




JSMBox—A Runtime Monitoring Framework for Analyzing and Classifying Malicious JavaScript

Phu H. Phung¹ (✉) , Allen Varghese¹, Bojue Wang², Yu Zhao², and Chong Yu²

¹ Intelligent Systems Security Lab, Department of Computer Science, University of Dayton,
300 College Park Ave, Dayton, OH 45469, USA
phu@udayton.edu

<https://isseclab-udayton.github.io/>

² Department of Computer Science, University of Cincinnati, 2901 Woodside Drive, Cincinnati,
OH 45221, USA

Abstract. In recent years, there has been a notable increase in the prevalence of malicious websites, leading to a majority of cyber-attacks and data breaches. Malicious websites often incorporate JavaScript code to execute attacks on web browsers. Despite existing methodologies documented in the literature, the analysis and detection of malicious JavaScript pose significant challenges due to the dynamic nature of JavaScript and the use of advanced evasion techniques. These challenges motivate the need for an innovative and efficient approach to comprehensively analyze the code to identify its malicious intent. In this paper, we introduce a monitoring approach for analyzing JavaScript code, which can capture all of the code's features at runtime. Our method leverages the security reference monitor technique to mediate JavaScript security-sensitive executions, including function calls and property accesses. Therefore, the proposed method can capture behaviors at runtime regardless of how the code is written, even with recent advanced evasion techniques like WebAssembly diversification. We have implemented our approach as a JavaScript dynamic analysis framework called JSMBox in a Chromium-based browser extension. Our experiments demonstrated that JSMBox is capable of effectively countering sophisticated evasion techniques found in modern malicious JavaScript code, including WebAssembly diversification. We have also evaluated the framework's ability to classify malicious behaviors based on a large-scale raw dataset comprising about 20,000 malicious and benign webpages. Our developed tool automatically launches the browser to execute these webpages, records JavaScript code execution events, and captures their execution frequency as extracted features. We have tested the extracted dataset with various machine-learning models, yielding promising experimental results that confirm the effectiveness of our approach and achieve a high accuracy rate.

Keywords: JavaScript · Dynamic Analysis · Runtime Monitoring · Maliciousness Classification

Phu H. Phung: Work done while the author was a Visiting Scholar in the Department of Electrical and Computer Engineering at the University of Cincinnati.

1 Introduction

The ubiquity of JavaScript in web development, as highlighted by W3Techs¹, poses both opportunities and risks. While JavaScript enhances user interaction and web functionalities, it has also become a vector for cyberattacks, particularly through malicious code on webpages. Indeed, malicious webpages with JavaScript code that launch attacks on web browsers have become increasingly problematic in recent years, carrying out threats against the user's browser, such as stealing the user's credentials or downloading additional malware. Unfortunately, the dynamic nature of the JavaScript language and its tight integration with the browser make it challenging to detect and block malicious JavaScript code. JavaScript-based attacks on webpages are a recent trend and top Internet security threats [1], which can defeat traditional signature-based approaches used in anti-virus tools [2].

Analyzing and detecting malicious JavaScript have received high attention and are still an active research direction in the literature [3], which employ static analysis, dynamic analysis, or combined static and dynamic analysis techniques [4]. Static analysis is a traditional approach that typically extracts the semantic structure of the source code, abstract syntax tree, strings, objects, and functions to provide features for detection or machine learning algorithms. However, conventional static analysis methods typically fail to deal with dynamically generated code and evasion techniques used by attackers to hide malicious code [5]. On the other hand, dynamic analysis techniques execute JavaScript code; therefore, they can capture dynamically generated code and runtime behaviors that static analysis methods might omit. Although JavaScript dynamic analysis approaches offer advantages in behavioral analysis and runtime features, their realizations suffer shortcomings [6]. For example, several methods, e.g., [7, 8], leverage platform-specific tools such as Windows-based in-browser debuggers that are not always available in other systems. Cova et al. [2] extract dynamic features from execution traces using the HtmlUnit with Rhino engine simulation environment, which attackers can bypass by leveraging the differences between the emulated environment and a real browser. Recent malicious JavaScript code employed advanced evasion techniques to detect and subvert dynamic analysis methods [6]. Notably, none of the existing work can tackle evasion techniques using WebAssembly [9, 10].

The challenges mentioned above highlight the need for a robust analysis method that can capture dynamic behaviors in potential malicious JavaScript code, especially in the presence of advanced evasion techniques. To this end, we propose a novel JavaScript runtime analysis method and framework encompassing all JavaScript executions, including traditionally on-the-fly generated code and advanced evasion techniques. Our approach mediates JavaScript's security-sensitive operations, including function calls and property accesses at runtime, by leveraging the traditional security reference monitor technique [11]. Since we monitor the code execution, our method can capture the code behaviors regardless of the code's structure or evasion techniques. Specifically, the contributions of our work are as follows:

¹ According to the World Wide Web Technology Surveys in July 2024 (<https://w3techs.com>), 98.9% of all websites contain JavaScript code, which will be loaded and executed in a browser at the end-user.

- We introduce a novel runtime analysis method and framework by leveraging the inlined security reference monitor technique to execute JavaScript code in webpages to capture its behaviors for maliciousness classification and detection. Our framework is designed to allow customization and fine-tuning of the feature extraction process, providing the most important features for machine learning models to improve their accuracy and reliability.
- We have developed the proposed method as a JavaScript library, utilizing the language’s flexibility and platform independence to create a lightweight runtime monitor. This allows us to efficiently capture all executions and their contexts, regardless of their appearances. We have implemented the framework as a browser extension, meeting the essential requirements for security reference monitors and preventing evasive code from concealing its behaviors.
- We have demonstrated that our framework is highly proficient in extracting runtime features that are crucial for machine learning models to accurately classify malicious JavaScript on large-scale raw datasets. As entailed in Sect. 4, our proposed method offers a more effective feature extraction solution than traditional static analysis techniques and advances beyond recent dynamic or hybrid analysis approaches in dealing with malicious code that employs sophisticated evasion tactics, including the latest WebAssembly obfuscation and diversification.

The remainder of this paper is structured as follows. In Sect. 2, we provide an overview of the background, review the literature, and discuss related work. Following that, Section 3 includes a running example that motivates our work, and presents our approach to developing and implementing the framework. In Sect. 4, we outline the evaluation of our proposed framework, in comparison with closely related work. Finally, we conclude our contributions and outline potential future work in Sect. 5.

2 Background and Literature Review

This section briefly describes the background of JavaScript and its analysis methods. We also discuss challenges in detecting malicious JavaScript code with evasion techniques and provide examples. Finally, we review the literature and compare related work with our JSMBBox framework.

2.1 JavaScript and Malicious Webpages

JavaScript is one of the most popular versatile scripting languages primarily used for web development, enabling interactive and dynamic elements on web-pages. When a browser renders a webpage, it executes embedded JavaScript code, whether inlined, sourced from the same host, or retrieved externally. This code can access and alter the webpage’s content and data stored in the browser. Furthermore, JavaScript can dynamically generate and execute new code, as well as load and run external scripts in real time. These dynamic features can be lever-aged by both developers and attackers [12]. By inserting harmful JavaScript code, attackers can craft webpages to exploit vulnerabilities in users’ web browsers. These pages can contain various types of malicious content,

such as malware, phishing forms, or other forms of harmful information. Despite existing detection tools, JavaScript-based attacks on webpages remain a recent prominent Internet security concern [1].

