



Towards Better Semantics Exploration for Browser Fuzzing

CHIJIN ZHOU, QUAN ZHANG, LIHUA GUO, MINGZHE WANG, and YU JIANG*, BNRist, Tsinghua University, China

QING LIAO, Harbin Institute of Technology, China

ZHIYONG WU and SHANSHAN LI, National University of Defense Technology, China

BIN GU*, Beijing Institute of Control Engineering, China

Web browsers exhibit rich semantics that enable a plethora of web-based functionalities. However, these intricate semantics present significant challenges for the implementation and testing of browsers. For example, fuzzing, a widely adopted testing technique, typically relies on handwritten context-free grammars (CFGs) for automatically generating inputs. However, these CFGs fall short in adequately modeling the complex semantics of browsers, resulting in generated inputs that cover only a portion of the semantics and are prone to semantic errors. In this paper, we present SAGE, an automated method that enhances browser fuzzing through the use of production-context sensitive grammars (PCSGs) incorporating semantic information. Our approach begins by extracting a rudimentary CFG from W3C standards and iteratively enhancing it to create a PCSG. The resulting PCSG enables our fuzzer to generate inputs that explore a broader range of browser semantics with a higher proportion of semantically-correct inputs. To evaluate the efficacy of SAGE, we conducted 24-hour fuzzing campaigns on mainstream browsers, including Chrome, Safari, and Firefox. Our approach demonstrated better performance compared to existing browser fuzzers, with a 6.03%-277.80% improvement in edge coverage, a 3.56%-161.71% boost in semantic correctness rate, twice the number of bugs discovered. Moreover, we identified 62 bugs across the three browsers, with 40 confirmed and 10 assigned CVEs.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Browser security**.

Additional Key Words and Phrases: Browser Security, Semantics-Aware Fuzzing, Context-Sensitive Grammar

ACM Reference Format:

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. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 243 (October 2023), 28 pages. <https://doi.org/10.1145/3622819>

1 INTRODUCTION

Web browsers have become an essential component of the modern web experience, serving not only as desktop applications but also as key components in various systems such as IoT devices and industrial control systems [Igalia 2021]. Browsers are expected to accurately implement the functionalities stipulated by W3C standards [W3C 2022a] while also optimizing their execution performance. Therefore, as the codebase and complexity of browsers grow, it becomes increasingly difficult to ensure that they are bug-free. Fuzz testing, also known as fuzzing, is an automated

*Yu Jiang and Bin Gu are the corresponding authors.

Authors' addresses: Chijin Zhou; Quan Zhang; Lihua Guo; Mingzhe Wang; Yu Jiang, BNRist, Tsinghua University, Beijing, China; Qing Liao, Harbin Institute of Technology, Shenzhen, China; Zhiyong Wu; Shanshan Li, National University of Defense Technology, Changsha, China; Bin Gu, Beijing Institute of Control Engineering, Beijing, China.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

© 2023 Copyright held by the owner/author(s).

2475-1421/2023/10-ART243

<https://doi.org/10.1145/3622819>

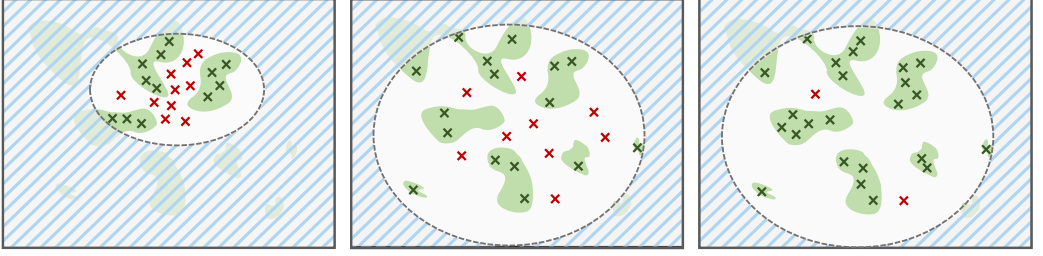
technique widely used for bug-finding. Over the past decade, building an effective fuzzer for browsers gained traction in both academia and industry. Major browser vendors continually run browser fuzzers to discover bugs in their browsers [Google 2019].

