Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment

Jiashui Wang[*]
Zhejiang University & Ant Group
Hang Zhou, China
12221251@zju.edu.cn

Peng Qian[*†]
Zhejiang University
Hang Zhou, China
pqian@zju.edu.cn

Xilin Huang
Ant Group
Hang Zhou, China
huangxilin.hxl@antgroup.com

Xinlei Ying
Ant Group
Hang Zhou, China
xinlei.yxl@antgroup.com

Yan Chen[†]
Northwestern University
Evanston, USA
ychen@northwestern.edu

Shouling Ji
Zhejiang University
Hang Zhou, China
sji@zju.edu.cn

Jianhai Chen
Zhejiang University
Hang Zhou, China
chenjh919@zju.edu.cn

Jundong Xie
Ant Group
Hang Zhou, China
jundong.xjd@antgroup.com

Long Liu
Ant Group
Hang Zhou, China
ll280345@antgroup.com

Abstract

Browsers are responsible for managing and interpreting the diverse data coming from the web. Despite the considerable efforts of developers, however, it is nearly impossible to completely eliminate potential vulnerabilities in such complicated software. While a family of fuzzing techniques has been proposed to detect flaws in web browsers, they still face the inherent challenge of generating test inputs with low semantic correctness and poor diversity.

In this paper, we propose TACOMA, a novel fuzzing framework tailored for web browsers. TACOMA comprises three main modules: a semantic parser, a semantic aligner, and an input generator. By taking advantage of fine-grained semantic alignment techniques, TACOMA is capable of generating semantically correct test inputs, which significantly improve the probability of a fuzzer in triggering a deep browser state. In particular, by integrating a scope-aware strategy into input generation, TACOMA is able to deal with asynchronous code generation, thereby substantially increasing the diversity of the generated test inputs. We conduct extensive experiments to evaluate TACOMA on three production-level browsers, i.e., Chromium, Safari, and Firefox. Empirical results demonstrate that TACOMA outperforms state-of-the-art browser fuzzers in both achieving code coverage and detecting unique crashes. So far, TACOMA has identified 32 previously unknown bugs, 10 of which have been assigned CVEs. It is worth noting that TACOMA unearthed two bugs in Chromium that have remained undetected for ten years.

[*]Jiashui Wang and Peng Qian are the co-first authors.
[†]Peng Qian and Yan Chen are the co-corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680351

CCS Concepts

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

Keywords

Browser, Fuzzing, Semantic Alignment, Vulnerability Detection

ACM Reference Format:
Jiashui Wang, Peng Qian, Xilin Huang, Xinlei Ying, Yan Chen, Shouling Ji, Jianhai Chen, Jundong Xie, and Long Liu. 2024. Tacoma: Enhanced Browser Fuzzing with Fine-Grained Semantic Alignment. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680351

1 Introduction

As one of the most influential software applications, web browsers play a key role in facilitating the access to web-based content. They adeptly render a variety of resources, including HTML, CSS, and JavaScript, allowing users to engage with a wide range of online content. Nevertheless, developing a flawless web browser - complete with compilers, interpreters, and parsers - to handle the multifarious data from the Internet is an extremely challenging task. The intricacies involved make it nearly impossible to completely eliminate defects from the sophisticated software.

Nowadays, browser security has become a focal point for both academia and industry due to the increasing prominence of browser bugs [3, 34, 45]. These flaws have led to severe consequences, exposing security threats to individuals, organizations, and the broader Internet ecosystem. The *Chrome Threat Intelligence Team* has disclosed a total of 12 exploited bugs in the last two years. For example, in April 2023, a zero-day vulnerability (CVE-ID: CVE-2023-2033) was found in Chrome. This bug, residing in the JavaScript engine, empowers attackers to crash the browser using a specially crafted web page and read or write memory data beyond the buffer limit. More recently, an information disclosure bug (CVE-ID: CVE-2023-4357) was found in Chrome, which enables a remote attacker to steal a user's local sensitive information by tricking the user into visiting a crafted HTML page. Such occurrences are not isolated incidents; rather, browser bugs are regularly discovered and exploited every few months, posing a significant threat to the entire web ecology. Undoubtedly, it is crucial to detect browser flaws as early as possible to prevent them from being exploited by attackers.

Fuzzing, recognized as a highly promising technique in software testing, has shown considerable potential in bug detection [10, 19, 23–25, 28]. A prevailing trend among numerous browser vendors is to apply fuzzing techniques to dig up the vulnerabilities hidden in browsers [47]. Notably, browser fuzzing can be broadly classified into several main domains, including syntax fuzzing, protocol fuzzing, document object model (DOM) fuzzing, and JavaScript engine fuzzing [8, 17, 35, 42, 48]. Among these, JavaScript engine fuzzing has proven to be a highly effective method for triggering crashes in web browsers, as evidenced by the development of several JavaScript fuzzers in recent years [13, 15, 18, 27]. However, despite extensive efforts directed at JavaScript engines, the associated binding code has been largely overlooked. This oversight can be attributed to the complicated implementation of the binding layer within JavaScript engines. While a recent advancement [8] has taken advantage of semantic information to inspect the binding code, it still has difficulties in generating test inputs without syntactic errors, and thus could only explore a limited set of browser backend logic. Therefore, there is an urgent need to develop a browser fuzzer that is tailored to effectively examine the JavaScript engine and pinpoint bugs within its binding code.

However, navigating the binding code introduces significant challenges since the JavaScript binding layer involves complex logic that enables interactions between JavaScript and other programming languages. Effective fuzzing at this layer requires a deep understanding of the underlying language interoperability, including data type conversion and memory management. Fuzzing, in this context, requires rigorous attention to ensure correct interaction between different languages, coupled with the challenge of generating semantically correct JavaScript code, which increases the difficulty of creating effective test inputs.

Upon scrutinizing existing browser fuzzers [8, 9, 42, 47], we found that, due to the lack of semantic alignment, they still face the following problems. (1) Current browser fuzzers exhibit remarkable room for improvement in terms of both the semantic correctness and the diversity of test inputs. They fundamentally rely on manually defined rules for input generation, resulting in the failure of the generated test inputs to explore a wide range of browser states. More importantly, in the absence of semantic correction, the generated input may contain semantic errors that trigger browser runtime errors, thus hindering fuzzing from reaching deep browser state. (2) They expose inadequate support for the binding code. This deficiency is evident in the limited control over parameters, scope, asynchronous operations, and other critical elements, making it difficult to generate effective inputs for the binding layer. (3) Existing browser fuzzers do not take into account different execution contexts, confining their scope to the `Window` execution context scenario. This oversight excludes other critical execution contexts such as `Worker` and `Worklet`, limiting the scalability of browser fuzzing.

To tackle the aforementioned challenges, we present Tacoma, a novel fuzzing framework for web browsers. Specifically, Tacoma consists of three main components: (1) First, we design a semantic parsing module, which extracts and analyzes the semantic information within browsers. Note that the browser semantic information is extracted via a parser based on WebIDL [37]. Browsers use WebIDL to describe the interfaces of the binding layer. Parsing the WebIDL provides most of the semantic information required to generate an effective JavaScript code. (2) Second, we propose a semantic alignment module that aims to complement and rectify the semantic information extracted by the parser, namely to correct the WebIDL definitions and make the expressed semantic information consistent with the implementation of the browser source code. (3) We introduce an input generation module composed of a JavaScript code generator and an HTML document producer. This module is used to generate test inputs for the browser fuzzer.

We perform extensive experiments on three mainstream browsers, i.e., Chromium, Safari, and Firefox. During the evaluation, Tacoma has successfully found 32 previously unknown bugs, to which 10 CVEs have been assigned. Tacoma significantly outperforms the state-of-the-art browser fuzzers, in terms of both the achieved branch coverage and the number of detected bugs. For example, Tacoma achieves 17.49% branch coverage in Chromium over a 24-hour run period, 8.38% more than the state-of-the-art fuzzer Favocado [8]. More importantly, Tacoma identifies a use-after-free bug (CVE-ID: CVE-2023-5996) in Chromium, which has been introduced for ten years. Notably, this bug has been exploited [6].

