

Real-Time Detection of Multi-File DOM-Based XSS Vulnerabilities Using Static Analysis: A Developer-Oriented Approach for Securing Web Applications

Akira Kanaoka^a and Shu Hiura

Faculty of Science, Toho University, 2-2-1, Miyama, Funabashi, Chiba, Japan

Keywords: DOM-Based XSS, Static Analysis, Web Security, Real-Time Detection.

Abstract: This paper introduces a static analysis method for real-time detection of DOM-Based Cross-Site Scripting (XSS) vulnerabilities that occur across multiple files in web applications. As modular development in JavaScript becomes increasingly common, the risk of DOM-Based XSS vulnerabilities grows due to complex interactions between separate files. Existing detection methods often struggle to comprehensively identify these vulnerabilities. Our approach focuses on real-time detection during the development process by expanding static analysis to cover multiple files. We implemented this method as an extension for Visual Studio Code (VSCode), offering developers immediate feedback on potential security risks. In addition to proposing and evaluating our method, we also address the lack of suitable datasets for evaluation by creating a neutral and comprehensive dataset that includes multi-file DOM-Based XSS vulnerabilities. The evaluation shows that our method enhances the accuracy of DOM-Based XSS detection, contributing to improved security in web applications.


1 INTRODUCTION

Web-based services have become crucial components of the internet, extending beyond traditional information browsing to various applications. These services have gained significance as they now provide environments that facilitate easy construction, even for developers with limited technical knowledge. As web application technologies continue to proliferate, there is a growing risk of vulnerabilities when development proceeds without adequate expertise. Therefore, support is essential for developers to ensure the secure development of these applications.

According to the OWASP Top 10, which lists the most critical security risks for web applications, 94% of web applications have been found to contain some form of vulnerability related to injection attacks (OWASP, 2021). Among these vulnerabilities, Cross-Site Scripting (XSS) is a particularly dangerous one, allowing attackers to execute arbitrary JavaScript within a web application. According to JVN iPedia, which registers vulnerabilities in software products, XSS was the most frequently registered vulnerability in the third quarter of 2023 (October to Decem-

ber) (IPA, 2024). A specific type of XSS, known as DOM-Based XSS, occurs when a vulnerability in the manipulation of the DOM (Document Object Model) by JavaScript running in the browser is exploited. Unlike other types of XSS, DOM-Based XSS does not involve the server, making it difficult to detect. Moreover, since attackers can examine JavaScript source code using browser developer tools, it is an easy target. Therefore, it is crucial for developers to ensure that their source code does not contain vulnerabilities that could lead to DOM-Based XSS.

Research on preventing DOM-Based XSS has been ongoing, but various challenges remain. Developer-oriented approaches for detecting DOM-Based XSS vulnerabilities can be broadly categorized into two methods: post-development source code analysis and real-time analysis in the development environment. The former method requires developers to correct all affected parts of the entire program, which can be a significant burden. Therefore, the latter approach is more desirable. However, research and development on real-time detection of DOM-Based XSS vulnerabilities that span multiple files have not yet been conducted. There are also challenges in evaluating detection methods. Firing Range, a testbed that contains a wide variety of web application vulnera-

^a  <https://orcid.org/0000-0002-3886-5128>

bilities, is often used for evaluation. However, these vulnerabilities are limited to those occurring within a single file. In contrast, past instances of DOM-Based XSS have often spanned multiple files, making such evaluation insufficient. Furthermore, research on real-time detection of DOM-Based XSS vulnerabilities must evaluate whether the detection methods are effective for developers. Yet, no existing research has conducted user experiments to assess the proposed methods.

In this study, we explore support mechanisms for developers to prevent the inclusion of DOM-Based XSS vulnerabilities in their source code. To this end, we propose and implement a method for real-time detection of DOM-Based XSS vulnerabilities that span multiple files during the coding process. To verify the accuracy of this detection method, we created an evaluation dataset specifically designed to test these vulnerabilities, which we have released as an open-source resource on GitHub. We conducted both accuracy evaluations using the dataset and performance evaluations to measure detection time. In addition to evaluating accuracy and performance, we prepared user experiments to assess the usability of the proposed method from the developers' perspective. As a result, we evaluated the effectiveness of our proposed method in preventing DOM-Based XSS across multiple metrics and outlined future prospects for its implementation.