Developing an effective fuzzer for web browsers is a challenging task due to the rich semantics of browser inputs. The primary input format for browsers is HTML, which defines the initial Document Object Model (DOM) tree. In addition to HTML, browsers also interpret CSS and JavaScript, which are used to define the styling and interactivity of web pages. These three languages allow developers to create a diverse range of web applications using various exported APIs, domain-specific strings, and data dependencies. For example, developers can use CSS to customize the appearance of an HTML element, defining styles through a combination of domain-specific strings like "margin-left", "small-caps", and "-webkit-inline-box", each with its own unique semantics. These styles can be attached to HTML elements and subsequently accessed, modified, detached, or re-attached using JavaScript during rendering. The complexity and richness of browser input formats create a vast input space, making it challenging for fuzzers to thoroughly explore.

Current browser fuzzers [Dinh et al. 2021; Google 2017; Mozilla 2015; Xu et al. 2020; Zhou et al. 2022] do not well address this challenge. They rely on (semi-)handwritten context-free grammars (CFG) to generate structured inputs. In addition to tedious manual efforts, such grammars have two main drawbacks. First, they are often *incomplete*, covering only a part of the semantics and leaving many functionalities uncovered by the fuzzers. As a result, it limits the possibility of exploring more program states. Second, the grammars are *error-prone*. Browsers perform both syntactic and semantic correctness checks before processing a statement. If a statement fails either check, browsers will immediately terminate processing this statement. Although CFGs enable fuzzers to bypass syntactic checks, they do not address the challenge of bypassing semantic checks.

Semantic Correctness for Fuzzing. While many of existing browser fuzzers [Dinh et al. 2021; Xu et al. 2020; Zhou et al. 2022] claim that their generated inputs are semantically-correct, they can only ensure that the parameters of generated function invocations are of the right types, i.e., type correctness, which is only a subclass of semantic correctness. Other types of semantic errors can also cause early termination by browsers. We observed that semantic errors are still widespread in inputs generated by existing browser fuzzers, even though they ensure type correctness of inputs. To better understand the semantic errors introduced by the fuzzers, we manually analyzed the semantic errors in their generated inputs, and classified them into three main types: compatibility errors, misuse errors, and variable reference errors. Compatibility errors often arise from browser compatibility issues. Misuse errors result from the use of production rules that cannot be used together semantically, even though they are valid when used alone. Reference errors arise when errors in previously generated statements impact subsequent statements that use variables defined in the erroneous statements. Despite their varying appearances, all these semantic errors are caused by incorrect combinations of production rules during input generation of fuzzers.

Semantics-Aware Generation. In this paper, we present SAGE (Semantics-aware Generator), a fuzzer that efficiently explores browser semantics by generating inputs that cover a wide range of browser semantics with fewer semantic errors. Its core idea is to extract a primitive CFG from W3C standards and then enhance it to a Production-Context Sensitive Grammar (PCSG) that includes semantic information. Its novelty lies in its automated generation for PCSG, and utilizing PCSG to address the root cause of semantic errors produced during fuzzing. Fig. 1 provides a simplified illustration of how this idea enhances semantics exploration of browsers. As Fig. 1a presents, existing browser fuzzers rely on (semi-)handwritten CFG, which limits their ability to cover all the semantics and may occasionally trigger semantic errors. In contrast, the first step of SAGE extracts complete browser semantics from W3C standards and represent them in a CFG, allowing generated inputs to explore a more diverse semantics as depicted in Fig. 1b. However, similar to other fuzzers,



(a) Fuzzing with (semi-)handwritten context-free grammar. (b) Fuzzing with W3C-augmented context-free grammar of SAGE. (c) Fuzzing with production-context sensitive grammar of SAGE.

Fig. 1. Illustration of exploring browser's input space using different grammars. The bright oval zone represents the search space of fuzzers with the corresponding grammar. Within the zone, the green areas represent syntactically and semantically valid input spaces of browsers. The crosses denote inputs generated by fuzzers. The green crosses denote valid inputs, while the red crosses denote invalid inputs.

CFG-based input generation inevitably introduces semantic errors. To address this challenge, SAGE infers semantic correctness of each production rule and enhances the extracted W3C-augmented CFG to a semantics-aware PCSG. Fig. 1c illustrates that SAGE with PCSG enhanced is able to explore a more diverse browser semantics with a higher proportion of semantically-correct inputs.