Main Contributions. To summarize, our key contributions are:

- We propose a semantic alignment technique to guide a browser fuzzer in generating semantically correct and diverse test inputs, thereby increasing the probability of triggering vulnerabilities in the JavaScript binding code. Specifically, the semantic alignment module furnishes a set of solutions to optimize the generated test inputs, including scope-aware mechanism, context-aware strategy, and cross-execution-context navigation scheme.
- We design and implement a novel browser fuzzing framework Tacoma, which consists of three key components, namely semantic parser, semantic aligner, and input generator.
- We evaluate Tacoma on three mainstream browsers, i.e., Chromium, Safari, and Firefox. Tacoma achieves higher branch coverage and triggers more unique crashes than state-of-the-art browser fuzzers. It has detected 32 previously unknown bugs, out of which 10 have been assigned CVEs so far.

2 Background

2.1 Fuzzing on Web Browser

Web browsers serve as critical gateways for accessing and rendering Internet content. However, they can also be vulnerable to attackers who seek to exploit potential weaknesses for malicious purposes. Browser fuzzing [20], a software testing technique, aims to uncover security flaws in browsers. It identifies those vulnerabilities by subjecting browsers to systematic testing with diverse inputs.

In the context of browser fuzzing, the goal is to generate various and unexpected inputs for browsers, including malformed HTML, JavaScript, or other web content, to discover and exploit potential security vulnerabilities. This allows security researchers and developers to effectively address the weaknesses that may be

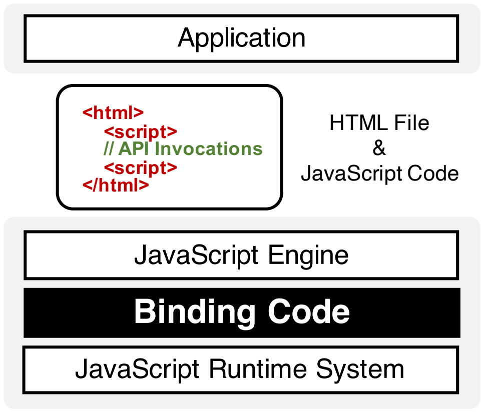


Figure 1: Binding code is used to extend the functionality of JavaScript by providing a bridge between the JavaScipt application code and the runtime system.

exploited by malicious actors. For example, FreeDom [42] is a state-of-the-art browser fuzzer that enhances the fuzzing effectiveness by categorizing browser APIs into various fuzzing operations based on their functionalities. Recently, Favocado [8] has been proposed to identify vulnerabilities in the JavaScript binding code. Despite these advancements, it is noteworthy that current browser fuzzers encounter challenges in detecting vulnerabilities within the complicated semantics of the JavaScript binding layer.

2.2 JavaScript Binding Code

JavaScript is a high-level programming language that is interpreted by JavaScript engines. Given its high-level nature, JavaScript relies on the runtime systems to implement low-level functionality, such as memory management and file access. These underlying functions are typically written in low-level and unsafe languages. To take advantage of such functionality, JavaScript runtime systems incorporate the binding code, which is an integral native component of JavaScript engines, as shown in Figure 1. Due to the different approaches to typing, memory management, and exception handling between the high-level and low-level languages, direct invocation of runtime routines by the scripting code is not feasible [5]. Instead, the scripting code calls upon the binding code to extend the functionality of JavaScript by facilitating the translation of data representations between different languages.

Binding code is used to establish the necessary data type mappings between JavaScript and low-level code. Native functions in the binding code provide JavaScript objects by defining them using an interface definition language (IDL). When JavaScript creates such objects, the binding code dynamically generates the required native data formats and types them with JavaScript variables. Scripts can then invoke native functions and manipulate data from native components. However, the translation process does not fully interpret the type and memory safety features inherent in JavaScript. Binding code, implemented in low-level and unsafe languages, is susceptible to vulnerabilities that are not uncommon. In practice, writing binding code is inherently complex and prone to failure at different stages. It is critical to accurately detect defects in the binding code and effectively communicate any errors back to JavaScript.

```
1  <template>
2     <script id = "audioworklet">    // Audioworklet execution context
3        try {
4           var Class_0 = class extends AudioWorkletProcessor {
5              process(inputs, outputs, parameters) {
6                 try { var var_1 = true } catch (e) {};
7                 return var_1;
8              }
9              static get parameterDescriptors() {
10                try { var var_3  = { name : "AAA" } } catch (e) {};
11                try { var var_2  = { var_3 } } catch (e) {};
12                return var_2;
13             }
14          }
15       } catch (e) {};
16       try { var var_4 = registerProcessor("Class_0", Class_0) } catch (e) {};
17    </script>
18 </template>
19 <script>    // Window execution context
20     let template = document.querySelector("template");
21     let audioWorkletSource = URL.createObjectURL(new Bob(
22        [template.content.querySelector("#audioworklet").textContent], { type: "text/javascript" } ));
23     try { var var_5 = new AudioContext() } catch (e) {};   // audio context
24     try { var var_6 = new AudioContext() } catch (e) {};   // audio context
25     try {
26        var resolve_7 = function () {
27           try { var var_8 = new AudioWorkletNode(var_5, "Class_0") } catch (e) {};
28           try { var var_9 = var_8.connect(var_5.destination) } catch (e) {};
29        }
30     } catch (e) {};
31     try { var var_10 = var_5.audioWorklet.addModule(audioWorkletSource).then(resolve_7); } catch (e) {};
32     try { var var_11 = var_6.createScriptProcessor(512, 8) } catch (e) {};
33     try {
34        var audioprocess_12 = function (arg_13)  {
35           try { var var_15 = var_5.createBufferSource() } catch (e) {};
36           try { var var_16 = var_15.connect(var_5.destination, 0, 0) } catch (e) {};
37           try { var var_17 = var_11.connect(var_6.listener.positionZ, 0) } catch (e) {};
38           try { var var_20 = var_11.connect(var_19.gain, 0) } catch (e) {};
39        }
40     } catch (e) {};
41     try { var var_14 = var_11.addEventListener("audioprocess", audioprocess_12) } catch (e) {};
42     try { var var_18 = var_11.connect(var_6.destination, 0, 0) } catch (e) {};
43     try { var var_19 = var_6.createGain() } catch (e) {};
44 </script>
```

Figure 2: A JavaScript binding code sample that is difficult to generate with existing browser fuzzers.

Several efforts [5, 8, 41] have been proposed to detect vulnerabilities in the binding code, focusing on the issues arising from the absence of necessary checks, violations of data translation rules, and mishandling of exceptions within the binding code. These works employ various static analysis techniques and fuzzing strategies. However, their applicability and scope are limited. Static analysis may not comprehensively capture vulnerabilities due to the dynamism and diversity of the JavaScript binding code, limiting its effectiveness to the identified scope. Conversely, while fuzzing has proven its practicality in uncovering browser vulnerabilities, current JavaScript fuzzers face challenges when applied to fuzzing binding code. Fuzzing requires a profound understanding of the interaction between JavaScript and the underlying language, along with the creation of test inputs that achieve high code coverage. This task becomes increasingly difficult in the context of complex applications and extensive code libraries.

3 Overview

In this section, we begin by providing a motivating example to highlight the key challenges of generating complex JavaScript code. We then discuss the limitations of current browser fuzzers and show that they fail to generate the test input in the example. Finally, we present our strategy and show how it copes with the example.

3.1 Motivating Example

Considering an example in Figure 2, we present the JavaScript binding code. This example comprises two audio contexts (lines 23-24)

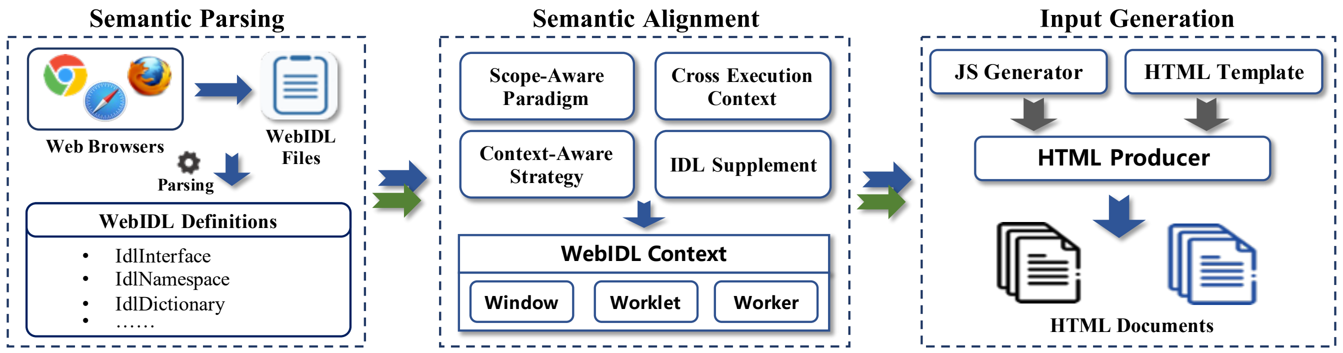


Figure 3: The overall architecture of Tacoma, which mainly consists of three components. (1) A semantic parser aims to capture the WebIDL definitions from web browsers. (2) A semantic aligner is designed to complement and correct the semantic information. (3) An input generator is used to produce the HTML document input for the browser fuzzer.

and a collection of audio nodes, which are responsible for rendering audio data in the browser. In addition, it involves two different execution context semantics, namely `Window` and `AudioWorklet`. It is worth mentioning that current browser fuzzers encounter significant difficulties when tasked with generating test inputs akin to this example. In the following, we summarize three major challenges faced by existing browser fuzzing approaches.