2.2 JavaScript Analysis Methods

Existing works propose solutions from several approaches, including static analysis, dynamic analysis, and hybrid analysis [4] to analyze and detect malicious JavaScript code. Specifically, static analysis is a traditional approach that typically extracts the semantic structure of the code to provide features for detection. Unlike static analysis, which analyzes code without executing it, dynamic analysis runs the code and observes its interactions with the environment in real-time [13]. By executing code, dynamic analysis can discern malicious activities that static analysis might overlook [13]. Furthermore, some existing works use a hybrid approach, combining static and dynamic analysis techniques. These works typically utilize static analysis to help identify known patterns and vulnerabilities before execution while using dynamic analysis to provide real-time insights into the actual behavior of the code during runtime execution [14]. We discuss these methods in detail in the related work sub-section (Sect. 2.4).

2.3 Evasion Techniques

Evasion techniques of malicious JavaScript code are a critical aspect of contemporary cyber threats, wherein attackers employ sophisticated strategies to evade detection mechanisms and execute malicious actions within web environments. These techniques circumvent traditional security measures, including antivirus software, intrusion detection systems, and web application firewalls, posing significant challenges to cybersecurity professionals and researchers [15]. Obfuscation is one of the commonly used evasion techniques. This technique complicates the readability and analysis of code by altering its structure and appearance to obscure its intended functionality. Techniques such as variable obfuscation, string obfuscation, property encryption, control flow flattening, dead code injection, debugging protection, self-defending, and polymorphic mutation are often utilized to impede code analysis [5, 16, 17]. Research indicates that 71% of examined malicious samples employ obfuscation techniques [18]. We describe common JavaScript evasion techniques below.

Obfuscation Techniques JavaScript obfuscation is a technique designed to make JavaScript code difficult to understand and analyze. This mechanism enhances the protection of the code and makes it more challenging to reverse engineer or replicate. The primary purpose of obfuscation is to conceal the true intent and structure of the code without altering its functionality.

Various obfuscation techniques are available for different aspects of JavaScript code. Below, we list common obfuscation methods identified in the literature, together with their code snippets to illustrate their techniques.

- *Variable and Function Renaming*: Changing the names of variables and functions to make the code more challenging to understand and analyze [15].

```

1 // Original code
2 function calculateArea(radius) {
3   const PI = 3.141592653589793;
4   return PI * radius * radius;
5 }
6 // Obfuscated code
7 function a(b) {
8   const c = 3.141592653589793;
9   return c * b * b;
10 }

```

Listing 2.1. Illustration of variable and function renaming obfuscation method

- *Code Compression (Minification)*: Compression is a technique used to reduce the size of data or code by encoding information in a more compact format. In the context of software development and obfuscation, code compression involves removing unnecessary characters, spaces, and lines from the source code to make it more concise [19].

```

1 // Original code
2 function addNumbers(a, b) {
3   return a + b;
4 }
5 // Compressed code
6 function addNumbers(a,b){return a+b;}

```

Listing 2.2. Illustration of code compression/minification obfuscation method

- *Code Transformation*: Altering the structure and form of the code, such as changing the form of conditional statements or using ternary operators [20].

```

1 // Original code
2 function isEven(num) {
3   if (num % 2 === 0) {
4     return true;
5   } else {
6     return false;
7   }
8 }
9 // Transformed code
10 function isEven(a){return 0===a%2}

```

Listing 2.3. Illustration of code transformation obfuscation method

- *Dead code injection*: Dead code injection is a technique used to inject unused or non-executing code into a program. This technique can be employed as a form of obfuscation to make the code more complex and difficult to analyze. Injecting dead code does not affect the program's functionality but can confuse and deter reverse engineers, making it more challenging for them to understand the program's logic and structure [21].

```

1 function calculateSum(a, b) {
2   // Dead code injection
3   if (false) {
4     console.log("This code will never execute.");
5   }
6   // Actual code
7   return a + b;
8 }
9 console.log(calculateSum(5, 3));

```

Listing 2.4. Example of dead code injection obfuscation method

- *String Encoding*: Converting string literals into other forms, like using Unicode encoding or Base64 encoding [22].

```

1 // Original code
2 const greeting = "Hello world!"
3 // Obfuscated code
4 const encodedString = "%48%65%6C%6C%6F%2C%20%77%6F%72%6C%64%21";
5 const decodedString = unescape(encodedString);
6 console.log(decodedString); // Output: "Hello, world!"

```

Listing 2.5. Example of string encoding obfuscation method

- *Indirect method call*: Indirect method call is a programming technique that allows the determination of which method or function to call dynamically at runtime. This is typically achieved using function pointers, callback functions, or function objects. While indirect method calls enhance code flexibility, they may also increase code complexity and difficulty of understanding [23].

```

1 // Define a function
2 function greet() {
3     console.log("Hello!");
4 }
5 // Store the function name in a variable
6 var funcName = "greet";
7 // Indirectly call the function
8 window[funcName](); // Outputs: Hello!

```

Listing 2.6. Example of indirect method call obfuscation method

- *Instruction substitution*: Instruction substitution is an obfuscation technique that involves replacing original instructions in a program with equivalent instructions that have a different structure or form, thereby increasing the complexity and difficulty of understanding the code while maintaining its functionality [24].

```

1 // Original addition function
2 function add(a, b) {
3     return a + b;
4 }
5 // Obfuscated addition function
6 function add_obfuscated(x, y) {
7     return x - (-y);
8 }

```

Listing 2.7. Illustration of instruction substitution obfuscation method

- *Non-alphanumeric code*: Non-alphanumeric code is an obfuscation technique primarily used to replace alphabetic and numeric characters in code with non-alphanumeric characters, such as symbols and special characters, to increase the complexity and difficulty of understanding the code [25].

```

1 alert((+[[]][+[]]+[+])[[+]]+([+[]]+[+])[[+][+][+]]+[+[]])
2 [+[]]+([+[]]+[+])[[+][+][+][+]]+[+[]]+[+[]]+([+[]]+[+])
3 [[+][+]]+[+[]]+([+[]]+[+])[[+]]// "alert"

```

Listing 2.8. Example of non-alphanumeric code obfuscation method

- *String splitting*: This method involves separating a string or function name into multiple smaller fragments and then reassembling them at runtime [5].

```
1 // Original code: alert('This could be malicious');
2 // Splited code
3 var jj = 's\'';
4 var by = 'rt(\'';
5 var dh = 's c';
6 var gf = 'ma';
7 var eu = 'oul';
8 var ii = 'iou';
9 var fg = 'd be';
10 var cg = 'Thi';
11 var ax = 'ale';
12 var hh = 'lic';
13 eval(ax + by + cg + dh + eu + fg +
14 gf + hh + ii + jj);
```

Listing 2.9. Illustration of string splitting obfuscation method

WebAssembly obfuscation and diversification WebAssembly (Wasm) is a binary instruction format that is designed to be executed in a web browser, aiming to provide a portable, high-performance for web applications that leverage existing codebases and libraries written in other common programming languages rather than JavaScript. With that design, WebAssembly has quickly become an essential part of the Web, providing a great alternative to JavaScript [26]. On the other hand, WebAssembly has also been used as a sophisticated evasion technique to conceal malicious code in webpages and evade code analysis and detection techniques. Wobfuscator [9] is a recent research tool demonstrating a WebAssembly obfuscation technique that transforms parts of the JavaScript computation into WebAssembly and evades JavaScript malware detection tools. In [10], the authors developed an automatic binary WebAssembly diversification evasion technique that can evade most of the cases in popular detectors such as VirusTotal and MINOS. The research findings motivate innovative approaches that can address the modern technology on the Web.