2 RELATED WORKS

2.1 DOM Based XSS

To determine the extent to which real web applications contain DOM Based XSS vulnerabilities, Lekies et al. surveyed the top 5,000 Alexa(Lekies et al., 2013). The results showed that 6,167 vulnerabilities were contained within 480 domains. More recently, DOM Based XSS vulnerabilities have also been found on sites such as Facebook and TikTok(Vulners.com, 2022; Leyden, 2020).

2.2 Detection Methods

Research aimed at preventing DOM-Based XSS has been conducted in the past. Generally, program analysis is divided into two types: dynamic analysis and static analysis. Similarly, research on DOM-Based XSS has also followed these two approaches, with dynamic and static analysis methods being explored.

In a study conducted by Parameshwaran et al., a technique for creating secure patches for websites

in the Alexa Top 1000 was proposed(Parameshwaran et al., 2015). They put forth a methodology for the identification of susceptible source code and its subsequent replacement with secure source code through the dynamic analysis of the code in operation while the application is executed against a corpus of source code representing services provided on the web. The proposed method is dynamic and has a relatively low overhead, as it can apply patches to vulnerable source code with an overhead of approximately 5%.

Additionally, studies have been conducted that do not include vulnerabilities through static analysis. Wang et al. demonstrated the deployment of API Hardening, a secure API for source code containing DOM Based XSS vulnerabilities, for two years to developers within Google(Wang et al., 2021). This resulted in a reduction in the occurrence of DOM Based XSS vulnerabilities. API Hardening is introduced as a compile-time checker after the source code is completed. It prevents vulnerabilities by replacing APIs that could be Sink with their own secure APIs.

Other studies have been conducted with the objective of preventing the occurrence of DOM-based XSS. These include the work of Liu et al. and others, who have organised, compared and analysed these methods and pointed out their advantages and disadvantages, including the aforementioned studies(Liu et al., 2019).

3 VULNERABILITY DETECTION

3.1 DOM Based XSS Occurring via Multiple Files

In modern development practices, the modular separation of JavaScript into multiple files has become common. This approach enhances code reusability and maintainability while supporting the integration of external and custom modules. However, managing interactions across a large number of files introduces potential risks of vulnerabilities, particularly DOM-Based XSS, due to insufficient handling of inter-file dependencies(ars TECHNICA, 2020).

A notable example is CVE-2022-23367, a DOM-Based XSS vulnerability spanning multiple files(CVE, 2022a; CVE, 2022b). This case involved a Source located at location.href and a Sink at location.search. The vulnerability arose from inadequate escaping mechanisms, with the Source and Sink existing in separate files linked via an import statement.

Addressing these challenges requires a systematic approach to inter-file vulnerability detection.

3.2 Necessary Countermeasures

To prevent DOM-Based XSS across multiple files, various techniques can be employed. Dynamic analysis, such as the approach by Parameshwaran et al. (Parameshwaran et al., 2015), effectively uses taint propagation to detect vulnerabilities independent of file separation. However, asynchronous JavaScript processing can limit its comprehensiveness by preventing certain event handlers from being executed, leaving potential vulnerabilities undetected.

Static analysis techniques, such as those proposed by Wang et al. (Wang et al., 2021), offer an alternative by replacing vulnerable APIs with secure ones. However, this approach is not universally applicable due to framework dependencies and development environment limitations. Additionally, static analysis is performed at compile-time, often after development is complete, necessitating extensive code modifications and redesigns in cases of detected vulnerabilities.

Given these limitations, there is a need for a real-time static analysis method that provides actionable feedback to developers during the development phase, addressing vulnerabilities spanning multiple files without reliance on post-development compile-time analysis.

3.3 Detecting DOM Based XSS Vulnerabilities Occuring Across Multiple Files

3.3.1 Overview

Effective detection of DOM Based XSS vulnerabilities requires not only comprehensive source code analysis after site completion but also real-time support during development. Static analysis by ESLint is limited to detecting DOM Based XSS within a single file, making it insufficient for identifying vulnerabilities that span multiple files.