SAGE consists of three main procedures. Firstly, it extracts a CFG from the W3C standards, which serves as a starting point for generating inputs with a wide range of semantics. Secondly, it generates inputs utilizing the CFG and executes them in browsers. By analyzing the semantic correctness of each statement, SAGE can infer whether selecting a production rule during input generation is likely to result in a semantic error under a specific context. This process results in the derivation of a PCSG. Finally, SAGE employs the PCSG to generate inputs that effectively explore a diverse range of browser semantics.

We implement SAGE and evaluate its performance on mainstream browsers, i.e., Chrome, Safari, and Firefox. After 24 hours of fuzzing campaigns, compared to state-of-the-art fuzzers, SAGE on average achieved a 6.03% - 277.80% improvement in edge coverage, achieved a 3.56% - 160.71% improvement in semantic correctness rate, discovered 2x more bugs. During our testing period with SAGE, it discovered 62 bugs in the three browsers, out of which 40 were confirmed with 10 CVE assigned.

In summary, this paper makes the following contributions:

- We identify the limitations of current browser fuzzers and propose a semantics-aware generation strategy to address the limitations.
- We design and implement SAGE. Its generated inputs can cover a wide range of browser semantics with fewer semantic errors. We released relevant artifacts to facilitate further research on this topic.
- We evaluate SAGE on mainstream browsers. It achieves significant improvements compared to state-of-the-art fuzzers. It has detected 62 bugs with 10 CVEs assigned.

2 BACKGROUND AND MOTIVATION

2.1 Grammar-Based Browser Fuzzing

Web browsers accept HTML documents as inputs. An HTML document consists of three parts: (1) an HTML part to define the initial Document Object Model (DOM) tree, (2) a CSS part to specify in which style the elements of DOM tree are rendered; and (3) a JavaScript part to programmatically

manipulate objects in DOM tree or enable other functionalities. Since these three parts require different programming languages, generating input can be challenging. Moreover, the three parts often have data dependencies. For example, HTML code defines an element, and JavaScript code can then access the element in the DOM tree by indexing it using `document.children[i]`. Once the element is accessed, JavaScript can manipulate it using browser APIs [mozilla 2021].

Due to the complex structured input format, browser fuzzers [Dinh et al. 2021; Google 2017; Mozilla 2015; Xu et al. 2020; Zhou et al. 2022] heavily rely on custom grammars, typically in the format of context-free grammars (CFGs), to generate test cases. A CFG is defined as a tuple $G = (N, T, P, S)$ of (1) a set of nonterminals N ; (2) a set of terminals T disjoint from N ; (3) a set of production rules P , which is a finite relation in $N \times (N \cup T)^k$ for some $k > 0$, in the form of $\alpha \rightarrow \beta_1\beta_2 \dots \beta_k$, mapping a nonterminal $n \in N$ to an expansion alternative; (4) a designed start symbol $S \in N$. In general, fuzzers leverage production rules P to perform expansions starting with S , and generate a derivation tree for input generation in the end. We use the classical definition of derivation trees as given in literature [Aho et al. 2007] and in line with other grammar-based fuzzing research [Gopinath et al. 2021; Havrikov and Zeller 2019]. In short, a **derivation tree** for $G = (N, T, P, S)$ is defined as a rooted ordered tree such that (1) the root is S ; (2) each node is a string that contain multiple symbols $[s_1, s_2, \dots, s_k]$ where $s_i \in N \cup T$ and $k \geq 0$ (3) each edge is a production rule $p \in P$; (4) all leaves do not contain any nonterminals.