Browser Fingerprinting Browser fingerprinting is a technology that creates a unique identifier (fingerprint) by collecting various attributes and behaviors of the client's web browser, allowing for user identification and tracking. These attributes may include the browser's user agent string, operating system, screen resolution, and installed plugins. Browser fingerprinting is commonly used for purposes such as user tracking, personalized advertising, and security verification [27–29].

Attackers can utilize this technology to examine specific attributes or configurations of the client's web browser to determine if they meet the conditions for an attack. For example, attackers may inspect the browser's user agent string or other characteristics to determine if it is the target browser and then execute malicious code or attacks accordingly. This type of inspection may be conducted to ensure the success of an attack or to tailor different attack strategies based on the targeted browser.

Browser fingerprinting or detection helps attackers ensure that their exploit is only triggered on the intended target browsers. This technology is used not only to detect the browser's version but also can be used to detect client-side content; it also possesses strong anti-detection capabilities, making it immune analysis methods [30].

2.4 Related Work

Methods to analyze webpages and JavaScript code for classifying and detecting malicious JavaScript in the literature can be categorized into three categories: static analysis, dynamic analysis, and a hybrid combination of static and dynamic analysis [4]. In this section, we briefly discuss the static analysis approaches and review the dynamic approaches in more detail compared with our approach.

Static Analysis Traditional methods of static analysis are engineered to identify malicious JavaScript code without executing the source code. These methods extract the features of malicious code to build a malicious feature library. Subsequently, they evaluate the detected code to determine if it matches the features within this malicious feature library. More recent approaches employ machine learning and deep learning to improve the detection rate. These works typically transfer detected code into vectors using various methods, such as fixed-length vector representation, abstract syntax tree (AST), Control Flow Graph, and Program Dependency Graph [31–35]. Based on these representations, detection models are built using machine learning classifier algorithms, including Random Forest, Naive Bayes, Support Vector Machine (SVM), and Random Forest. For example, ZOZZLE [36] generates features based on the hierarchical structure of the JavaScript AST and employs rapid pattern matching and Naive Bayes classifier for detection. JStep [33] is a static malicious JavaScript detector that uses lexical analysis, AST, control flow, and data flow information, utilizing a Random Forest classifier. Ren et al. [15] studies the effects of obfuscation on existing malicious JavaScript detectors, employing a range of classifiers. However, conventional static analysis methods typically fail to deal with dynamically generated code and evasion techniques (e.g., obfuscation code) used by attackers to conceal malicious code [5]. In a recent study [15] of representative static analysis-based approaches of detecting obfuscation code, they find “the feature spaces of existing detectors can only reflect shallow differences in code, not about the nature of benign and malicious, which can be easily affected by the differences brought by obfuscation.” In other words, state-of-the-art static analysis-based approaches are still unable to detect malicious code that employs evasion techniques accurately.

Dynamic Analysis Dynamic analysis-based methods involve executing the program to uncover specific behaviors, even when the program is obfuscated, as indicated by Kim et al. [30]. Researchers employ these approaches to extract behavioral features during the execution of code for the classification of malicious code within test environments, including sandboxes [2, 7, 37, 38], honeypots [39, 40], and browsers [6]. Snyder et al. [41] investigated the usage patterns of JavaScript features in modern web browsers, revealing that most features are rarely used and are often blocked by ad and tracking blockers. Based on how third-party trackers manipulate browser state, Roesner et al. [42] developed an in-band client-side method for detecting and classifying five kinds of third-party trackers. Yagemann et al. [43] designed an offline control flow analysis method for attack detection using deep learning on hardware execution traces to model a program’s behavior and detect control flow anomalies. In addition, Ratana-worabhan et al. [44] introduced a runtime heap-spraying detector that examines individual objects in the heap, interpreting them as code and performing a static analysis to detect malicious intent.

However, similar to static analysis, one limitation of these methods is their inefficiency in detecting evasion techniques.

Many studies [4, 45–49] have focused on addressing obfuscated code to overcome this aforementioned limitation. Li et al. [45] proposed a forensic engine that can efficiently record fine-grained details on the execution of JavaScript programs within the browser. Additionally, Fang et al. [46] proposed a malicious JavaScript detection model based on LSTM that extracts features from the semantic level of bytecode and optimizes the word vector. Furthermore, Song et al. [4] constructed the Program Dependency Graph to generate semantic slices. Based on this, they designed a malicious JavaScript detection model utilizing bidirectional LSTM. Neasbitt et al. [47] presented an online forensic data collection system that allows for recording enough detailed information to enable a full reconstruction of web security incidents, including phishing attacks. Moreover, Wang et al. [49] designed a deep learning framework that integrates sparse random projection, a deep learning model, and logistic regression to detect malicious JavaScript code. Rieck et al. [48] inspected web pages to block the delivery of malicious JavaScript code and collected static and dynamic code features for malicious pattern analysis. Besides, Jueckstock et al. [6] proposed a dynamic analysis framework hosted inside V8, the JavaScript engine of the Chrome browser, that logs native function or property accesses during any JavaScript execution to monitor browser behaviors. In comparison to others, this method proves significantly more efficient in detecting evasion techniques, as it can deal with both obfuscated code and browser fingerprinting. However, none of the existing work can address all evasion techniques discussed previously.

In contrast to the aforementioned research, our proposed method addresses the challenges of analyzing malicious JavaScript arising from dynamic JavaScript features and all advanced evasion techniques by capturing the behaviors of both static and dynamically generated code. We present our technical approach and discuss how it can tackle the challenges in the next section.

3 Technical Approach and Implementation

3.1 A Motivating Example

To present our approach, we consider the following JavaScript snippet example illustrated in Listing 3.1, providing key concepts underlying our proposed approach. The provided example highlights the code obfuscation of the `HTMLCanvasElement.prototype.toDataURL` method, typically used in malicious code that exploits fingerprinting attacks to identify and track users [50]. We note that actual malicious obfuscated JavaScript codes are substantially more sophisticated.

```

1 const values = [
2   72, 84, 77, 76, 67, 97, 110, 118, 97, 115, 69, 108, 101, 109, 101, 110,
3   116, 46, 112, 114, 111, 116, 111, 116, 121, 112, 101, 46, 116, 111, 68,
4   97, 116, 97, 85, 82, 76];
5 const code = values.map(value => String.fromCharCode(value)).join('');
6 eval(code);

```

Listing 3.1. A motivating example

Since static analysis-based techniques do not execute the code, they encounter challenges in recognizing these obfuscated scripts [5]. This limitation stems from the inadequacies of current machine-learning-based static analysis techniques to accurately identify malicious code that employs evasion strategies, as highlighted in recent empirical studies [15]. As a result, the obfuscated JavaScript is executed, activating `HTMLCanvasElement.prototype.toDataURL` method, which malicious actors can manipulate. To address the challenges in detecting the malicious intent of obfuscated code, various dynamic analysis strategies [2, 6, 7, 37–40] have been proposed. These strategies aim to monitor the runtime behavior of JavaScript code because obfuscated code cannot disguise its activities during execution. However, a notable gap in existing research is the lack of focus on monitoring the potentially malicious use of the `HTMLCanvasElement.prototype.toDataURL` method and the application of machine learning techniques to determine their maliciousness [50, 51].