To address this limitation, this study builds upon ESLint to propose a real-time detection method capable of analyzing a broader and more complex range of DOM Based XSS vulnerabilities. By extending the analysis scope to multiple files, this approach aims to provide developers with immediate feedback during the development process, enhancing vulnerability detection and mitigation in modularized JavaScript projects.

3.3.2 Processing Details

A comprehensive overview of the process is shown in Figure 1.

The detection pre-processing involves analyzing file dependencies from the source code open in the editor, constructing abstract syntax trees (ASTs) for all relevant files, and preparing them for exploration. ASTs are initially built for files open in the editor. If imports from other files are detected, ASTs for the corresponding source files are also constructed, with repeated checks confirming file dependencies.

Vulnerability detection proceeds in two phases: Provisional Sink Detection and Provisional Source Tracking. In the first phase, sections with data output, such as HTML, are identified as Provisional Sinks. The second phase tracks variables processed in Provisional Sinks to identify potential Sources. Variables in these processes are labeled as Temporary Sources and tracked backward through assignments, switching labels as needed. If tracking requires cross-file analysis, file dependencies guide AST traversal to continue tracking.

Finally, in the DOM-Based XSS Decision phase, combinations of Provisional Sources and Sinks are evaluated. If constants that cannot cause vulnerabilities are identified, the combination is dismissed. Otherwise, the pair is flagged as a DOM-Based XSS vulnerability.

Details are shown in the flowchart in Figure 2.

3.3.3 Implemented as an Extension to VSCode

To validate the proposed detection method, a prototype was developed as a VSCode extension using the Language Server Protocol (LSP), enabling compatibility with multiple development tools.

Unlike ESLint, which performs static analysis on single files, the custom JavaScript Linter supports multi-file analysis. The Linter is triggered when a JavaScript file is opened or modified, constructing abstract syntax trees (ASTs) via espree and analyzing them with estraverse to detect vulnerabilities across files.

4 EVALUATION DATA SET

4.1 Existing Evaluation Methods and Their Problems

4.1.1 Existing Evaluation Methods

Parameshwaran et al. conducted detection experiments on publicly available web applications (Parameshwaran et al., 2015), while Wang et al. evaluated their proposed method by observing its impact on developers reducing DOM-Based

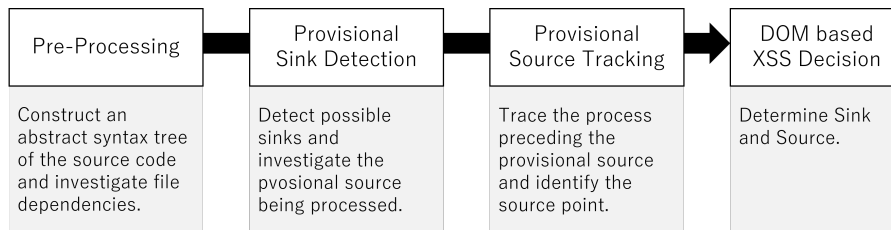


Figure 1: Overview of the process of the proposed method.