Challenge #1. Current browser fuzzers struggle to effectively manage scopes, especially in asynchronous code generation, leading to a limited diversity in generating test inputs. For example, one line of work, exemplified by Domato [9], relies on manually defined rules for input generation, but lacks a scope management strategy during the process, making it unable to generate asynchronous code, such as lines 33-40 in Figure 2. Another line of work, exemplified by Favocado [8], adopts a recursive generation strategy based on semantic dependencies. While this method takes into account the scenario where variables are situated in different scopes, its scope management remains constrained. For example, after completing code generation within a scope, Favocado loses scope information upon exiting the scope, which hinders its ability to continue generating code within that scope. As depicted in Figure 2, the generation of the statement on line 38 depends on the information of `var_19`. When Favocado leaves the scope `audioprocess_12` to create `var_19` (line 43) after generating the first three lines of code (lines 35-37) within that scope, it will lose the scope information of `audioprocess_12`, and thus fail to generate the subsequent code within scope `audioprocess_12`, i.e., line 38. In summary, browsers contain complex and diverse scopes, and existing fuzzers cannot effectively capture information across different scopes, resulting in generated test inputs that are prone to triggering browser crashes.

Challenge #2. Recent advancements aim to address the challenges of input generation by exploring rule rectification methods. For instance, SaGe [47] dynamically adjusts the generation rules of the fuzzer based on abnormal state feedback observed during browser runtime. However, certain semantic errors are difficult to correct automatically via runtime state feedback. As shown in Figure 2, where two audio contexts are generated (lines 23-24), a potential issue arises when an audio node (e.g., `var_15`) tries to connect to another audio node (e.g., `var_16`) using the `connect` function. The browser checks whether the two audio nodes belong to the same audio context. If not, an exception is triggered. It is noteworthy that in such scenarios, SaGe encounters limitations as it struggles to capture contextual semantics, and thus cannot conclusively determine which audio nodes belong to the same audio context. Consequently, the test inputs generated by SaGe could lead to exceptions.

Challenge #3. While existing browser fuzzers are adept at generating inputs within the `Window` execution context, they often struggle with handling semantic information across other different execution contexts, such as `Worklet`, `ServiceWorker`, and `DedicatedWorker`. For example, current browser fuzzers face significant challenges when generating the JavaScript code that involves scenarios like the one depicted in Figure 2, where two different execution contexts - `Window` and `AudioWorklet` - are present. Indeed, enabling a fuzzer to generate the test inputs involving different execution contexts requires a comprehensive understanding of browser semantics, which is undoubtedly difficult for prevailing methods.

3.2 Our Strategy

To tackle the above challenges, we integrate a semantic alignment mechanism into Tacoma to improve both correctness and diversity of the generated test inputs. This module embraces three key strategies. (1) We propose a scope-aware mechanism that allows the generator to generate asynchronous code in different scopes guided by a scope tree. In addition, a scope pending pool is specifically designed to ensure that the generator can automatically find the appropriate scope for code generation without introducing additional redundant code. (2) We design a context-aware strategy to dynamically correct semantic information. This strategy is capable of tracking the relationships between variables during code generation, thereby ensuring the accurate recognition of variable mapping relationships. (3) We introduce a cross-execution-context navigation scheme that facilitates code generation across different execution contexts. Specifically, we make it possible to switch between different execution contexts for code generation at any point during input generation by independently storing the semantics of different execution contexts. Furthermore, the code generated in different execution contexts can share state information, allowing for the generation of more complex inputs that more accurately reflect real-world application scenarios. To the best of our knowledge, Tacoma represents the first initiative to effectively enable the code generation across different execution contexts, marking a significant advancement in the field of browser fuzzing.

4 Design

In this section, we present the technical details of the proposed fuzzing framework. Figure 3 illustrates the overall architecture of Tacoma, which consists of three main components: (i) a semantic parsing module, which is capable of extracting and analyzing the semantic information of web browsers. Specifically, the parser extracts semantics by capturing the WebIDL definitions from browsers. (ii) a semantic alignment module, which is designed to complement and rectify the semantic information under the guidance of advanced correction strategies. This module aims to correct the WebIDL definitions so that the expressed semantic information is consistent with the implementation in the browser source code. Notably, corrected WebIDL definitions are stored in different execution contexts. As such, the JavaScript generator can decide to select which WebIDL context to obtain semantic information based on the execution context. (iii) an input generation module, consisting of a JavaScript code generator and an HTML document producer, which work together to generate test inputs for the browser fuzzer.

4.1 Semantic Parsing

To ensure effective input generation for the browser fuzzer, a critical step is to extract accurate semantic information from the web browser. Towards this, we develop a WebIDL parser specifically designed to analyze the syntax of WebIDL files. Note that WebIDL is commonly used to define the JavaScript binding layer interface implemented by a browser [33]. By parsing the semantics of WebIDL, we can obtain a comprehensive set of information necessary for generating JavaScript code, including but not limited to object creation methods, object dependencies, and callable methods associated with objects. An advantage of using WebIDL for this purpose is its inherent connection to the browser source code. As the content of WebIDL is automatically synchronized with browser source code upgrades, it ensures that the semantic information remains up-to-date. As such, when generating test inputs for a new version of a browser, there is no need for manual intervention to add interface semantic information. Instead, a straightforward update to the WebIDL file is sufficient, streamlining the process and maintaining accuracy in line with the evolving browser codebase.

To be specific, the WebIDL parser aims to capture the WebIDL definitions from web browsers. We implement a parsing framework IDLark[1], based on *lark* which is a Python parsing toolkit. IDLark parses the WebIDL file and translates various definitions in the file into the corresponding Python objects. Tacoma then performs semantic alignment and input generation based on these Python objects. As an example, the interface definition in WebIDL is parsed as follows:

$$\begin{gathered} interface < name > \{< attributes >< operations >\} \\ \Big\downarrow \text{IDLark} \\ class\ IdlInterface\{name, attributes[\,], operations[\,]\} \end{gathered}$$

[1]Code is available at https://github.com/Messi-Q/IDLark

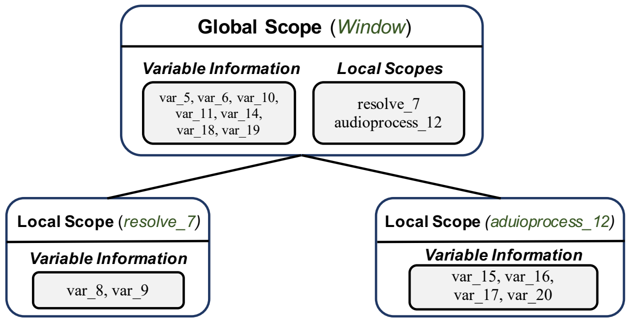


Figure 4: The scope tree (simplified version) extracted from the example in Figure 2.

4.2 Semantic Alignment

It is important to emphasize that the original intent of the WebIDL design is to establish a broad standard without specifying specific code implementations. However, relying solely on the original WebIDL semantics to generate JavaScript code may lead to substandard test inputs, resulting in inadequate coverage during browser fuzzing or potential browser crashes. Moreover, WebIDL merely defines the interface of the browser's JavaScript binding layer and does not prescribe the browser's internal implementation. Consequently, browsers only need to implement interfaces within the specification, leading to a narrower implementation scope compared to the specification itself. For example, while the parameter type of a certain method may be defined as unsigned long in WebIDL, the browser implementation may restrict the parameter values to a smaller range, such as [0, 2]. If the generator creates parameters based on the range of unsigned long, the actual code coverage could be significantly reduced. More importantly, many WebIDLs may contain certain semantic deficiencies or inaccuracies after parsing, underscoring the critical importance of supplementing and correcting the semantics of the parsed WebIDL definitions. This ensures that the generated test inputs accurately reflect the browser behavior and effectively exercise its functionalities during fuzzing.