3.2 Approach and the Overview of the Proposed Framework

The running example discussed above is one of many challenges in JavaScript code analysis that motivate our work. To address these challenges, we leverage the runtime monitoring approach that executes JavaScript code to log its behaviors, regardless of their appearance or evasion techniques. Specifically, we propose JSMBBox, a dynamic analysis JavaScript framework that adopts a behavioral sandbox approach [52]. Our proposed approach aims to monitor and record.

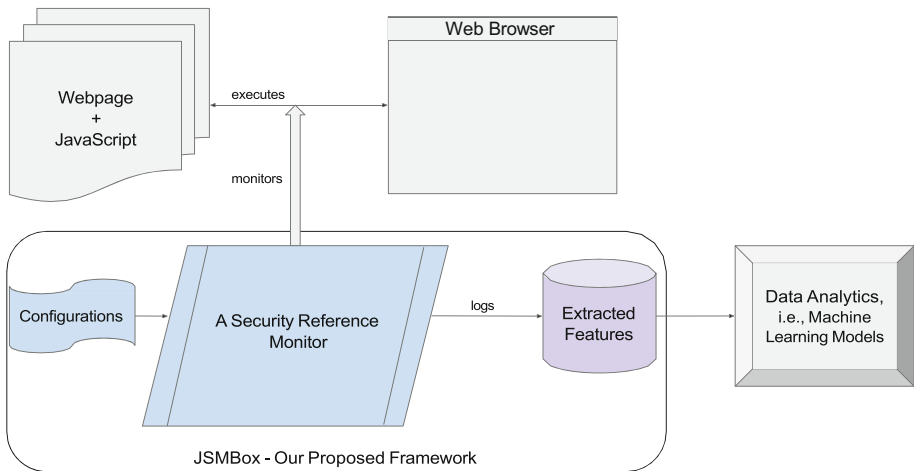


Fig. 1. Approach Overview

JavaScript code execution by intercepting its operations, such as property access and method calls, within the JavaScript execution environment. This method enables JSMBBox to conduct real-time, dynamic analysis of the code, extracting its execution trace for data engineering and machine learning models. Our primary objective is to

develop a robust and effective technique for analyzing malware capable of circumventing the sophisticated evasion methods employed by modern malicious webpages. In pursuit of this goal, we employ the traditional security reference monitor technique [11] to oversee code execution, implemented exclusively in JavaScript, thus providing a more dependable and holistic solution, addressing the limitations of existing static and dynamic analysis techniques. Furthermore, our approach is lightweight and platform-independent, allowing for flexible deployment and feature customization. To the best of our knowledge, no prior research has utilized the reference monitoring method in JavaScript code for dynamic malware analysis.

Figure 1 depicts the overview of our proposed framework JSMBBox. Within this framework, we incorporate a reference monitor, which runs before the browser loads and executes any other JavaScript code on a given webpage. This mechanism ensures the monitor maintains a unique and original reference to the intercepted JavaScript events, i.e., function calls or property accesses, implemented in the browser. This approach effectively preserves the original functionality of the webpage while mitigating potential detection techniques employed in evasive malicious JavaScript code [53]. The monitor utilizes configuration data, defining intercepted JavaScript events and the properties of extracted features to record and retain the code execution details, such as the frequency of event execution, in a log file. This log file is then employed as input for a machine-learning algorithm to classify the maliciousness of the code. We discuss key components of our behavioral sandbox approach below.

3.3 The Monitor Initialization and Protection

We developed the monitor using pure JavaScript as a library within an anonymous function to preserve local references to all original built-in methods that will be utilized later in the monitoring process, along with other behavioral events to be monitored. By encapsulating these references within an anonymous function, external code cannot access them since local variables are protected within an anonymous function. As the library is the first code to be executed in the browser, we have the advantage of safeguarding against potential malicious attempts to subvert these built-in methods or monitored functions. This mechanism empowers the monitor to regulate all subsequent JavaScript code execution, ensuring complete mediation. Moreover, we can define policies to detect and prevent malicious code that attempts to bypass the monitoring at runtime. These mechanisms make our approach tamper-proof [54] and shielded from evasive detection methods [53]. In addition, they allow us to adapt event monitoring and policies to tackle potential new evasion techniques over time.

To make our framework more flexible and customizable, we can define JavaScript execution events, such as function calls and property accesses, as well as behaviors like the call frequency or sequence, in a configuration file. The monitor will then load this file and create wrapper functions based on the information provided. We will demonstrate the initialization steps using pseudo-code in Listing 3.2.

```

1  (()=>{
2      let $builtins = {}; $builtins.__proto__ = null;
3
4      //code to store built-in references, i.e.,:
5      $builtins.Function.apply=Function.prototype.apply;
6      // other code (not included) to store built-in references ..
7
8      //code to load the configuration file
9      let monitored_methods = loadMethods();
10     let monitored_properties = loadProperties();
11
12     //main code (not included) to intercept and log execution events
13
14 })();

```

Listing 3.2. Pseudo-code demonstrating the monitor’s initialization steps

3.4 Intercepting Execution Events

We intercepted JavaScript native functions and properties of a global object, such as document, window to monitor their invocations and accesses, i.e., execution events. For every method call specified in the configuration file, we start by preserving the original reference and its aliases. This mechanism ensures that the monitor captures any existing prototype inheritance chain of the reference to prevent possible attacks in malicious code [54]. Semi-pseudo code in Listing 3.3 illustrates this interception process. For property accesses, e.g., document.cookie, we leverage the Object.defineProperty(..) standard API and define the handler functions whenever a property is accessed, i.e., read or write.

```

1  for each {object, method} in monitored_methods {
2      //... code (not included) find function corresponding to aliases
3
4      //keep the original reference:
5      var original_method = object[method];
6
7      //ensure that the stored original apply function will be invoked:
8      original_method.apply = $builtins.Function.apply;
9
10     //define the method:
11     object[method] =>() {
12         //log the event execution:
13         event_log(object, method);
14
15         //execute the event using the original reference:
16         return original_method.apply(this, arguments);
17     }
18 }

```

Listing 3.3. Semi-pseudo code illustrating the interception process

3.5 Implementation

Developed as a pure JavaScript library, our JSMBox framework can be injected into a website in multiple ways to monitor the JavaScript code execution on the site. For example, we can inject the library as the first script to be executed in a webpage using webpage instrumentation, a web proxy, or a browser extension/add-on. As a framework for JavaScript analysis, we implement JSM-Box as a browser extension so that we can effectively collect the logged data and automate the browser with our extension on a large-scale dataset. A browser extension or add-on is additional code that can be loaded into

a browser to modify and enhance its capability. Major browsers, including Chromium-based browsers, such as Google Chrome, Brave, Microsoft Edge, Opera, Vivaldi, and Firefox, support browser extensions or add-ons [55]. Since our main code is written in JavaScript, it should be deployable in any browser supporting extensions/add-ons. We implement and test our code in the Chromium browser, a widely used codebase in many other browsers. As noted in [55], Chromium-based browser extensions can run in Firefox with just a few changes. To ensure that our code is executed first before the browser loads a webpage and executes its JavaScript code, we place our code in the background script of the browser extension. As discussed in Sect. 3.2, we have confirmed this mechanism by performing experimental tests.