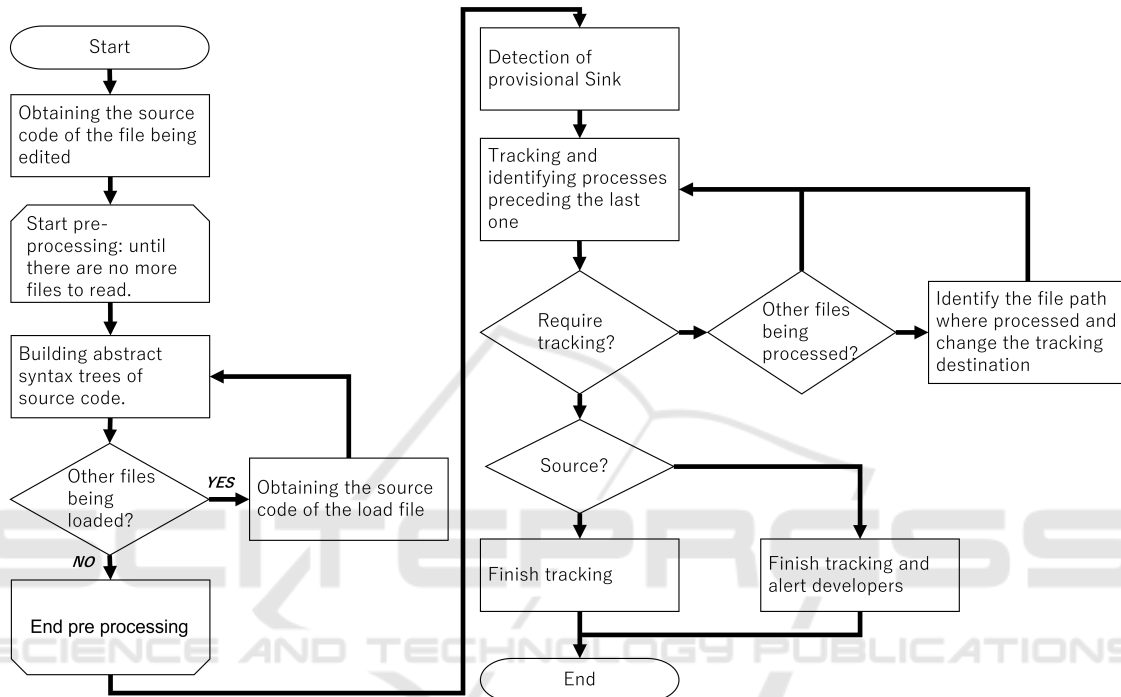


Figure 2: Process flowchart.

XSS vulnerabilities(Wang et al., 2021). However, these experiments lacked reproducibility as results varied depending on experimental environments, making it difficult to assess the detection methods' effectiveness.

In 2016, Pan et al. proposed a micro-benchmark for evaluating DOM-Based XSS, incorporating script triggers and Source-to-Sink flows in addition to Source and Sink types. However, it did not address cases involving multiple files(Pan and Mao, 2016).

4.2 Existing Dataset

Firing Range(google, 2018) is a testbed commonly used to evaluate existing methods. It includes a variety of web application vulnerabilities, including DOM-Based XSS. However, these are limited to vulnerabilities occurring within single files. Actual cases of DOM-Based XSS spanning multiple files(CVE, 2022a; CVE, 2022b) highlight the limitations of Fir-

ing Range as an evaluation dataset, making it insufficient for assessing multi-file vulnerability detection methods.

4.3 Overview of the Proposed Dataset

4.3.1 Vulnerabilities Employed in Firing Range

In this study, we propose an extended dataset for evaluating DOM-Based XSS vulnerabilities spanning multiple files, building on the Firing Range framework. Firing Range provides accessible source code from web applications, with pages categorized by vulnerability type. These pages are scripted in HTML and include both Sources and Sinks.

The proposed dataset reorganizes the extracted HTML scripts into separate JavaScript files. This process identified a total of 69 DOM-Based XSS vulnerabilities.

Table 1: File-to-file structure of the dataset.

No.	Methods of splitting files
01	Export variables (export at the end)
02	Export variables (export before declaration)
03	Export function (export at the end)
04	Export functions (export before declaration)
05	Import two in an intermediary file.

4.3.2 Structure Between Files

To address the single-structure vulnerabilities extracted from Firing Range, they were split into multiple files.

Identifying suitable syntax patterns for import and export statements required consulting StackOverflow and GitHub. On StackOverflow, we filtered DOM-Based XSS cases using tag information and search keywords, excluding low-rated posts, and investigated those spanning multiple files. On GitHub, we identified high-rated repositories based on star counts and analyzed cases of DOM-Based XSS across multiple files.

From these investigations, we identified commonly used patterns for JavaScript import and export statements. Five structural patterns between files were determined, as summarized in Table 1.

4.3.3 Creating Datasets with a Combination of Firing Range and File-To-File Structure

A dataset containing a total of 342 DOM based XSS cases occurring in multiple files was created by combining 69 Source and Sink combinations and five different file splitting methods.

