser API instrumentation may introduce compatibility issues with evolving JavaScript engines and security policies such as Content Security Policy (CSP). Additionally, interception of native DOM APIs may conflict with other browser extensions or security tools operating in the same environment. Performance trade-offs must also be managed carefully to avoid degrading user experience under high interaction workloads. These factors highlight the need for standardized browser-level support for security instrumentation.

6.5. Limitations

Despite promising results, several limitations remain.

1. **Dataset Availability**

High-quality labeled DOM XSS datasets remain scarce. Although care was taken to construct a representative dataset, the diversity of real-world JavaScript ecosystems may introduce unseen patterns.

Future work should incorporate larger-scale crawling of live applications. The dataset employs a near-equal class distribution which does not reflect real-world XSS prevalence, where malicious cases are significantly rarer than benign traffic. Evaluation under imbalanced class conditions is identified as a direction for future word.

2. **Advanced Obfuscation Techniques**

Highly sophisticated multi-stage obfuscation, including dynamically constructed payloads using runtime-generated code fragments, may partially evade detection.

Although contextual modeling improves robustness, adversarial adaptation remains an ongoing challenge. This is directly evidenced by the robustness score of 0.847 and the 18.7 percentage-point precision gap between clean and obfuscated payloads.

3. **Framework Instrumentation Dependency**

The system assumes reliable taint propagation tracking. Errors in taint instrumentation could lead to missed flows.

More advanced static-dynamic hybrid analysis could further improve coverage.

4. **Browser-Specific Variations**

Execution semantics may vary slightly across browser engines. Cross-browser validation is necessary to ensure portability. The reported results reflect Chrome-based evaluation only and may not generalize to other browser engines.

5. **Scope Boundaries**

The framework targets DOM-based XSS specifically and does not address:

- Server-side injection vulnerabilities;
- CSP misconfiguration exploitation;
- Browser engine exploitation;
- Side-channel attacks.

It should be considered part of a layered defense model rather than a complete web security solution.

7. Conclusions and Future Work

DOM-based Cross-Site Scripting remains one of the most challenging web application vulnerabilities due to its client-side execution semantics and invisibility to traditional

server-based defenses. Existing machine learning approaches achieve high recall but suffer from poor precision when applied specifically to DOM XSS scenarios, primarily due to context insensitivity.

This work introduces a context-aware hybrid detection framework integrating dynamic taint tracking with a DOM Context Analyzer and a lightweight Random Forest classifier. By combining 22 lexical payload features with 20 runtime contextual features, the system models exploitability as a semantic property of execution context rather than a superficial textual pattern.

Experimental evaluation demonstrates that contextual modeling significantly improves precision while maintaining real-time performance suitable for browser deployment. The framework achieves a favorable trade-off between detection effectiveness and computational efficiency, addressing key limitations of prior deep learning-heavy approaches.

The principal contribution of this work lies in elevating runtime semantic context to a first-class feature space for DOM XSS detection. This shift from payload-centric classification to execution-aware modeling provides a practical extension to existing DOM XSS detection approaches in client-side web security.

Future work will focus on four directions: evaluation on larger-scale real-world datasets, integration with static program analysis, improved robustness against advanced obfuscation techniques, and adaptation of the context-aware feature representation to other client-side vulnerability classes such as DOM-based open redirects and prototype pollution.

Future work will also address the class imbalance limitation by evaluating the framework under realistic conditions using datasets with 95% benign and 5% malicious samples, reflecting real-world web traffic distributions.

Author Contributions: Conceptualization, M.I.; Methodology, M.I. and D.B.; Software, M.I. and D.B.; Validation, M.I., D.B. and A.L.; Formal Analysis, M.I.; Investigation, D.B.; Resources, M.I. and D.B.; Data Curation, D.B.; Writing—Original Draft Preparation, D.B.; Writing—Review & Editing, M.I. and A.L.; Visualization, D.B. and M.I.; Supervision, M.I. and A.L.; Project Administration, M.I.; Funding Acquisition, M.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset used in this study was constructed from publicly available XSS payload repositories: https://github.com/swisskyrepo/PayloadsAllTheThings (accessed on 1 March 2026), https://portswigger.net/web-security/cross-site-scripting/cheat-sheet (accessed on 1 March 2026) and https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html (accessed on 1 March 2026); these public sources were supplemented with synthetically generated malicious and benign samples created using a custom mutation engine based on OWASP and PortSwigger vectors.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. OWASP Foundation. OWASP Top 10:2021—The Ten Most Critical Web Application Security Risks. Available online: https://owasp.org/Top10/2021/ (accessed on 27 February 2026).
2. Verizon. 2023 Data Breach Investigations Report (DBIR). Available online: https://www.verizon.com/business/resources/reports/dbir/ (accessed on 27 February 2026).
3. Ponemon Institute; IBM Security. Cost of a Data Breach Report 2023. Available online: https://www.ibm.com/reports/data-breach (accessed on 27 February 2026).
4. Baykara, M.; Das, R. A novel hybrid approach for detection of web-based attacks in intrusion detection systems. *Int. J. Comput. Netw. Appl.* **2017**, *4*, 149–158. [CrossRef]