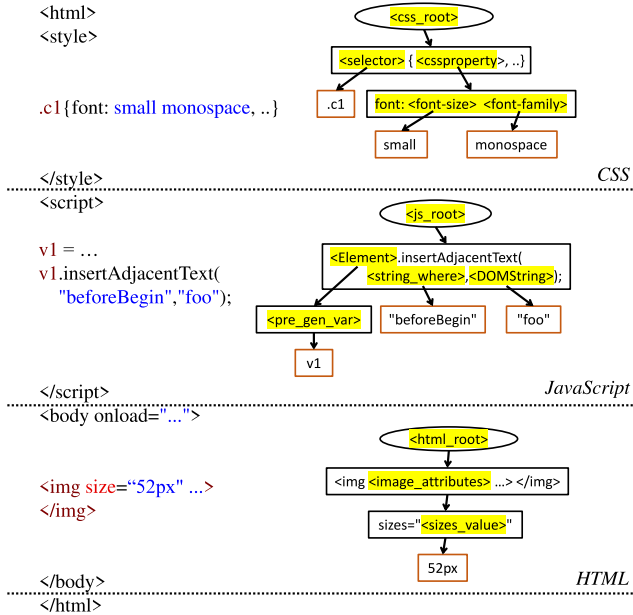


Fig. 2. Illustration of how grammar-based fuzzers generate an HTML document. The trees on the right side are derivation trees generated by the fuzzer. Symbols with a yellow background on nodes denote nonterminals. Edges denote production rules. The statements on the left side are instantiated from the derivation trees.

Fig. 2 takes Domato [Google 2017], the most successful browser fuzzer so far, as an example to illustrate grammar-based browser fuzzing. To generate a derivation tree, the fuzzer recursively expands production rules for each nonterminal until all nonterminals are replaced by terminals. For example, the `<cssproperty>` nonterminal is expanded by the production rule `<cssproperty> → font: <font-size> <font-family>`, and the `<font-size>` and `<font-family>` are recursively expanded

until all symbols are terminals. When a nonterminal symbol needs to be expanded, the fuzzer randomly selects one production rule from all relevant rules. In this example, the fuzzer selects $\langle\text{cssproperty}\rangle \rightarrow \text{font}: \langle\text{font-size}\rangle \langle\text{font-family}\rangle$ from all " $\langle\text{cssproperty}\rangle \rightarrow$ " rules. To generate a test case, the fuzzer often generates hundreds or thousands of derivation trees, each will be instantiated to a statement in the end. Besides, the fuzzer maintains a set of previously-generated variables (e.g., $v1$ in Fig. 2) in order to leverage these previously-generated variables when generating a new derivation tree.

For now, all existing browser fuzzers choose generation-based techniques. Although mutation-based coverage-guided fuzzing has been successful in other areas [Chen et al. 2019; Padhye et al. 2019b; Sun et al. 2021; Wang et al. 2021b; Wu et al. 2022; Zalewski 2013; Zhong et al. 2020], it has limited effectiveness in browser fuzzing as the investigation of [Xu et al. 2020]. This may be due to the fact that browsers have multiple processes, each spawning several background threads to handle tasks that are not related to the inputs [Zhou et al. 2022], making it difficult to collect stable coverage data to guide fuzzers' mutation.

2.2 Challenges in Browser Fuzzing

Previous research [Liu et al. 2023; Nguyen and Grunske 2022] highlights that the effectiveness of fuzzing largely depends on the diversity and semantic correctness of inputs. This holds true for browser fuzzing as well. However, upon investigating the inputs generated by existing browser fuzzers, we found that there is significant room for improvement in terms of both diversity and semantic correctness. This issue stems from the grammars used by these fuzzers. First, the grammars typically only include a part of semantics, resulting in generated inputs that cannot explore diverse program states. Second, the grammars do not take semantics information into account during input generation, causing the generated inputs prone to trigger semantic errors. We will discuss in detail these two challenges faced by existing fuzzers in the following paragraphs.

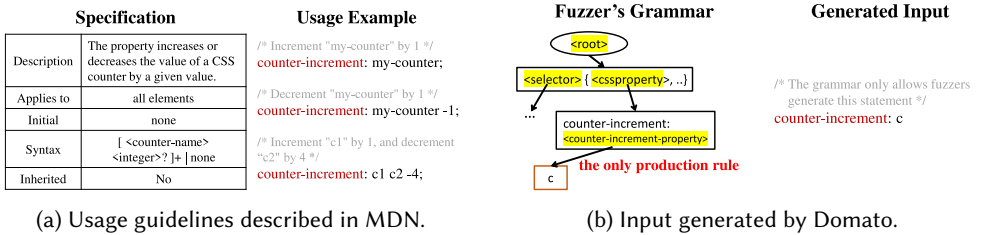


Fig. 3. An example of *counter-increment* to present the incomplete semantics of existing fuzzers' grammars.