In particular, depending on the granularity of the required correction, the semantic alignment strategy can be categorized into three main types, namely scope-aware management, static correction, and dynamic correction.

4.2.1 Scope-Aware Management. Let us begin by diving into the specific technical details of the scope-aware mechanism.

Scope Tree. To address **Challenge #1** shown in Figure 2, we introduce a scope tree that encapsulates scope information and delineates the dependency relationships between scopes during input generation. Figure 4 depicts the corresponding scope tree for the example. Each node within the scope tree symbolizes a distinct scope [7], while the tree meticulously characterizes the dependencies between scopes. Within the scope node, variable details are documented in the form of a dictionary. As such, with the aid of the scope tree, the generator can access variable information spanning multiple scopes, thus facilitating asynchronous code generation.

Scope Pending. In addition, the generator might inadvertently produce redundant code, impeding the efficiency of the input execution. Considering the example in Figure 5, this code snippet contains an audio context (line 3), two audio nodes (lines 6 and 12),

```
1   ```js
2   // global scope
3   let context = new AudioContext();
4   let source = context.createConstant();    // Redundant node
5   { // local scope A
6       let mediaStreamDestination = context.createMediaStreamDestination();
7       let source = context.createConstantSource();    // Redundant node
8       ......
9   }
10  { // local scope B
11      // merger: an audio node with both input and output interfaces
12      let merger = context.createChannelMerger();
13      let source = context.createConstantSource();
14      source.connect(merger);
15  };
16  ......
```

Figure 5: An example of scope pending-based correction.

and three scopes. When creating a new node (i.e., `source`) to connect to other nodes that have output interfaces, the general logic is as follows: (i) Randomly select a scope associated with the audio context. (ii) Construct a statement to instantiate the new node and insert it into the selected scope. (iii) Traverse the current scope to identify audio nodes that meet certain criteria (i.e., have both input and output capabilities). Upon discovery, connect the two nodes using a `connect` function, such as line 14 in Figure 5. Otherwise, exit the current scope and repeat the previous steps. However, the overall process can yield redundant code, as exemplified by lines 4 and 7 in Figure 5. In the worst case, the generator may exhaustively explore all available scopes. In practical scenarios where the generated test inputs entail numerous scopes, this can result in the production of large amounts of superfluous code, thereby significantly reducing the efficiency of the input execution.

To this end, we incorporate a scope pending mechanism that mitigates the generation of redundant code while ensuring that variables are created within the appropriate scope. Specifically, whenever a variable is created, the associated scope enters a suspended state. At this point, the newly created variables and statements are not immediately incorporated into the scope but instead deposited into a *pending pool*. Upon encountering a node within the current scope that satisfies certain criteria, the code residing in the *pending pool* is merged into the scope. Conversely, the generation of the current statement is revoked. This mechanism effectively prevents the introduction of additional code during the test input generation process, further enhancing the overall fuzzing efficiency.

Cross Execution Context. Additionally, to address **Challenge #3**, TACOMA incorporates a cross-execution-context navigation scheme that allows code generation across different execution contexts. TACOMA autonomously stores the WebIDL semantics of various execution contexts, facilitating seamless switching between them at any time during the code generation process. Furthermore, TACOMA allows state information to be shared between the code generated in different execution contexts, enabling the creation of more complex inputs that accurately reflect real-world application scenarios.

Consider the example generated by TACOMA in Figure 2, which enables code generation in two different execution contexts: `Window` and `AudioWorklet`. Initially, TACOMA utilizes the semantic information of the `AudioWorklet` to create `Class_0` within the `AudioWorklet` execution context and completes the registration process through

```
1   ```idl
2   constructor(unsigned long numberOfChannels,
3                unsigned long numberOfFrames, float sampleRate);
4   ```python
5   try {var var_0 = new offlineAudioContext(4363346, 23855302, 16.0)} catch (e) {};
6   try {var var_0 = new offlineAudioContext(22, 50967462, 338455.95)} catch (e) {};
```

Figure 6: An example of static correction.

`RegisterProcessor` (line 16 in Figure 2). Subsequently, TACOMA transitions to the `Window` execution context and automatically generates an `AudioWorkletNode` based on the information previously generated in the `AudioWorklet` context (line 21). To the best of our knowledge, TACOMA is the first browser fuzzer that is able to generate code in different execution contexts, expanding the code coverage of browser fuzzing. This scheme enhances TACOMA's ability to explore diverse execution states within the browser, ultimately leading to more thorough testing and identification of potential vulnerabilities.

4.2.2 *Static Correction.*

Static correction mainly focuses on numeric or type correction, relying on predefined rules or patterns. Once the WebIDL parsing is completed, TACOMA undertakes static correction by rectifying the parameter value range and parameter type on the corresponding Python object in accordance with the WebIDL definition. This process ensures that the semantic information captured in the Python objects accurately reflects the specified constraints and requirements outlined in the WebIDL specifications, increasing the accuracy and consistency of generated test inputs.

Let us look at a static correction example of TACOMA. Figure 6 shows the `constructor` function of `OfflineAudioContext` in the WebIDL definition (lines 2-3). After parsing, the `OfflineAudioContext` definition is translated into a Python object, as depicted in line 5. Notably, the generated statement is fully compliant with the WebIDL semantics. However, the parameters `numberOfChannels` and `sampleRate` in line 5, if used as is, may result in browser exceptions in practice. This discrepancy arises because browsers impose additional restrictions on the parameter ranges when implementing interfaces, which are often not explicitly reflected in WebIDL files. For instance, the value range for the `numberOfChannels` parameter spans from 1 to 32 for all browsers, while the `sampleRate` parameter exhibits varying ranges across different browsers, namely: Chromium: [3000, 768000], Firefox: [8000, 192000], and Safari: [3000, 384000]. If the generated value falls outside these specified ranges, the resulting code will trigger a runtime exception during execution. To address this issue, TACOMA performs static correction by modifying the range of parsed parameter values to ensure consistency between the WebIDL semantics and the browser implementation. The corrected code, as displayed in line 6 of Figure 6, adheres to the parameter range requirements of the browser, thereby mitigating the risk of exceptions during execution.

4.2.3 *Dynamic Correction.*

Dynamic correction adaptively rectifies the semantic information based on contextual analysis, allowing for more flexible and effective corrections. TACOMA engages with two types of dynamic correction strategies.

Context-Aware Strategy. TACOMA executes dynamic correction in a context-aware manner by systematically tracking specific values along with context information. The context-aware function is called to store context information for both locally and globally generated variables, with them organized in a tree structure.

```
1 ```idl
2 partial interface Document {
3     HTMLVideoElement createVideoElement();
4 };
5 ```js
6 Document.prototype.createVideoElement = function(){
7     return this.createElement('video');
8 }
```

Figure 7: An example of dynamic correction.

This strategy enables Tacoma to document associations between variables throughout code generation, ensuring the accurate recognition of variable mapping relationships. For example, to address **Challenge #2** shown in Figure 2, when generating an audio node, Tacoma marks the audio context variables linked to that node. Specifically, `var_8`, `var_9`, `var_10`, `var_15`, and `var_16` are associated with an audio context (line 23), while `var_11`, `var_14`, `var_17`, `var_18`, and `var_19` pertain to another audio context (line 24). Consequently, when the `connect` function is called, the generator automatically selects an audio node created within the same audio context for the connection. This process ensures semantic coherence and correctness by facilitating appropriate connections within the designated audio context. By evaluating the context during code generation, Tacoma ensures that the corrected semantic information is optimally adapted to the specific requirements of browsers. The context-aware strategy enhances Tacoma's ability to generate semantically correct test inputs that faithfully reflect browser behavior.