Before loading a webpage upon request, the browser executes our code, which will intercept defined methods and properties. When a webpage is loaded in the browser, any JavaScript event that triggers these methods and properties will invoke our code, which will log the event and then invoke the original reference. This mechanism ensures that our code is set to monitor the behaviors of scripts at runtime, capturing right from the moment a page begins to load. Since we monitor the code at runtime, potential performance overhead exists. While we have not studied the overhead in this work, prior work demonstrated that the JavaScript inlined reference monitor approach poses lightweight performance overhead [52, 56, 57].

Although hundreds of commonly used JavaScript method calls exist, not all are susceptible to malicious JavaScript [58, 59]. Monitoring an excessively broad range of events can introduce noise and increase system overhead. Noises in extracted features reduce the accuracy of machine learning-based detection. In our current implementation of JSMBBox, we have curated a selection of the most critical events with 59 method calls and property accesses. Benign JavaScript behaviors are selected based on the most commonly used by any website to maintain the dynamic nature of the website and keep it functioning. The malicious ones are selected based on their potential to be misused in web-based attacks, such as executing unauthorized code or scripts, e.g., `eval`, `window.open` for unwanted pop-up ads and `navigator.sendBeacon` can be used for unauthorized tracking. For instance, the `charAt(..)` method of `String` is considered susceptible to misuse as it can be employed to encode data or generate obfuscated code that is difficult to decipher, thereby facilitating evasion techniques. These methods were identified based on analysis of human-labeled malicious JavaScript code, used in conjunction with each other [60–66]. Table 1 lists the 12 selected representative misused function calls with their descriptions implemented in JSMBBox.

Our current JSMBBox prototype monitors each event execution, i.e., behavior, and accumulates its frequency within a session to log the data as features. The resultant counting is transformed into a feature vector $[a_1, a_2, a_3, a_4, a_5, \dots, a_{59}]$, where a_i denotes the frequency of the i -th behavior. As discussed in Sect. 3.2, our JSMBBox framework supports customization of input events and features. However, we leave this implementation prototype for future work.

4 Evaluation and Experimental Results

To evaluate our approach and the proposed JSMBBox framework, we consider two research questions:

Table 1. JavaScript behaviors, method calls or property accesses, and their potential misuse by malicious actors

Method/Property	Potential Misuse
charCodeAt	A method of the String object used to encode data or create obfuscated code that is not easily readable, aiding in evasion techniques
Uint8Array	To handle raw binary data which can be used in memory exploits, such as buffer overflows, or to process downloaded malicious payloads
Math.random	Domain generation algorithms (DGAs) use Math.random to generate a large number of domain names that malware can use to communicate without being easily blocked
document.cookie	A sensitive property that can accessed and manipulated to perform attacks such as session hijacking
toDataURL	A method of the HTMLCanvasElement object to capture the content of a canvas element, which can be used for browser fingerprinting or stealing information rendered on the canvas
document.write	A method to inject malicious content into a webpage, such as through XSS attacks
atob/btoa	Methods used to encode and decode base64 strings, which is commonly used in obfuscating payloads and command and control communications
document.createElement	A method used to dynamically create elements like script or iframe to load malicious code
window.location	A global property used to redirect users to malicious sites or manipulate page content for phishing or site spoofing
fromCharCode	A method of the String object likely used in obfuscation techniques to hide malicious code
eval	A global method that executes text as code, allowing for arbitrary JavaScript code execution
Image.src	A property that can be modified to exfiltrate data

RQ1 : How effective that JSMBBox captures monitored behaviors in the existence of sophisticated evasion techniques in modern malicious JavaScript?

RQ2 : What is the performance of JSMBBox as a dynamic analysis tool for malicious webpage classification in machine-learning models?

We present and discuss our evaluation and experimental results for each research question below.

4.1 Defeating JavaScript Evasion Techniques

To evaluate how our JSMBBox framework can deal with and defeat sophisticated evasion techniques, as discussed in Sect. 2.3, to monitor defined behaviors for analysis and data

extraction, we replicate these techniques in webpages. We load these webpages in the browser with our extension and observe that all behaviors hidden in obfuscation code or other advanced evasion techniques are captured by JSMBBox.

For instance, we simulate the fingerprinting attack by obfuscating this method as discussed in running example in Listing 3.1, where existing analysis approaches failed to capture [50, 51]. While running this attack in the browser, our JSMBBox framework can still track and log its execution.

```

1 #include <stdio.h>
2 #include <emscripten.h>
3 int main()
4 {
5     EM_ASM(alert('Alert from WebAssembly'));
6     return 0;
7 }

```

Listing 4.1. A C program invoking a JavaScript alert() method

Another notable example is the case of the WebAssembly evasion technique discussed earlier. To confirm our approach can capture this new technique, we develop a simple C program that invokes a JavaScript method, illustrated in Listing 4.1.

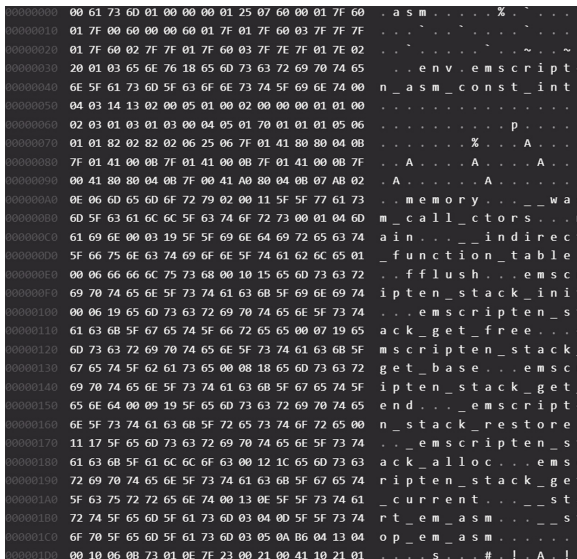


Fig. 2. WebAssembly binary code format (in a.wasm file) compiled from a C program

The C program is compiled into WebAssembly binary code (in a.wasm file, shown in Fig. 2) and embedded into a webpage. When the webpage is loaded in the browser with JSMBBox, the JavaScript method invoked from WebAssembly code is executed and logged by our framework, demonstrated in Fig. 3.

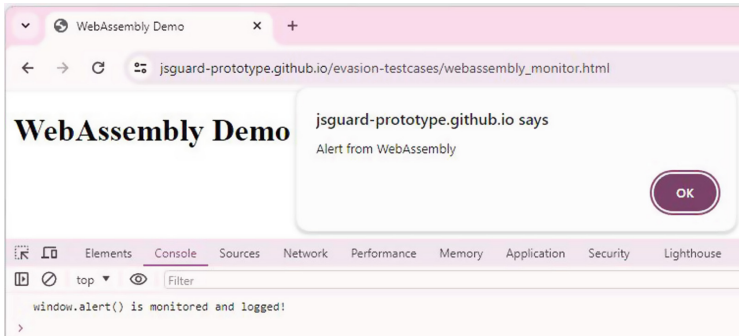


Fig. 3. Test Case: A JavaScript method called from WebAssembly is captured by JSMBBox

4.2 Maliciousness Classification

In this section, we will outline our experiments designed to assess the performance of our framework using machine learning models. Our experiments were conducted on a powerful CyberRange environment utilizing a virtual machine with Ubuntu 22.04 OS, 12 CPUs, 32 GB of RAM, and a 500 GB hard drive.