Among the 69 DOM Based XSS vulnerabilities introduced in Firing Range, three sample data sets whose Source is document.referrer are excluded from this dataset because it is not syntactically possible to put export before the variable declaration in the case of file partitioning method No. 2. They were excluded from this data set.

These datasets are available on GitHub¹.

5 EVALUATION OF DETECTION METHODS

5.1 Evaluation Item

A prototype of the proposed detection method was used to evaluate its effectiveness. Two key indicators were assessed: accuracy, measuring the correct

¹<https://github.com/kanaoka-laboratory/multifile-domxss-dataset>

```

JS deep-level-1.js X
test-workspace > JS deep-level-1.js > ...
1 import { dl2f } from "../deep-level-2.js";
2
3 const dl1a = decodeURI(location.hash.substring(1));
4
5 dl2f(dl1a);
6
    
```

(a) deep-level-1.js

```

JS deep-level-2.js X
test-workspace > JS deep-level-2.js > ...
1 import { dl3f } from "../deep-level-3.js";
2
3 function dl2f(dl1p) {
4
5     const dl2a = dl1p;
6
7     dl3f(dl2a);
8
9 }
10
11 export { dl2f };
12
    
```

(b) deep-level-2.js

```

JS deep-level-10.js X
test-workspace > JS deep-level-10.js > ...
1 function dl10f(dl9p) {
2
3     const dl10a = dl9p;
4
5     document.getElementById("detail").innerHTML = dl10a;
6
7 }
8
9 export { dl10f };
10
    
```

(c) deep-level-10.js

Figure 3: Source code for sample data.

detection of vulnerabilities, and performance, evaluating the processing time required for detection.

For accuracy assessment, the dataset proposed in this paper was utilized. While the dataset primarily represents DOM-Based XSS across two files, an extended environment involving 10 files was created to evaluate scalability. This environment was also included in the evaluation.

An example of the source code used for the 10-file assessment is shown in Figure 3. In this case, a DOM-Based XSS vulnerability occurs when a string from location.hash is propagated through multiple files and ultimately processed by innerHTML. The source, location.hash, originates in deep-level-1.js, is traced through deep-level-2.js to deep-level-9.js, and is finally processed in deep-level-10.js. Each file alters the variable name, demonstrating the tracking capability of the proposed method.

As a performance evaluation, the processing time spent on detection was measured and evaluated.

The performance of the computers used in the experiment is shown in Table 2.

Table 2: PC specifications used for the evaluation.

Item	Description
CPU	Intel Core i5-1135G7 (2.4GHz)
RAM	16GB
OS	Windows 11 Home 23H2

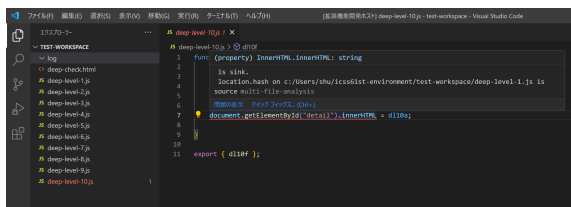


Figure 4: Prototype implementation detects DOM Based XSS vulnerability and notifies the screen on the sample data development screen.

5.2 Evaluation Results

5.2.1 Accuracy Evaluation

In detecting DOM-Based XSS vulnerabilities across 10 files, the prototype successfully traced the vulnerability through all files and notified the developer with a wavy line at the Sink point. A dialog box was also displayed, identifying the file containing the Source (location.hash) and highlighting both the Source and Sink for easier debugging (Figure 4).

From the dataset, 252 vulnerabilities were detected out of 342 cases (73.7%). Detection rates were unaffected by file splitting methods, as no differences were observed across methods.

Detection failures were attributed to limitations in the prototype’s handling of specific Source and Sink combinations, rather than issues with the proposed method itself. Table 3 summarizes the undetected combinations, aggregating 90 failures into 18 unique cases.

Failures due to Sources included 10 cases in Table 3 and 50 cases (14.6%) in the dataset. These involved Sources starting with 'localStorage' or 'sessionStorage' or strings like "badValue" defined as variables. The prototype does not currently handle variable-dependent strings, but extending the Provisional Sink Detection phase to account for variables would resolve this limitation.