Incomplete Semantics Covered. A fuzzer is supposed to generate diverse inputs to cover as many semantics of the target system as possible. However, existing handwritten (or semi-handwritten) grammars cannot meet this end. We take Domato as an example to demonstrate the incomplete semantics problem. On the one hand, only a part of functionalities can be covered by the grammar of Domato. 496 out of 882 exported CSS style properties are involved in the grammar. As a result, Domato is unlikely to explore the backend logic that handles the remaining CSS style properties. On the other hand, even though a functionality is involved in the grammar, it cannot sufficiently explore the semantics of the functionality. Fig. 3 demonstrates how Domato generates a statement related to *counter-increment* CSS style property. Fig. 3a shows the specification and usage examples of this style property, as described in MDN web development documents [Mozilla 2005]. From the figure we can see that this style property has rich semantics such as increasing a counter, decreasing a counter, or increasing multiple counters. However, as Fig. 3b shows, Domato can only

generate *counter-increment: c* statement because its grammar only involves one production rule, i.e., $\langle \text{counter-increment-property} \rangle \rightarrow c$, for this property. In conclusion, existing browser fuzzers cover limited browser semantics due to both incomplete functionalities and insufficient explorations.

Incorrect Semantics Generated. To prevent early bail-out due to semantic errors, many existing browser fuzzers [Dinh et al. 2021; Google 2017; Xu et al. 2020; Zhou et al. 2022] pay much attention to ensure the semantic correctness of their generated inputs. However, they can only ensure the parameters of generated function invocations are of the right types, i.e., type correctness. Semantic errors of other types could also lead to early bail-out by semantic sanity checks of browsers. We observed that semantic errors are still widespread in the inputs generated by browser fuzzers even though they ensure type correctness the inputs. We conducted a manual analysis to identify semantic errors caused by the fuzzers. To make it easier to understand, we categorized them into three main types: compatibility errors, misuse errors, and reference errors. We observed that, despite their varying appearances, all semantic errors are caused by incorrect combinations of production rules during input generation of fuzzers. Fig. 4 gives three examples to demonstrate the semantic errors caused by Domato [Google 2017].

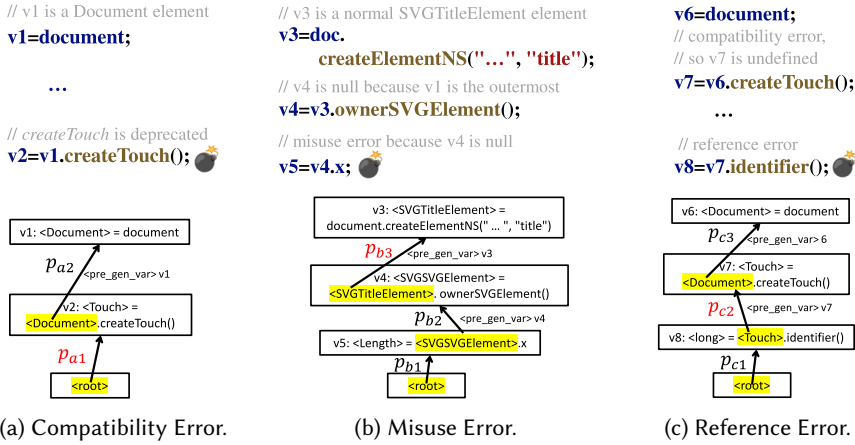


Fig. 4. Three types of semantic errors incurred by existing fuzzer's grammar. Production rules highlighted in red (i.e., p_{a1} , p_{b3} , and p_{c2}) are the root error causes during production rule selection.

Compatibility errors arise from browser compatibility issues. Web standards have undergone significant changes over the years, with the aim of enhancing the user experience, security, and accessibility of the web. As a result, many functionalities have been deprecated from or added to the standards. Browsers typically support only a portion of new functionalities and gradually remove deprecated ones. However, handwritten grammars do not take this into account. This means that inputs generated by the grammars may use functionalities that are not compatible with certain browsers. As Fig. 4a shows, based on the custom grammar, Domato generates an input that uses the `createTouch()` function of a `Document` instance. However, many browsers remove this functionality in their recent versions, so browsers immediately terminate the execution of this statement due to this compatibility error. Its root cause is the incorrect selection of the production rule p_{a1} during input generation. If the fuzzer knows selecting p_{a1} definitely leads to a semantic error, it can then exclude this rule from consideration during input generation, effectively avoiding the error.

Misuse errors often result from the use of production rules that cannot be used together semantically, even though they are valid when used alone. Fig. 4b presents a straight-forward example.