Dataset and feature extraction We collected a large number of malicious and benign website samples from two different datasets: the URLhaus database [67] for malicious websites, and the Tranco dataset [68] for benign websites. The URLhaus database, which is part of the Abuse.ch project, is well-known for its comprehensive collection of malicious URLs and is used by organizations such as the FBI, demonstrating its reliability. On the other hand, Tranco provides a strong website ranking by combining various data sources to ensure stability and resistance to manipulation, making it an excellent source of benign websites [68]. Our initial dataset consists of over 200,000 benign websites and over 200,000 malicious websites from these two sources. We maintain these websites in two lists to label and evaluate them separately.

For each website, we need to load it into the browser with our extension so that the JavaScript code can be executed and captured by our framework at run-time. To automate this process for large-scale datasets, we leverage Puppeteer², a Node.js library that allows us to control Chromium-based or Firefox browsers, which is particularly helpful for browser automation and data collection. We develop and run a script with a list of websites, launching a new browser instance for each one using Puppeteer, which is then set to load the browser extension. Once a website is fully loaded, our code checks for captured data and writes it into a CSV file labeled as malicious or benign. This process has resulted in a total of approximately 10,000 records from the malicious list, as well as a similar number of records from the benign list. The data from the two CSV files are combined to create extracted features for further analysis.

Machine-learning models We have utilized eight well-known machine learning models, which include Support Vector Machine (SVM), Logistic Regression, Naïve Bayes, K-Nearest Neighbors (KNN), Decision Tree, Random Forest, XG-Boost, and Ensemble

² <https://pptr.dev/>

methods [69, 70], to assess our collected dataset. Each model demonstrates different levels of accuracy in detecting malicious JavaScript, based on feature vectors extracted from the JavaScript code. Our comprehensive evaluation results enable users to choose the most suitable model for optimizing the detection process.

The metadata for our machine learning models can be found in Table 2. We have carefully fine-tuned specific hyperparameters as outlined in the second column of the table. One crucial aspect of this fine-tuning process involves optimizing model hyperparameters through the use of the GridSearchCV method [71, 72]. GridSearchCV conducts an exhaustive search over a specified parameter grid.

It trains the model on every combination of hyperparameters and selects the best combination based on cross-validation performance. This method performs a comprehensive search across a predefined grid of parameters, training the model with each parameter combination and identifying the optimal set based on cross-validation performance [71]. In addition to hyperparameter tuning, our training pipeline incorporates the Synthetic Minority Over-sampling Technique (SMOTE) [73] to address the class imbalance by generating synthetic samples within the feature space and enhancing model training effectiveness. We also use the Standard Scaler as a preprocessing step to standardize the features by removing the mean and scaling to unit variance. Additionally, we employ the Support Vector Classifier (SVC) [74, 75], an adaptation of the Support Vector Machine algorithm, for classification tasks.

Table 2. Differentiation of Machine Learning Models

Model Name	Hyperparameter Tuning	Pipeline Definition
SVM	Yes (GridSearchCV) C: 10000, gamma: 1	SMOTE, StandardScaler, SVC
Logistic Regression	Yes (GridSearchCV) C: [0.001, 0.01, 0.1, 1, 10, 100, 1000]	SMOTE, StandardScaler
GaussianNB	Yes (GridSearchCV) varsmoothing: [1e-9, 1e-8, 1e-7]	SMOTE, StandardScaler
KNN	Yes (GridSearchCV) n-neighbors: [3, 5, 7]	SMOTE, StandardScaler
Decision Tree	No	No
Random Forest	Yes (GridSearchCV) n-estimators: 300, min-samples-split: 2, max-depth: None	SMOTE
XGBoost	Yes (GridSearchCV) n-estimators: 200, max-depth: 15, learning-rate: 0.1, subsample: 1.0, colsample-bytree: 0.6	SMOTE, StandardScaler
Ensemble	Yes (GridSearchCV for individual models) same as for Random Forest and XGBoost	Voting Classifier (Random Forest and XGBoost)

JSMBBox’s effectiveness in classifying malicious JavaScript We have chosen specific performance metrics to directly address our research objectives, including accuracy, precision, recall, and F-scores [76, 77]. This methodology enables us to systematically categorize each JavaScript snippet, whether malicious or benign, into one of four potential outcomes. Using the classification of malicious snippets as an example: (1) Classified as malicious if it indeed contains malicious code, marking a true positive (TP) identification. (2) Classified as malicious erroneously when it is, in reality, benign, resulting in a false positive (FP). (3) Classified as benign mistakenly when it contains malicious code, leading to a false negative (FN). (4) Accurately classified as benign when it contains no malicious code, constituting a true negative (TN).

- **Accuracy:** the number of instances correctly classified over the total number of instances.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

- **Precision:** the number of instances correctly classified as malicious codes over all instances classified as malicious codes. It indicates the model’s effectiveness in correctly classifying content as malicious. A high-precision model excels in identifying malicious content.

$$P = \frac{TP}{TP + FP}$$

- **Recall:** the number of instances correctly classified as malicious codes over the total number of malicious codes. It reflects the model’s ability to correctly identify the actual positives, emphasizing its capacity to detect most of the malicious content.

$$R = \frac{TP}{TP + FN}$$

- **F-Score:** a composite measure of precision and recall for malicious code detection.

$$F(b) = \frac{2 * P * R}{P + R}$$

Table 3 shows the performance of eight machine learning models, with their effectiveness in spotting malicious JavaScript code scoring between 0.77 and 0.88, and for benign code, between 0.69 and 0.88, according to the F1-score. The Ensemble model shines by pinpointing malicious code with the greatest precision, 0.89. Meanwhile, the Random Forest model is the best at catching almost all malicious codes, achieving the highest recall of 0.88. When we look at the F1-score, which balances both precision and recall, Random Forest comes out on top for identifying malicious code, with the highest score of 0.88. Similarly, when finding benign code, both Random Forest and the Ensemble models are the best choices, each with top F1-scores of 0.88. The **accuracy** column provides an over- all measure of a model’s performance. High accuracy indicates that the model effectively distinguishes benign and malicious JavaScript content. As observed from the table, the Random Forest model achieved the highest accuracy of 0.88, closely followed by the Ensemble model at 0.87.

Table 3. Results of Models for Classifying JavaScript Content

Model	Benign			Malicious			Accuracy
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Score
SVM	0.97	0.64	0.77	0.73	0.98	0.84	0.81
Logistic Regression	0.985	0.57	0.72	0.70	0.99	0.82	0.78
Naïve Bayes	1.00	0.53	0.69	0.68	1.00	0.81	0.77
KNN	0.94	0.64	0.77	0.73	0.96	0.83	0.80
Decision Tree	0.97	0.68	0.80	0.76	0.98	0.85	0.83
Random Forest	0.88	0.88	0.88	0.88	0.88	0.88	0.88
XGBoost	0.92	0.78	0.84	0.81	0.93	0.87	0.86
Ensemble	0.86	0.90	0.88	0.89	0.85	0.87	0.87

5 Conclusion and Future Work

In this paper, we introduced JSMBBox, a novel behavioral sandbox approach designed to address the issues of analyzing and classifying malicious JavaScript code. Our method effectively addresses the limitations of traditional static and dynamic analysis techniques by monitoring and controlling JavaScript code behavior at runtime. By leveraging an inlined security reference monitor, our approach captures the behaviors of both static and dynamically generated code, including those employing advanced evasion techniques. We implemented JSM-Box as a runtime JavaScript analysis framework, which can monitor customizable events and their behaviors. The experimental results demonstrated the effectiveness and efficacy of our method, with machine learning models trained on features extracted by the framework achieving high accuracy rates, even when advanced evasion techniques are used to conceal malicious behaviors.