Failures due to Sinks included 8 cases in Table 3 and 40 cases (11.7%) in the dataset. These involved Sinks where syntax variations obscured the detection of the target string. Since the Sink string does not appear explicitly in the program, the current string-matching approach fails to detect it. Addressing this issue will require detailed technical research and further refinement of the detection methodology, posing a challenge for future work.

5.2.2 Performance Evaluation

In order to check whether the tracking is actually performed across multiple files, the transition progress was output as a debug log and the processing time when the vulnerability was detected was measured.

The average time required for detection was 4.079 milliseconds, measured 10 times. According to the Nielsen Norman Group index, which is typically used in application UIs, a delay of 100 milliseconds or less is considered to be the time that the user perceives an instantaneous response to be taking place. 4 milliseconds is therefore a sufficiently acceptable processing time.

6 DISCUSSION

6.1 Vulnerability Detection

6.1.1 Performance

Although the performance evaluation showed good results in this assessment, these performances could well increase with processing complexity. The dataset used in this study was mostly data where DOM based XSS was generated by two files with a direct link. If this were to occur in three or more files, the complexity of the file structure would increase dramatically and the increase in detection time would be considerable. The construction and analysis of data sets with more than three files will be the subject of future research.

6.1.2 Accuracy of Detection

The detection on the dataset created in this study detected 252 vulnerabilities out of 342 vulnerability data, with a detection rate of approximately 73%, which leaves room for improving the accuracy. Therefore, it is required to increase the number of detectable Source and Sink types in the future.

As mentioned in the evaluation part, a large number of the 90 undetected 18 cases (50 in 10 cases) can be attributed to the prototype implementation, and the intrinsic accuracy of the proposed method can be said to be 88% $(=(242+50)/342)$.

The undetected cases caused by Sink (40 cases in 8 cases, 11.7 %) need to be further investigated. These are also related to sanitisation, which is the shaping of strings, and will require more detailed study.

Table 3: Source/Sink combinations that the prototype failed to detect in detection using the dataset.

Folder#.	Source	Sink
07-12	localStorage['badValue']	eval
08-09	localStorage.getItem('badValue')	document.write
08-12	localStorage.getItem('badValue')	eval
08-18	localStorage.getItem('badValue')	innerHTML
09-09	localStorage.badValue	document.write
11-14	location.hash	from.action
11-15	location.hash	function
11-17	location.hash	inlineevent
11-20	location.hash	javascript-url
11-23	location.hash	onclik-addEventListener
11-24	location.hash	onclik-setAttribute
11-30	location.hash	parmCodeEvent.value
14-04	location.search	a.xlink-href
16-12	sessionStorage['badValue']	eval
17-09	sessionStorage.getItem('badValue')	document.write
17-12	sessionStorage.getItem('badValue')	eval
17-18	sessionStorage.getItem('badValue')	innerHTML
18-09	sessionStorage.badValue	document.write

6.1.3 Application to Fix Support

Although this research was a proposal to detect DOM Based XSS, proposals for fixing vulnerabilities by applying the detection mechanism can also be considered. Since VSCode extensions can perform string transformation of source code in real-time, they can not only detect but also fix vulnerabilities. For example, in innerHTML, which is mentioned as Sink in this article, replacing it with textContent allows HTML tags, etc. to be output as strings and prevents DOM Based XSS from occurring. However, as the method of modification differs depending on the combination of Sink and Source, it is necessary to propose modifications suitable for each combination.

6.2 Dataset for Evaluation

6.2.1 Comparison with Firing Range

The dataset created is a variation of the file structure of the Firing Range so as not to alter the behaviour of the web application itself. Therefore, the data behaves in a similar way to the Firing Range and remains compatible. As such, the reliability of the dataset would be equivalent to that of the Firing Range.

6.2.2 Dataset Comprehensiveness

The dataset created in this study is a partial extension of the Firing Range and could be expanded in the future to include more complex or special vulnerable data.