The first statement creates an instance of *SVGTitleElement*, namely *v3*. Next, the input calls the *ownerSVGElement()* function from *v3*. By design, this function is supposed to return its nearest ancestor *SVGSVGElement* instance. However, in this case, *v3* is the outermost element, so the function can only return a *null*, which is assigned to *v4*. When generating the input, the fuzzer regards *v4* as a *SVGSVGElement* instance, so it generates a *v5=v4.x* statement following production rules of its grammar. When browsers execute this statement, they bail out immediately because calling *null.x* is illegal. The root cause of misuse errors is selecting a semantically invalid production rule under a specific context. In this case, selecting *p_{b3}* is semantically invalid because it implicitly makes the *ownerSVGElement()* function return *null*. If we select a different production rule instead of *p_{b3}* to ensure that *v3* has an ancestor element, e.g., *v3=<SVGSVGElement>.children[0]* instead of *v3=doc.createElementNS(..)*, then *v4* will not be a *null* value and the *v5=v4.x* statement will be semantically valid. Therefore, the fuzzer is likely to avoid such errors if it considers the context of derivation tree (i.e., *p_{b1}* and *p_{b2}*) when deciding whether to select the production rule *p_{b3}*.

Reference errors arise when previously-generated statements contain errors and the subsequent statements are affected. Specifically, if a previous statement triggers a compatibility or misuse error, and a subsequent statement uses the outcome variable of the error statement, then the subsequent statement will trigger a reference error. This definition is identical with the common definition in JavaScript, i.e., a reference error only occurs when variables that are being used are not defined. As we can see from Fig. 4c, *v7=v6.createTouch()* triggers a compatibility error, so the variable *v7* is an undefined value. However, when generating the input, the fuzzer regards that *v7* is a *Touch* instance, so it generates a *v8=v7.identifier()* statement following production rules of its grammar. When browsers execute this statement, they bail out immediately because calling any functions of an undefined variable is illegal. To mitigate such errors, fuzzers need to effectively address the other types of errors.

2.3 Towards Better Semantics Exploration

Both of the challenges listed above reveal that the inherent limitation of handwritten CFG prevents fuzzers from efficiently exploring browser semantics. In this paper, we aim to automatically build a production-context sensitive grammar (PCSG) with complete semantics for fuzzing to address the challenges. Our method extracts complete semantics from W3C standards to ensure that most functionalities described in the standards are covered. Additionally, our method aims to mitigate the three semantic errors. Through our observation, we identified *the root cause of these semantic errors as the lack of consideration for the generation context when selecting production rules*. To address this issue, our approach attempts to infer the semantic correctness of the generation context for each production rule. By doing so, our fuzzer can identify which production rule is likely to cause semantic errors in a given context, and consequently avoid selecting it to prevent semantic errors. This approach enables our fuzzer to generate inputs that explore a wider range of browser semantics, with a higher proportion of semantically-correct inputs, ultimately leading to more efficient and effective bug-finding. We will provide further details on our approach in the next section.

3 APPROACH

This section presents the technical details of our proposed approach SAGE, an automatic semantics-aware input generator for browser fuzzing. The overall workflow of SAGE is illustrated in Fig. 5. It includes three modules: (1) a *grammar extraction* module to extract a preliminary CFG; (2) a *semantics inference* module to refine the CFG to a PCSG; and (3) an *input generation* module to generate fuzzing inputs based on the PCSG.

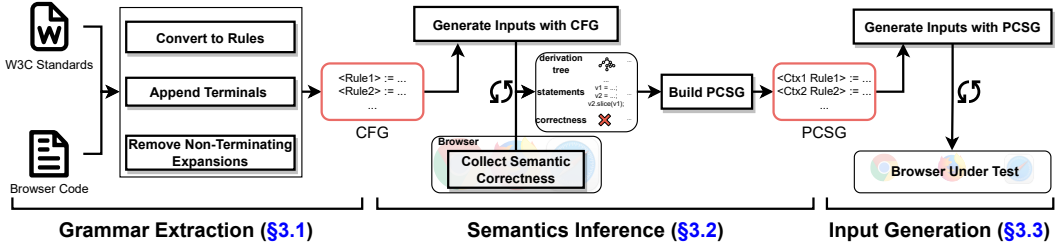


Fig. 5. Overall workflow of SAGE. First, it extracts a CFG from W3C standards and browser source code. Next, it infers semantics by collecting semantic correctness of each input generated by the CFG, and embeds the semantics into a PCSG. Third, it leverages the PCSG to generate semantics-aware inputs for browser fuzzing.