IDL Semantic Supplement Strategy. In addition, Tacoma also corrects test inputs by supplementing IDL semantics and JavaScript code. Often, the definitions provided in WebIDL files lack the necessary granularity, resulting in the generation of code with inaccurate or missing semantics. For example, consider the IDL definition for creating DOM elements, namely `Element Document.createElement(DOMString localName)`, with a return value type of `Element`. While this definition is technically correct, it presents a challenge for the generator in determining the specific type of return value. Specifically, when `Document.createElement('video')` is called, it should ideally return an `HTMLVideoElement`. However, if `Element` is used as the return value type, essential semantic information pertaining to `HTMLVideoElement` may be overlooked. Towards this, Tacoma extends the WebIDL definitions by enriching them with additional IDL semantics and associated JavaScript code. Illustratively, in the example depicted in Figure 7, we extend the definition of `Document` by supplementing it with IDL semantics (lines 2-4). Additionally, we introduce the necessary JavaScript code (lines 6-8) to ensure the accurate generation of `HTMLVideoElement` objects without compromising the semantic integrity. This supplementation process allows Tacoma to make more informed corrections and adjustments, thereby enhancing the fidelity of the generated test inputs.

4.3 Input Generation

With the corrected WebIDL semantics, an input generator is called to create test inputs. The input generator comprises two submodules: a JavaScript generator and an HTML producer, working in tandem to generate test inputs for the fuzzer.

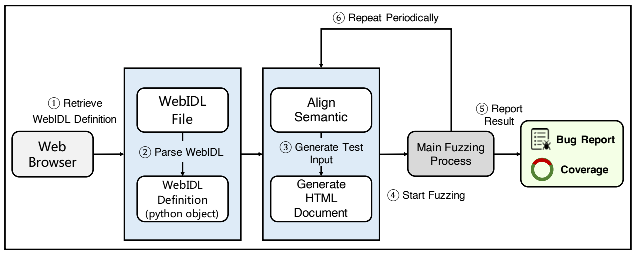


Figure 8: The overall workflow of Tacoma.

JavaScript Generator. The JavaScript generator focuses on generating JavaScript code. It takes advantage of the semantic information extracted from the corresponding WebIDL context to construct JavaScript statements such as variable definitions, function invocations, and more. Note that the corrected semantic information is stored in the appropriate WebIDL context. During the code generation process, the JavaScript generator recursively parses the dependencies of the target statement, ensuring comprehensive code completion for all dependent elements. Then, the generator exposes all functions to the HTML producer in the form of interfaces.

HTML Producer. Tasked with creating an executable target input, i.e., an HTML document, for a browser, the HTML producer plays a key role. It leverages predefined HTML templates and combines them with the JavaScript code provided by the JavaScript generator to generate the HTML document. Notably, depending on the fuzzing goals, the producer is able to adopt a targeted input generation strategy. For example, it can generate test inputs for all browser interfaces or selectively generate input for a specific module, allowing for a tailored and focused fuzzing approach.

Insight: *The semantic alignment module works throughout the entire input generation process. Before input generation, static correction is applied to rectify the WebIDL semantics, mainly focusing on the range of variables. During input generation, dynamic correction is invoked using the context information of the generated code to adjust the WebIDL semantics. Meanwhile, the scope-aware mechanism continuously performs code correction on the generated code during input generation.*

4.4 The Workflow of Tacoma

The overall workflow of Tacoma consists of six main steps. Given the browser source code as input, Tacoma initiates the process by retrieving WebIDL files ①. It then parses the WebIDL files to extract the corresponding definitions and transforms them into Python objects ②. Next, Tacoma generates HTML documents that serve as test inputs for the fuzzer ③. In particular, Tacoma employs the semantic alignment technique to ensure the correctness of the extracted semantic information. After that, the input generator is invoked to produce the HTML documents. Then, Tacoma launches the fuzzer to interact with the web browser ④ and reports the fuzzing results, including exposed bugs and coverage information ⑤. The workflow concludes by looping back to the third step and repeating the subsequent process ⑥. Notably, when applying the

Table 1: Previously unknown bugs discovered by Tacoma. We have evaluated the bug-finding ability of Tacoma on three mainstream browsers. In total, Tacoma has unearthed 32 new bugs. Among them, 21 bugs have been confirmed with 17 fixed. In particular, 10 have been assigned CVEs. Note that the bugs found in WebKitGTK are first verified in Safari. If they can be triggered in Safari, we report them to Safari. Otherwise, we report them to the WebKitGTK community.

ID	Browser	Version	Bug Type	Bug Location	Impact	Status
1	Chromium	95.0.4628.3	Out-of-bounds Read	blink::AudioDelayDSPKernel::ProcessKRate	Medium	CVE-2021-37992
2	Chromium	102.0.4956.0	Out-of-bounds Read	blink::AudioDelayDSPKernel::ProcessKRate	Medium	Issue #40059351 (fixed)
3	Chromium	103.0.5058.0	Null Dereference	blink::WebCryptoResult::Cancelled	None	Issue #40225230 (fixed)
4	Chromium	103.0.5058.0	Null Dereference	PromiseRejectInternal	None	Issue #40260504 (confirmed)
5	Chromium	118.0.5951.0	Use-after-poison	blink::NodeMoveScope::SetCurrentNodeBeingRemoved	Low	Issue #40070829 (duplicated)
6	Chromium	118.0.5993.129	Use-after-free	blink::AudioNodeOutput::Pull	High	CVE-2023-5996
7	Chromium	119.0.6045.159	Use-after-free	blink::AudioNodeOutput::Pull	High	CVE-2023-6346
8	Chromium	120.0.6099.129	Use-after-free	blink::DelayHandler::Process	High	CVE-2024-0224
9	Chromium	120.0.6099.225	Use-after-free	blink::AudioWorkletHandler::Process	High	CVE-2024-0807
10	Chromium	120.0.6099.225	Stack Buffer Overflow	blink::SetSinkIdResolver::Start	None	Issue #41493757 (fixed)
11	Chromium	120.0.6099.225	Null Dereference	blink::SetSinkIdResolver::OnSetSinkIdComplete	None	Issue #41492759 (confirmed)
12	Chromium	120.0.6099.225	Null Dereference	cc::PaintRecord::num_slow_paths_up_to_min_for_MSAA	None	Issue #41492786 (fixed)
13	Chromium	125.0.6382.0	Access Violation	blink::ScriptState	None	Issue #332382759 (fixed)
14	Chromium	125.0.6410.0	Assert Failure	Element::setAnchorElement	None	Issue #41493748 (confirmed)
15	Chromium	125.0.6410.0	Assert Failure	blink::Animation::TimeToEffectChange	None	Issue #333795269 (reported)
16	Chromium	125.0.6410.0	Null Dereference	blink::OutOfFlowLayoutPart::SaveStaticPositionOnPaintLayer	None	Issue #333957174 (reported)
17	Chromium	125.0.6410.0	Null Dereference	IsScrollContainer	None	Issue #333795271 (reported)
18	Chromium	125.0.6410.0	Null Dereference	HasLayer	None	Issue #333952115 (reported)
19	Safari	14.1	Buffer Overflow	CoreAudioModule	High	CVE-2021-30957
20	Safari	14.1	Buffer Overflow	CoreAudioModule	Medium	CVE-2021-30958
21	Safari	14.1	Buffer Overflow	CoreAudioModule	Medium	CVE-2021-30959
22	Safari	14.1	Buffer Overflow	CoreAudioModule	Medium	CVE-2021-30960
23	Safari	14.1	Buffer Overflow	CoreAudioModule	Medium	CVE-2021-30963
24	Safari	17.4.1	Use-after-free	SincResampler::SincResampler	High	OE197284258884 (reported)
25	Safari	17.4.1	Heap Buffer Overflow	computeSampleUsingLinearInterpolation	High	OE197316913793 (reported)
26	Safari	17.4.1	Heap Buffer Overflow	AudioBufferSourceNode::renderFromBuffer	High	OE197316182173 (reported)
27	Safari (WebKitGTK)	2.43.4	Null Dereference	FloatingObjects::moveAllToFloatInfoMap	None	Webkit bug #272296 (reported)
28	Safari (WebKitGTK)	2.43.4	Null Dereference	RenderLayerCompositor	None	Webkit bug #272289 (reported)
29	Safari (WebKitGTK)	2.43.4	Null Dereference	LayoutIntegration::BoxTree::layoutBoxForRenderer	None	Webkit bug #272294 (reported)
30	Firefox	125.0a1	Stack Buffer Overflow	WebCore::HRTFKernel::HRTFKernel	High	Bugzilla #1881947 (duplicated)
31	Firefox	125.0a1	Null Dereference	mozilla::MediaTrackGraphImpl::OneIterationImpl	None	Bugzilla #1882924 (duplicated)
32	Firefox	126.0a1	Null Dereference	NS_ABORT_OOM	None	Bugzilla #1891164 (reported)

fuzzer to a new browser, Tacoma automatically sets up the fuzzing requirements, eliminating the need for additional human resources.