Future work will focus on enhancing the range of features extracted by the framework, including more sophisticated behaviors and different behavioral patterns. We will also investigate how to extend the implementation of JSMBBox to support multiple web browsers, ensuring its effectiveness and usability across different browsing environments. Additionally, we aim to develop a version of the framework that supports multiple web browsers or can be integrated directly into core browsers for more seamless and comprehensive monitoring. Experiments with a wider variety of datasets, including JavaScript files and newer web technologies like WebAssembly, will also be conducted to ensure the robustness and adaptability of our approach.

Acknowledgments. This work was partially supported by the Ohio Department of Higher Education (ODHE) through the Strategic Ohio Council for Higher Education (SOCHE) and Ohio Cyber Range Institute (OCRI) sub-awards and by the National Science Foundation (NSF) award CCF-2342355. We thank the anonymous reviewers for their insightful comments and suggestions for improving the manuscript.

References

1. Hu, C. et al.: Recent Trends in Internet Threats: Common Industries Impersonated in Phishing Attacks, Web Skimmer Analysis and More. Palo Alto Networks (2023). <https://unit42.paloaltonetworks.com/internet-threats-late-2022/>
2. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proceedings of the 19th International Conference on World Wide Web, pp. 281–290 (2010)
3. Aboaoja, F.A., Zainal, A., Ghaleb, F.A., Al-rimy, B.A.S., Eisa, T.A.E., Elnour, A.A.H.: Malware detection issues, challenges, and future directions: a survey. *Appl. Sci.* **12**(17), 8482 (2022)
4. Song, X., Chen, C., Cui, B., Fu, J.: Malicious JavaScript detection based on bidirectional LSTM model. *Appl. Sci.* **10**(10), 3440 (2020)
5. Xu, W., Zhang, F., Zhu, S.: The power of obfuscation techniques in malicious JavaScript code: a measurement study. In: 2012 7th International Conference on Malicious and Unwanted Software (MALWARE), pp. 9–16. IEEE (2012)
6. Jueckstock, J., Kapravelos, A.: VisibleV8: In-browser monitoring of javascript in the wild. In: Proceedings of the Internet Measurement Conference, pp. 393–405. IMC '19, Association for Computing Machinery, New York, NY, USA (2019)
7. Kim, H.C., Choi, Y.H., Lee, D.H.: Jssandbox: a framework for analyzing the behavior of malicious javascript code using internal function hooking. *KSII Trans. Internet Inf. Syst. (TIIS)* **6**(2), 766–783 (2012)
8. Gorji, A., Abadi, M.: Detecting obfuscated javascript malware using sequences of internal function calls. In: Proceedings of the 2014 ACM Southeast Regional Conference, pp. 1–6 (2014)
9. Romano, A., Lehmann, D., Pradel, M., Wang, W.: Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 1574–1589 (2022)
10. Cabrera-Arteaga, J., Monperrus, M., Toady, T., Baudry, B.: WebAssembly diversification for malware evasion. *Comput. Secur.* **131**, 103296 (2023)
11. Anderson, J.P.: Computer security technology planning study. Technical report, ESDTR-73-51. Anderson (James P) And Co Fort Washington Pa Fort Washington (1972)
12. Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, Inc (2008)
13. Fang, Y., Huang, C., Su, Y., Qiu, Y.: Detecting malicious javascript code based on semantic analysis. *Comput. Secur.* **93**, 101764 (2020)
14. Wang, R., Zhu, Y., Tan, J., Zhou, B.: Detection of malicious web pages based on hybrid analysis. *J. Inf. Secur. Appl.* **35**, 68–74 (2017)
15. Ren, K., Qiang, W., Wu, Y., Zhou, Y., Zou, D., Jin, H.: An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1420–1432 (2023)
16. Howard, F.: Malware with Your Mocha? Obfuscation and Anti Emulation Tricks in Malicious JavaScript (2023). Online: <https://www.yumpu.com/s/0PX6x19R5gw0KWvt>
17. Kaplan, S., Livshits, B., Zorn, B., Siefert, C., Curtsinger, C.: NOFUS: Automatically Detecting+ String.fromCharCode(32)+ ObfuscateD.toLowerCase()+ JavaScript Code. Technical report, MSR-TR 2011-57, Microsoft Research (2011)
18. Xu, W., Zhang, F., Zhu, S.: JStill: mostly static detection of obfuscated malicious JavaScript code. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 117–128 (2013)
19. Malware, B.: Obfuscation: The Hidden Malware. IEEE Security & Privacy (2011)

20. Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A.: Formal methods meet mobile code obfuscation identification of code reordering technique. In: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 263–268. IEEE (2017)
21. Wang, X., Zhang, Y., Zhao, L., Chen, X.: Dead code detection method based on program slicing. In: 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 155–158. IEEE (2017)
22. Kılıç, H., Katal, N.S., Selçuk, A.A.: Evasion techniques efficiency over the ips/ids technology. In: 2019 4th International Conference on Computer Science and Engineering (UBMK), pp. 542–547. IEEE (2019)
23. Luoma-aho, M.: Analysis of Modern Malware: obfuscation techniques. Master's thesis, JAMK University of Applied Sciences (2023)
24. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300. IEEE (2010)
25. Heiderich, M., Nava, E.A.V., Heyes, G., Lindsay, D.: Web Application Obfuscation: WAFs.. evasion.. filters//alert (/obfuscation/)-'. Elsevier (2010)
26. Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., Liu, X.: Research on webassembly runtimes: a survey (2024). [arXiv:2404.12621](https://arxiv.org/abs/2404.12621)
27. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: a survey. *ACM Trans. Web (TWEB)* **14**(2), 1–33 (2020)
28. Zhang, D., Zhang, J., Bu, Y., Chen, B., Sun, C., Wang, T., et al.: A survey of browser fingerprint research and application. *Wirel. Commun. Mobile Comput.* **2022** (2022)
29. Iqbal, U., Englehardt, S., Shafiq, Z.: Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 1143–1161. IEEE (2021)
30. Kim, K., Kim, I.L., Kim, C.H., Kwon, Y., Zheng, Y., Zhang, X., Xu, D.: J-force: forced execution on javascript. In: Proceedings of the 26th international conference on World Wide Web, pp. 897–906 (2017)
31. Fass, A., Krawczyk, R.P., Backes, M., Stock, B.: JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, Proceedings 15, pp. 303–325. Springer (2018)
32. Ndichu, S., Kim, S., Ozawa, S., Misu, T., Makishima, K.: A machine learning approach to detection of javascript-based attacks using ast features and paragraph vectors. *Appl. Soft Comput.* **84**, 105721 (2019)
33. Fass, A., Backes, M., Stock, B.: JStap: a static pre-filter for malicious JavaScript detection. In: Proceedings of the 35th Annual Computer Security Applications Conference, pp. 257–269 (2019)
34. Fang, Y., Huang, C., Zeng, M., Zhao, Z., Huang, C.: JStrong: malicious JavaScript detection based on code semantic representation and graph neural network. *Comput. Secur.* **118**, 102715 (2022)
35. Tellenbach, B., Paganoni, S., Rennhard, M.: Detecting obfuscated javascripts from known and unknown obfuscators using machine learning. *Int. J. Adv. Secur.* **9**(3/4), 196–206 (2016)
36. Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C.: ZOZZLE: fast and precise in-browser javascript malware detection. In: USENIX Security Symposium, pp. 33–48. San Francisco (2011)
37. Jagpal, N., Dingle, E., Gravel, J.P., Mavrommatis, P., Provos, N., Rajab, M.A., Thomas, K.: Trends and lessons from three years fighting malicious extensions. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 579–593 (2015)