5. Hannousse, A.; Yahiouche, S.; Nait-Hamoud, M.C. Twenty-two years since revealing cross-site scripting attacks: A systematic mapping and a comprehensive survey. *Comput. Sci. Rev.* **2024**, *52*, 100634. [CrossRef]
6. Melicher, W.; Fung, C.; Bauer, L.; Jia, L. Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning. In Proceedings of the Web Conference 2021 (WWW '21), Ljubljana, Slovenia, 19–23 April 2021; pp. 1–12. [CrossRef]
7. Mokbal, F.M.M.; Dan, W.; Imran, A.; Jiuchuan, L.; Akhtar, F.; Xiaoxi, W. MLPXSS: An integrated XSS-based attack detection scheme in web applications using multilayer perceptron technique. *IEEE Access* **2019**, *7*, 100567–100580. [CrossRef]
8. OWASP Foundation. Testing for DOM-Based Cross Site Scripting (WSTG-CLNT-01). In *OWASP Web Security Testing Guide, Version 4.2*; OWASP Foundation: Bel Air, MD, USA, 2021. Available online: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/11-Client-side_Testing/01-Testing_for_DOM-based_Cross_Site_Scripting (accessed on 1 March 2026).
9. Kaur, J.; Garg, U.; Bathla, G. Detection of cross-site scripting (XSS) attacks using machine learning techniques: A review. *Artif. Intell. Rev.* **2023**, *56*, 12725–12769. [CrossRef]
10. Lekies, S.; Stock, B.; Johns, M. 25 million flows later: Large-scale detection of DOM-based XSS. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13), Berlin, Germany, 4–8 November 2013; pp. 1193–1204. [CrossRef]
11. Stency, V.S.; Mohanasundaram, N. A study on XSS attacks: Intelligent detection methods. *J. Phys. Conf. Ser.* **2021**, *1767*, 12047. [CrossRef]
12. Thajeel, I.K.; Samsudin, K.; Hashim, S.J.; Hashim, F. Machine and deep learning-based XSS detection approaches: A systematic literature review. *J. King Saud Univ. Comput. Inf. Sci.* **2023**, *35*, 101628. [CrossRef]
13. HackerOne. *2023 Hacker-Powered Security Report*, 7th ed.; HackerOne Inc.: San Francisco, CA, USA, 2023. Available online: https://www.hackerone.com/resources/reports/2023-hacker-powered-security-report (accessed on 27 February 2026).
14. Alhamyani, R.; Alshammari, M. Machine learning-driven detection of cross-site scripting attacks. *Information* **2024**, *15*, 420. [CrossRef]
15. Li, X.; Wang, T.; Zhang, W.; Niu, X.; Zhang, T.; Zhao, T.; Wang, Y.; Wang, Y. An LSTM based cross-site scripting attack detection scheme for cloud computing environments. *J. Cloud Comput.* **2023**, *12*, 118. [CrossRef]
16. Hu, T.; Xu, C.; Zhang, S.; Tao, S. Cross-site scripting detection with two-channel feature fusion embedded in self-attention mechanism. *Comput. Secur.* **2023**, *124*, 103015. [CrossRef]
17. Dawadi, B.R.; Adhikari, B.; Srivastava, D.K. Deep learning technique-enabled web application firewall for the detection of web attacks. *Sensors* **2023**, *23*, 2073. [CrossRef] [PubMed]
18. Hydara, I.; Sultan, A.B.M.; Zulzalil, H.; Admodisastro, N. Current state of research on cross-site scripting (XSS): A systematic literature review. *Inf. Softw. Technol.* **2015**, *58*, 170–186. [CrossRef]
19. Charmet, F.; Tanuwidjaja, H.C.; Ayoubi, S.; Gimenez, P.; Han, Y.; Takahashi, H.; Kageyama, T.; Campowsky, K.; Labiod, H. Explainable artificial intelligence for cybersecurity: A literature survey. *Ann. Telecommun.* **2022**, *77*, 789–812. [CrossRef]
20. Arrieta, A.B.; Díaz-Rodríguez, N.; Del Ser, J.; Bennetot, A.; Tabik, S.; Barbado, A.; García, S.; Gil-López, S.; Molina, D.; Benjamins, R.; et al. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Inf. Fusion* **2020**, *58*, 82–115. [CrossRef]
21. Breiman, L. Random forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.