5 Evaluation

In this section, we carry out extensive experiments to evaluate the effectiveness and practicality of Tacoma. Our evaluation seeks to answer the following research questions.

RQ1. How effective is Tacoma in finding new bugs in mainstream browsers, i.e., Chromium, Safari, and Firefox? Can Tacoma uncover bugs in the JavaScript binding code? (§5.2)

RQ2. How well does Tacoma perform compared to existing state-of-the-art browser fuzzers? (§5.3)

RQ3. How much does the semantic alignment technique contribute to performance improvement? (§5.4)

RQ4. What is the running overhead of Tacoma in generating test inputs? What is the impact of semantic alignment? (§5.5)

5.1 Experimental Setup

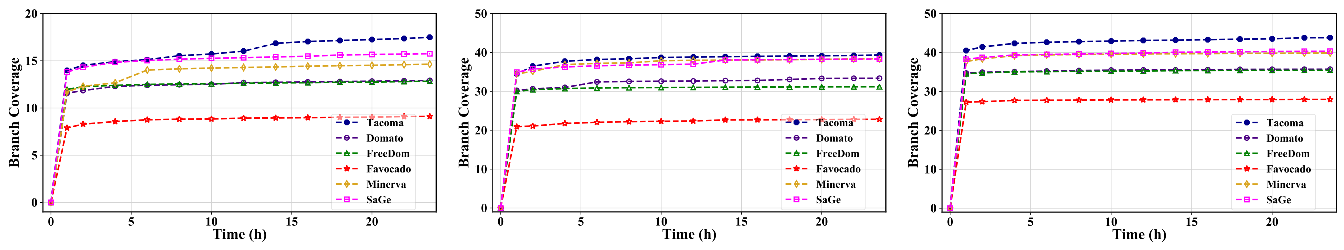
Benchmarks. To comprehensively evaluate the effectiveness of Tacoma, we compare it to five state-of-the-art browser fuzzers, namely Domato [9], FreeDom [42], Favocado [8], Minvera [48], and SaGe [47]. Table 2 demonstrates the key characteristics of each browser fuzzer. Domato stands out as an industrial-grade DOM fuzzer, with both Minvera and SaGe built upon its foundation. FreeDom is a state-of-the-art DOM fuzzer, while Favocado is a novel browser fuzzing framework that focuses on the JavaScript binding code. We compare Tacoma against them in terms of both bug-finding ability and code coverage achieved over three production-level browsers, i.e., Chromium, Safari, and Firefox. Note that all fuzzers in our evaluation are generation-based and do not require any initial seed input for the fuzzing process. This standardizes the comparison and allows for a fair evaluation.

Table 2: The characteristics of each browser fuzzer.

Fuzzer	Year	Publication	Type	Automation
Domato [9]	2017	–	Structure-Aware	Hand-written
FreeDom [42]	2020	CCS'20	Context-Aware	Hand-written
Favocado [8]	2021	NDSS'21	Semantic-Aware	Fully-auto
Minerva [48]	2022	ESEC/FSE'22	API-Oriented	Semi-auto
SaGe [47]	2023	OOPSLA'23	Semantic-Aware	Fully-auto
Tacoma **(ours)**	2024	ISSTA'24	Semantic-Aware	Fully-auto

Implementation. We have implemented Tacoma, which is able to test modern web browsers, including Chromium, Safari, and Firefox. Tacoma is implemented in approximately 15K lines of Python code, consisting of three main modules: a semantic parser, a semantic aligner, and an input generator.

Device. We perform our evaluation on an AMD 5950X CPU (2.20GHz) equipped with 16 cores and 64GB RAM running Ubuntu 22.04 LTS. Since Safari cannot be executed on a Linux system, we utilize WebKitGTK, a comprehensive port of Safari's rendering engine, as an alternative. All experiments are conducted consistently on the same hardware configuration. In practice, we launch 10 independent browser instances simultaneously. Each instance undergoes an automatic restart in case of a crash or a 30-second timeout. Tacoma compiles the target browsers with AddressSanitizer (ASan) [31] for bug detection. We leverage Gcov [2] to collect the code coverage information during the fuzzing campaign.



(a) Chromium (b) Safari (c) Firefox

Figure 9: The tendency of the branch coverage achieved by each browser fuzzer on Chromium, Safari, and Firefox, respectively.

```
1  <html>
2    <body>
3      <script>
4          let ctx = new AudioContext();
5          let sp = ctx.createScriptProcessor();
6          let delay = ctx.createDelay(1);
7          sp.onaudioprocess = function (event){
8              delay.delayTime.automationRate = "k-rate";
9              delay.delayTime.automationRate = "a-rate";
10         };
11         delay.delayTime.linearRampToValueAtTime(1, 2);
12         delay.delayTime.value = -100;
13         delay.connect(ctx.destination);
14         sp.connect(delay);
15     </script>
16   </body>
17 </html>
```

Figure 10: CVE-2024-0224 on Chromium found by Tacoma. The code fragment triggers both UAF and Segv.

5.2 Discovering New Browser Bugs

We now evaluate the bug-finding ability of Tacoma (to answer **RQ1**). We ran Tacoma intermittently on the three mainstream browsers for a month. Table 1 provides the details of all the new bugs found by Tacoma in Chromium, Safari, and Firefox, respectively. In total, Tacoma has found 32 previously unknown bugs. 21 bugs have been confirmed, out of which 17 have been fixed. 10 of the bugs are assigned CVEs, and 9 of which are rated as high impact by the National Vulnerability Database [36]. Notably, more than 100K USD in bug bounty rewards have been awarded thus far. Encouragingly, Tacoma identifies a ten-year-old bug in Chromium (ID 6 in Table 1), which has been exploited [6]. These findings underscore that Tacoma is effective in finding new bugs in browsers. In what follows, we present a case study that shows the ability of Tacoma to identify bugs in the JavaScript binding code.

Case Study: CVE-2024-0224. Tacoma detected a use-after-free (UAF) vulnerability in `WebAudio` in Chromium (ID 8 in Table 1). This bug may allow remote attackers to potentially exploit heap corruption via a crafted HTML page. Figure 10 presents a PoC code snippet of the JavaScript layer generated by Tacoma. This code creates a binding object `ctx` of type `AudioContext` (line 4). Subsequently, `ctx` activates two variables of a script processor node `script_processor` and a delay node `delay` (lines 5-6), respectively, and establishes the connection of the three objects in the order of `sp -> delay -> ctx.destination` (lines 13-14). Particularly, the property `delay.delayTime.automationRate` is updated in the thread `sp.onaudioprocess`, leading to the creation and release of an audio array inside the `delay` node. It is important to note that an audio thread is automatically constructed when the `AudioContext` is initiated, and the audio thread also accesses this property. Unfortunately, the property `delay.delayTime.automationRate` lacks protection from any mutex lock for thread safety, thus its modifications may result in a use-after-free vulnerability.

On the other hand, in Chrome, the default range for the property `delay.delayTime.value` is [0, 1]. However, due to an oversight by Chrome developers, any value can be set to this property. Although Chrome implements a `clamp` function to restrict the illegal values to the range [0, 1], it fails to correctly execute the `clamp` logic during runtime. As a result, when `delay.delayTime.value` is set to -100, the browser encounters an invalid value during processing, thereby leading to referencing of illegal memory and causing a segv crash.

Notably, we ran the five state-of-the-art browser fuzzers, but they failed to trigger the two aforementioned bugs. By contrast, Tacoma successfully discovered them within approximately about a day and a half. The UAF bug has been assigned CVE-2024-0224 by Chrome Inc., with a high security severity rating.

5.3 Comparison to Existing Browser Fuzzers

To answer **RQ2**, we benchmark Tacoma against five state-of-the-art browser fuzzers listed in Table 2. We evaluate each fuzzer based on the achieved branch coverage and the detected unique bugs. Each experiment runs for 24 hours and is repeated five times. We report their average statistical performance.

Branch Coverage. The growth in branch coverage of each fuzzer over a 24-hour period is depicted in Figure 9. It is evident that Tacoma not only consistently outperforms the other fuzzers, but also achieves higher coverage in a shorter duration over all three browsers. On average, Tacoma achieves 17.49%, 39.35%, and 43.81% branch coverage on Chromium, Safari, and Firefox, respectively, making 1.74%, 1.05%, and 3.79% improvement over the state-of-the-art fuzzer SaGe. Note that 1% indicates about 10K branches.