One approach to increasing data is to increase the number of syntactic patterns when dealing with Sink

and Source, in addition to further increasing the number of file partitioning methods and combinations of Sink and Source. For the former, it is desirable to expand the data set by increasing the number of Sink and Source combinations that are not listed in the Firing Range, as well as sample data considering file splitting methods other than the patterns referred to here. The latter syntax pattern for handling Sink and Source is considered to be a code that can be more easily included during development by investigating the usage of each Sink and Source on StackOverflow and GitHub, etc., which were used to determine the file division method this time.

Another approach is to increase the number of files comprising the vulnerability: if the vulnerability is composed of $n \geq 3$ files, the structure representing the location of its Sink and Source and their relationship becomes more complex. It becomes a graph structure with files as nodes and file calling relationships as edges, suggesting that a graph-theoretic approach becomes important.

6.3 Usability Consideration

In this prototype, the dialogue generation and the relevant parts of the source code are marked with wavy lines on the VSCode. The appropriateness of this user interface also needs to be fully discussed. Although this paper focuses on the functional aspects of vulnerability detection and does not evaluate them, the suitability of the UI is a factor that should be discussed further: if the UI is not suitable, even if vulnerabilities are detected and further modifications are proposed, the user, the software developer, may not be aware of these proposals or may not accept them. UI research

to make better use of technological advantages will also be important.

7 CONCLUSIONS

In this study, we proposed and implemented a static analysis method for the real-time detection of DOM-Based XSS vulnerabilities that occur across multiple files in web applications. As JavaScript development increasingly adopts modular approaches, the risk of undetected vulnerabilities, particularly DOM-Based XSS, has grown due to the complex interactions between separate files. Our method addresses this issue by providing developers with immediate feedback during the coding process, helping to prevent the introduction of these vulnerabilities.

To evaluate the effectiveness of our proposed method, we created a comprehensive and neutral dataset that includes multi-file DOM-Based XSS vulnerabilities, which was previously unavailable. This dataset not only enabled a thorough evaluation of our detection method but also serves as a valuable resource for the broader research community.

Our evaluation demonstrated that the proposed method significantly improves the accuracy of DOM-Based XSS detection, thereby contributing to enhanced security in web applications. Furthermore, by integrating this tool into development environments, developers can more effectively mitigate potential security risks in real-time. Future work will focus on refining the detection accuracy and extending the dataset to cover a broader range of vulnerability scenarios, as well as conducting user studies to further assess the usability and effectiveness of the tool in real-world development environments.

ACKNOWLEDGEMENTS

This work was supported in part by JSPS KAKENHI Grant Number JP22K12035, and JST, CREST Grant Number JPMJCR22M4, Japan.

REFERENCES

ars TECHNICA (2020). Sourcegraph: Devs are managing 100x more code now than they did in 2010 — ars technica.

CVE (2022a). Cve - cve-2022-23367.

CVE (2022b). Cve - cve-2022-45020.

google (2018). Github - google/firing-range.

IPA (2024). Vulnerability countermeasure information database jvn ipedia registration status [2023 4th quarter (oct. - dec.)] — enhancing information security — ipa information-technology promotion agency, japan.

Lekies, S., Stock, B., and Johns, M. (2013). 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*, page 1193–1204, New York, NY, USA. Association for Computing Machinery.

Leyden, J. (2020). Xss vulnerability in 'login with facebook' button earns \$20,000 bug bounty — the daily swig.

Liu, M., Zhang, B., Chen, W., and Zhang, X. (2019). A survey of exploitation and detection methods of xss vulnerabilities. *IEEE Access*, 7:182004–182016.

OWASP (2021). A03 injection - owasp top 10:2021.

Pan, J. and Mao, X. (2016). Domxssmicro: A micro benchmark for evaluating dom-based cross-site scripting detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 208–215.

Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., and Saxena, P. (2015). Auto-patching dom-based xss at scale. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, page 272–283, New York, NY, USA. Association for Computing Machinery.

Vulners.com (2022). Tiktok: Dom xss on ads.tiktok.com - vulnerability database — vulners.com.

Wang, P., Bangert, J., and Kern, C. (2021). If it's not secure, it should not compile: Preventing dom-based xss in large-scale web development with api hardening. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1360–1372.