38. Xue, Y., Wang, J., Liu, Y., Xiao, H., Sun, J., Chandramohan, M.: Detection and classification of malicious JavaScript via attack behavior modelling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 48–59 (2015)
39. Kim, H.G., Kim, D.J., Cho, S.J., Park, M., Park, M.: Efficient detection of malicious web pages using high-interaction client honeypots. *J. Inf. Sci. Eng.* **28**(5), 911–924 (2012)
40. Aloferer, Y., Rana, O.: Honeyware: a web-based low interaction client honeypot. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, pp. 410–417. IEEE (2010)
41. Snyder, P., Ansari, L., Taylor, C., Kanich, C.: Browser feature usage on the modern web. In: Proceedings of the 2016 Internet Measurement Conference, pp. 97–110 (2016)
42. Roesner, F., Kohno, T., Wetherall, D.: Detecting and defending against third-party tracking on the web. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp. 155–168 (2012)
43. Yagemann, C., Sultana, S., Chen, L., Lee, W.: Barnum: Detecting document malware via control flow anomalies in hardware traces. In: Proceedings of 22nd International Information Security Conference (ISC 2019), pp. 341–359. Springer, Berlin, Heidelberg (2019)
44. Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: Nozzle: A defense against heap spraying code injection attacks. In: USENIX Security Symposium, pp. 169–186 (2009)
45. Li, B., Vadrevu, P., Lee, K.H., Perdisci, R., Liu, J., Rahbarinia, B., Li, K., Antonakakis, M.: JSgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In: Network and Distributed Systems Security (NDSS) Symposium (2018)
46. Fang, Y., Huang, C., Liu, L., Xue, M.: Research on malicious JavaScript detection technology based on LSTM. *IEEE Access* **6**, 59118–59125 (2018)
47. Neasbitt, C., Li, B., Perdisci, R., Lu, L., Singh, K., Li, K.: WebCapsule: towards a lightweight forensic engine for web browsers. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 133–145. CCS '15, Association for Computing Machinery, New York, NY, USA (2015)
48. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 31–39 (2010)
49. Wang, Y., Cai, W.d., Wei, P.C.: A deep learning approach for detecting malicious JavaScript code. *Secur. Commun. Netw.* **9**(11), 1520–1534 (2016)
50. Obidat, M.A., Obeidat, S., Holst, J., Lee, T.: Canvas deceiver—a new defense mechanism against canvas fingerprinting. *J. Syst. Cybern. Inf.* **18**(6) (2020)
51. Canvas fingerprinting: What is it and how to bypass it (2023). Online: <https://www.zenrows.com/blog/canvas-fingerprinting#what-is>
52. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, pp. 47–60 (2009)
53. Sarker, S., Jueckstock, J., Kapravelos, A.: Hiding in plain site: detecting javascript obfuscation through concealed browser API usage. In: Proceedings of the ACM Internet Measurement Conference (IMC) (2020)
54. Magazinius, J., Phung, P.H., Sands, D.: Safe wrappers and sane policies for self-protecting JavaScript. In: Information Security Technology for Applications: 15th Nordic Conference on Secure IT Systems, NordSec 2010, pp. 239–255. Springer, Espoo, Finland
55. Mozilla: Browser extensions (2024). Online: <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions>
56. Phung, P.H., Monshizadeh, M., Sridhar, M., Hamlen, K.W., Venkatakrishnan, V.: Between worlds: securing mixed javascript/actionsript multi-party web content. *IEEE Trans. Dependable Secure Comput.* **12**(4), 443–457 (2015)

57. Phung, P.H., Pham, H.D., Armentrout, J., Hiremath, P.N., Tran-Minh, Q.: A user-oriented approach and tool for security and privacy protection on the web. *SN Comput. Sci.* **1**(4) (2020)
58. MDN contributors: Web APIs (2023). Online: <https://developer.mozilla.org/en-US/docs/Web/API>
59. Shehoze Farooqi, Billy Melicher, B.K., Starov, A.: Malicious JavaScript Injection Campaign Infects 51k Websites (2023). <https://unit42.paloaltonetworks.com/malicious-javascript-injection/>
60. Breaking down NOBELIUM's latest early-stage toolset (2021). Online: <https://www.microsoft.com/en-us/security/blog/2021/05/28/breaking-down-nobeliums-latest-early-stage-toolset/>
61. HTML smuggling explained (2018). Online: <https://www.outflank.nl/blog/2018/08/14/html-smuggling-explained/>
62. Dynamic resolution: Domain generation algorithms (2022). Online <https://www.attack.mitre.org/techniques/T1568/002/>
63. HTML Smuggling: Recent observations of threat actor techniques (2023). Online on <https://blog.delivr.to>, shorten URL: <https://tinyurl.com/yck279xb>
64. Attackers Turn to SVG Files to Distribute QBot Malware (2022). Online on <https://www.tanium.com>, shorten URL: <https://tinyurl.com/mrx2udtr>
65. JavaScript invokes ms-appinstaller handler from malicious landing page at time of user click (2023). Online: <https://www.microsoft.com/en-us/security/blog/2023/12/28/financially-motivated-threat-actors-misusing-app-installer/>
66. Windows malware categories and families (2024). Online: <https://portal.av-atlas.org/malware/statistics>
67. URLhaus database-malicious URLs that are being used for malware distribution (2024). Online: <https://urlhaus.abuse.ch/>
68. Pochat, V.L., Van Goethem, T., Tajalizadehkhooob, S., Korczyn'ski, M., Joosen, W.: Tranco: A research-oriented top sites ranking hardened against manipulation (2018). *arXiv:1806.01156*
69. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious JavaScript detection using classification techniques. In: 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), pp. 47–54 (2009)
70. Liu, Q., Li, P., Zhao, W., Cai, W., Yu, S., Leung, V.C.M.: A survey on security threats and defensive techniques of machine learning: a data driven view. *IEEE Access* **6**, 12103–12117 (2018)
71. GridSearchCV (2023). Online: <https://www.analyticsvidhya.com/blog/2021/06/tune-hyperparameters-with-gridsearchcv/>
72. Manan, W.N.W., Han, C.Y.: Detection of distributed denial-of-service (ddos) attack with hyperparameter tuning based on machine learning approach. In: 2023 7th International Symposium on Innovative Approaches in Smart Technologies (ISAS), pp. 1–8 (2023)
73. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **16**, 321–357 (2002)
74. Support vector classification (2023). Online: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
75. Support vector machines (SVMs) (2024). Online: <https://scikit-learn.org/stable/modules/svm.html>
76. Wang, W.H., Lv, Y.J., Chen, H.B., Fang, Z.L.: A static malicious javascript detection using SVM. In: Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013), pp. 214–217. Atlantis Press (2013)
77. Zhu, N., Zhao, G., Yang, Y., Yang, H., Liu, Z.: AEC GAN: unbalanced data processing decision-making in network attacks based on ACGAN and machine learning. *IEEE Access* **11**, 52452–52465 (2023)