Unique Bugs. We count the number of crashes triggered by each fuzzer using ASan. Table 4 showcases the number of unique bugs discovered by each fuzzer on Chromium and Safari, respectively. We deduplicate each crash based on its root cause as reported by Asan. Notably, within a 24-hour period on Firefox, none of the fuzzers triggered any ASan-reported bugs, so we omit Firefox from Table 4. From the table, we can see that Tacoma outperforms state-of-the-art fuzzers by a large margin. Compared to its counterparts, Tacoma not only leads in the number of crashes but also uncovers

Table 3: Running overhead of input generation with TACOMA and its variants.

Method	Generating 100 inputs			Generating 2,000 inputs			Generating inputs for one hour		
	#Line	Running Time	Generation Speed	#Line	Running Time	Generation Speed	#Line	#Sample	Generation Speed
TACOMA	39,466	8.46s	4,663.50 line/s	791,586	168.74s	4,691.23 line/s	16,543,948	41,632	4,595.53 line/s
TACOMA$^{wo\text{-}d}$	39,923	8.21s	4,865.06 line/s	802,554	164.48s	4,879.40 line/s	17,080,005	42,532	4,744.14 line/s
TACOMA$^{wo\text{-}ds}$	40,372	34.94s	1,155.41 line/s	801,439	691.59s	1,158.83 line/s	3,969,508	9,904	1,102.51 line/s

Table 4: The number of bugs detected by TACOMA and its counterparts within a 24-hour run.

Method	Chromium		Safari (WebKitGTK)	
	#Crash	#Unique	#Crash	#Unique
DOMATO [9]	1	1	45	7
FREEDOM [42]	12	4	26	5
FAVOCADO [8]	1	1	0	0
MINERVA [48]	0	0	16	6
SAGE [47]	1	1	74	7
TACOMA (ours)	**23**	**5**	**114**	**9**

the highest count of unique bugs within a 24-hour timeframe, for both Chromium and Safari.

We conjecture that the advantages of TACOMA stem from three aspects. First, the scope-aware mechanism integrated into the semantic alignment module indeed helps TACOMA generate diverse test inputs more efficiently, facilitating the exploration across a broader range of browser functionalities. Second, the context-aware strategy significantly enhances the accuracy of the generated test inputs, thereby reducing the probability of browser execution exceptions during fuzzing. Third, the newly developed cross-execution-context navigation scheme empowers the fuzzer to effectively navigate through binding code, increasing the probability of finding bugs.

5.4 Effectiveness of Semantic Alignment

By default, TACOMA engages with the semantic alignment to generate test inputs. We are curious about how much this module contributes to the performance gain of TACOMA. To evaluate its effectiveness, we conduct an ablation study (to answer **RQ3**). Specifically, we disable the strategy in TACOMA, resulting in a variant termed TACOMA without semantic alignment, *i.e.,* TACOMA$^{wo\text{-}sa}$. Note that the scope tree of scope-aware mechanism is an integral aspect of the input generator and thus cannot be disabled independently. We then run TACOMA and TACOMA$^{wo\text{-}sa}$ on Chromium for 24 hours.

First, we record the code coverage achieved and the number of crashes detected during execution. Table 5 presents the fuzzing results, from which we can see that TACOMA completely outperforms TACOMA$^{wo\text{-}sa}$ across all metrics. TACOMA achieves a 2.11% increase in branch coverage and finds 9 more unique crashes compared to TACOMA$^{wo\text{-}sa}$. Note that we only use the *audio* template of TACOMA in the ablation experiment, so the achieved coverage is lower than that in the comparison experiments shown in Figure 9. In addition, we also measure the occurrence of runtime errors raised by both TACOMA and TACOMA$^{wo\text{-}sa}$, as summarized in Table 6. Whereas TACOMA$^{wo\text{-}sa}$ consumes samples more efficiently, it results in more runtime errors. Overall, 18.87% of the lines caused runtime errors when running TACOMA$^{wo\text{-}sa}$, while the probability of TACOMA causing a runtime error is less than 1%.

Table 5: The fuzzing results of TACOMA and TACOMA$^{wo\text{-}sa}$ over Chromium for a 24-hour run.

Method	Coverage				#Crash	#Unique
	Region	Function	Line	Branch		
TACOMA	13.68%	18.31%	13.95%	10.74%	2,502	19
TACOMA$^{wo\text{-}sa}$	11.51%	16.31%	12.17%	8.63%	30	10

Table 6: Runtime errors raised by TACOMA and TACOMA$^{wo\text{-}sa}$ over Chromium for a 24-hour run.

Method	#Line	#Sample	#Error	#Error Rate
TACOMA	6,765,173	17,045	63,466	**0.94%**
TACOMA$^{wo\text{-}sa}$	8,422,483	21,005	1,589,726	18.87%

All of the findings underscore that adopting semantic alignment significantly improves the performance of TACOMA. The performance gain can be attributed to the fact that the semantically corrected parameters and attributes are constrained within the valid range, resulting in fewer exceptions generated by the test inputs. As a result, this allows the fuzzer to execute more valid code, facilitating deeper exploration of the browser space.

5.5 Overhead of Input Generation

Finally, to answer **RQ4**, we evaluate the runtime overhead of TACOMA with respect to input generation.

Efficiency of Input Generation. Our evaluation metrics include the number of generated sample lines, the time required to generate test inputs, and the speed of input generation. To provide a comprehensive evaluation, we recorded TACOMA's performance in generating 100 inputs, 2,000 inputs, and continuous inputs for one hour, respectively. Table 3 summarizes the experimental results, revealing that TACOMA can generate approximately 4,600 lines of code per second. Notably, TACOMA exhibits an average input generation time of 0.0846 seconds, whereas browsers require an average of 0.9149 seconds to execute an input. As such, we can conclude that the test inputs are generated significantly faster than the rate at which the fuzzer consumes them, suggesting that input generation does not affect the overall performance of TACOMA.

Impact of Semantic Alignment. Furthermore, it is also interesting to evaluate the impact of the semantic alignment module on the input generation overhead. To this end, we conducted experiments as part of an ablation study. First, TACOMA was modified by selectively disabling the dynamic correction part, denoted as TACOMA$^{wo\text{-}d}$. Then, we disable both the dynamic and static correction parts, referred to as TACOMA$^{wo\text{-}ds}$. The experimental results are illustrated in Table 3, leading to two main observations. (1) TACOMA$^{wo\text{-}d}$ generates test inputs slightly faster than TACOMA, suggesting that the dynamic correction strategy has only a minimal impact on the speed of input generation, an effect considered negligible. (2) TACOMA$^{wo\text{-}ds}$ experiences a significant slowdown, approximately 4 times slower than TACOMA in generating test inputs. This highlights the substantial improvement in input generation speed afforded by the static correction strategy.

6 Discussion

Upgrade of Semantic Alignment. Inspired by the mutation-based input generation approach, in future work, we consider taking advantage of browser runtime information during fuzzing to iteratively upgrade the semantic alignment mechanism of Tacoma. For example, based on the feedback information from code coverage during browser runtime, the input generator is guided to generate code with a higher probability of triggering those browser areas that remained uncovered in previous fuzzing rounds.

Overhead of Semantic Alignment. We have disclosed the overhead of semantic alignment in §5.5. It should be noted, however, that not all semantic alignment operations affect the efficiency of input generation. For example, the static correction method merely constrains the value range and the enumeration value of certain parameters or object attributes, thus yielding negligible runtime overhead. Moreover, the context-aware strategy, which requires computational time (e.g., to retrieve previously generated variable information), may introduce additional overhead in generating test inputs. Nevertheless, it does not impact the fuzzing process, as Tacoma consistently generates test inputs at a pace that exceeds the consumption rate of the fuzzer.

Potential Manual Audit. To improve the correctness of input generation, a manual audit of the browser source code is necessary for addressing unclear IDL definitions. The semantic information identified during the audit is then integrated into the semantic alignment module. It is noteworthy to highlight that Tacoma operates as an automated end-to-end fuzzing framework, and the auditing process is a one-time requirement that is conducted before Tacoma is initiated. Subsequent launches do not require manual auditing unless there are updates to the source code of the relevant browser interface. The goal of such an audit is to ensure that the semantic alignment module consistently receives accurate semantics, thereby optimizing the effectiveness of input generation.

7 Related Work

Generation-based Fuzzing. Generation-based fuzzing is an effective method for testing software programs that require highly structured inputs [21, 26]. This approach is particularly well established for structured input formats, including but not limited to HTML documents. Two categories of generation-based fuzzing strategies are commonly used. The first is to predefine grammar rules for generating syntactically correct test inputs. For example, JsfunFuzz [1] and Csmith [43] construct random code to test a target program based on manually defined grammar models and templates. Domato [9] and FreeDom [42] follow the predefined DOM rules to generate effective HTML documents. The second is to capitalize on existing test cases to learn a grammar model for input generation. Skyfire [39], for instance, builds a probabilistic, context-sensitive grammar model for HTML and XSL files by learning from valid inputs, and then uses the grammar to generate inputs that are accepted by a target software. DeepFuzz [22] employs a sequence-to-sequence model to automatically and continuously generate well-formed *C* programs. Recently, FuzzGAN [14] learns the representation of the input space of deep neural networks, and yields test cases without the constraint of any concrete seed input.

Browser Fuzzing. Browser fuzzing is considered to be one of the most effective ways to find bugs in web browsers [32, 38]. Among the different types of browser fuzzing techniques, DOM fuzzing and JavaScript engine fuzzing are the most relevant works to ours. Early DOM fuzzers embedded themselves in a page loaded by the target browser and called random DOM APIs on-the-fly [30, 46]. The popularity of such fuzzers has declined because a target browser instance ages after a long run, which results in unstable executions and irreproducible crashes. Recently, the generation-based DOM fuzzers have become mainstream [8, 9, 29, 42]. Domato [9] takes advantage of manually-created grammars and semantics to generate DOM API invocations. FreeDom [42] efficiently generates HTML documents by relying on a context-aware intermediate representation. Favocado [8] mainly focuses on testing the browser's binding code with semantically correct test inputs. Minerva [48] explores deeper paths by generating highly relevant API invocations for browser API bug detection using mod-ref relations.

In addition to DOM fuzzers, the active research area of fuzzing JavaScript engines has gained prominence [4, 27, 40, 44]. Current efforts in JavaScript engine fuzzing prioritize the production of semantically correct JavaScript. For example, Montage [18] employs abstract syntax trees for mutation, while CodeAlchemist [15] and IDE [27] strive to generate semantically valid JavaScript inputs. Fuzzil [11] introduces an intermediate representation (IR) to construct both syntactically and semantically correct test cases. SoFi [16] presents an innovative semantic-aware fuzzing technique, leveraging fine-grained analysis, automatic repair, and reflection. Fuzzilli [12] generates semantically correct code using an IR, targeting JIT vulnerabilities in JavaScript engines of modern web browsers. It is worth noting that, despite advances in JavaScript engine fuzzing, the testing of the JavaScript binding code remains relatively unexplored, primarily due to the complex implementation within the binding layer. Surpassing existing efforts, Tacoma excels in performing semantic alignment on the generated test inputs, thus improving the effectiveness of fuzzing the binding code.

8 Conclusion

In this paper, we present Tacoma, a new fuzzing framework tailored for web browsers with a special focus on JavaScript binding code. Tacoma takes advantage of a newly designed parser and aligner to generate semantically correct test inputs for the browser fuzzer. Furthermore, Tacoma exhibits the capability to produce diverse test inputs under the guidance of the semantic alignment mechanism, allowing the detection of deeply embedded defects within web browsers. It is noteworthy that the techniques proposed in Tacoma have the potential to be applied to the fuzzing of other browsers as well. Extensive experimental results show that Tacoma achieves significant improvements compared to state-of-the-art browser fuzzers in terms of bug-finding and code coverage. Quantitatively, Tacoma has successfully detected 32 previously unknown bugs on three mainstream browsers, out of which 10 are assigned CVEs. We anticipate that our tool will assist developers in strengthening browsers, thus contributing to the advancement of browser security.

References

[1] [n. d.]. A collection of fuzzers in a harness for testing the spidermonkey javascript engine. https://github.com/MozillaSecurity/funfuzz.

[2] [n. d.]. Gcov. https://en.wikipedia.org/wiki/Gcov.

[3] Devdatta Akhawe and Adrienne Porter Felt. 2013. Alice in warningland: a {Large-Scale} field study of browser security warning effectiveness. In *22nd USENIX security symposium (USENIX Security 13)*. 257–272.

[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364.

[5] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. 2017. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 559–578.

[6] Google Chrome. 2023. Security: TFC 2023 UAF in WebAudio / Renderer RCE. https://issues.chromium.org/issues/40075943.

[7] Antoine Colin and Guillem Bernat. 2002. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. IEEE, 50–59.

[8] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *NDSS*.

[9] Ivan Fratric. [n. d.]. DOM fuzzer. https://github.com/googleprojectzero/domato.

[10] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.

[11] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[12] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium*.

[13] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. 2013. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics & Applications (ICIA)*. IEEE, 212–215.

[14] Ge Han, Zheng Li, Peng Tang, Chengyu Hu, and Shanqing Guo. 2022. FuzzGAN: A Generation-Based Fuzzing Framework for Testing Deep Neural Networks. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 1601–1608.

[15] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.

[16] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. 2021. SoFi: Reflection-augmented fuzzing for JavaScript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2229–2242.

[17] Sunwoo Kim, Young Min Kim, Jaewon Hur, Suhwan Song, Gwangmu Lee, and Byoungyoung Lee. 2022. {FuzzOrigin}: Detecting {UXSS} vulnerabilities in Browsers through Origin Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1008–1023.

[18] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. 2613–2630.

[19] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[20] Ying-Dar Lin, Feng-Ze Liao, Shih-Kun Huang, and Yuan-Cheng Lai. 2015. Browser fuzzing by scheduled mutation and generation of document object models. In *2015 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 1–6.

[21] Jiawei Liu, Yuheng Huang, Zhijie Wang, Lei Ma, Chunrong Fang, Mingzheng Gu, Xufan Zhang, and Zhenyu Chen. 2023. Generation-based Differential Fuzzing for Deep Learning Libraries. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–28.

[22] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[23] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. 2023. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1237–1251.

[24] Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the Fuzzing Methods: A Comprehensive Survey. *Comput. Surveys* 56, 3 (2023), 1–38.

[25] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[26] Chengbin Pang, Hongbin Liu, Yifan Wang, Neil Zhenqiang Gong, Bing Mao, and Jun Xu. 2023. Generation-based fuzzing? Don't build a new generator, reuse! *Computers & Security* 129 (2023), 103178.

[27] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642.

[28] Peng Qian, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He. 2023. MuFuzz: Sequence-Aware Mutation and Seed Mask Guidance for Blockchain Smart Contract Fuzzing. *arXiv preprint arXiv:2312.04512* (2023).

[29] Mozilla Security. [n. d.]. dharma. https://github.com/posidron/dharma.

[30] SensePost. [n. d.]. Wadi Fuzzing Harness. https://github.com/sensepost/wadi.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[32] Chaofan Shou, Ismet Burak Kadron, Qi Su, and Tevfik Bultan. 2021. CorbFuzz: Checking browser security policies with fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 215–226.

[33] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*. 97–110.

[34] Peter Snyder, Cynthia Taylor, and Chris Kanich. 2017. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 179–194.

[35] Suhwan Song and Byoungyoung Lee. 2023. Metamong: Detecting Render-Update Bugs in Web Browsers through Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1075–1087.

[36] NVD team. 2024. NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov.

[37] WebIDL Team. 2024. Web IDL Standard. https://webidl.spec.whatwg.org.

[38] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webfuzz: Grey-box fuzzing for web applications. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 152–172.

[39] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[40] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium. USENIX*.

[41] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. 2022. COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation.. In *NDSS*.

[42] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 971–986.

[43] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[44] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*. 435–450.

[45] Jianjia Yu, Song Li, Junmin Zhu, and Yinzhi Cao. 2023. CoCo: Efficient Browser Extension Vulnerability Detection via Coverage-guided, Concurrent Abstract Interpretation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2441–2455.

[46] M Zalewski. [n. d.]. cross_fuzz. https://lcamtuf.coredump.cx/cross_fuzz.

[47] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards Better Semantics Exploration for Browser Fuzzing. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 604–631.

[48] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1135–1147.

Received 2024-04-12; accepted 2024-07-03