3.1 Grammar Extraction





The goal of the grammar extraction module is to extract a preliminary CFG from W3C standards and browser source code. Overall, this module utilizes a combination of parsing techniques and static analysis to extract the necessary production rules from the W3C standards and the browser source code. Next, It extracts production rules from the W3C standards and then refines them to ensure that the rules are both terminable and expandable. The extracted CFG will be used as a starting point to enable our fuzzer to generate inputs with a wide range of semantics. This module consists of three steps to achieve its goal. First, it converts W3C standards into production rules. Next, it appends terminals to ensure that every nonterminal has at least one production rule to expand it. Finally, it removes non-terminating expansions to guarantee that every nonterminal can be expanded to a string that only consists of terminals.

Convert to Rules. This step involves extracting the relevant information from the W3C standards documents and converting them into a preliminary set of production rules for the browser fuzzer. W3C provides detailed specifications that define the usage of all CSS properties, HTML elements, and JavaScript APIs. For example, as Fig. 3a shows, the specification of the *counter-increment* CSS property includes details on the initial value, applied elements, and value syntax of this property. As of JavaScript APIs, the specifications use a WebIDL [W3C 2022b] format to define API declarations, including API names, member variables and functions, and inheritance relations. Our aim is to embed the semantics of these specifications into production rules of a CFG.

Table 1 outlines the basic conversion rules utilized in our approach. For the CSS component, conversion rule 1 generates production rules that adhere to CSS paradigms, while conversion rules 2-4 ensure syntactic correctness of CSS values. This enables our fuzzer to generate statements such as `"s1 {scale: none}"` and `"s1 {scale: 100 3 90%}"` for the *scale* property in the working example. As for the HTML component, tag names and attributes of elements are extracted from API declarations using conversion rules 5-6. This allows our fuzzer to generate statements such as `<button disabled id="button1"></button>` for the *button* element in the working example. In the JavaScript component, our conversion process takes into account member variables, member functions, and inheritance relations of every interface in the WebIDL specifications. For each member variable, we create two production rules - one for reading and one for writing. If a variable is read-only, then we only generate the read production rule. For each member function, we convert it to a function invocation. Additionally, if an interface is inherited from another interface, we add a production rule such as $\langle \text{var HTML}Element \rangle = \langle \text{HTML}ButtonElement \rangle$ to represent implicit type conversions.

Append Terminals. This step enriches terminals in the extracted grammar. On the one hand, the extracted grammar may not ensure that every nonterminal can be expanded. During input generation, the derivation tree requires all leaves to be terminals. Therefore, if there is a nonterminal

Table 1. List of basic conversion rules.

Component	Conversion Rule	Working Example
CSS	1. Construct production rules of the CSS start symbol based on CSS paradigms.	<pre>Selector { Declaration1, Declaration 2, ... }</pre>  <pre> <css_root> → {selector} { <declaration> } <css_root> → {selector} { <declaration>, <declaration> } </pre>
	2. Convert descriptions of CSS properties to production rules of $\langle declaration \rangle$.	<pre> PropertyName: "scale", Value: "none" [<number> <percentage>][2,3], ... </pre>  <pre> <declaration> → scale: <scale_value> <scale_value> → none <scale_value> → <number> <number> <scale_value> → <number> <number> <number> <scale_value> → <number> <number> <percentage> ... </pre>
	3. Expand production rules based on combination operators between value symbols.	
	4. Expand production rules based on repetition operators after value expressions.	
HTML	5. Construct the production rules of the HTML start symbol based on tag names.	<pre>// interface of <button> tag interface HTMLButtonElement { attribute boolean disabled; ... }</pre>  <pre> <html_root> → < button <button_attr> id = "button1" > </button > <button_attr> → "disabled" "" </pre>
	6. Construct attributes of elements based on the declarations of element interfaces.	
JavaScript	7. Construct r/w statements for member variables based on interface declarations.	<pre> interface HTMLButtonElement: HTMLElement { attribute HTMLFormElement form; boolean checkValidity(); ... } </pre>  <pre> (js_root) → <HTMLButtonElement>.form = <HTMLFormElement>; <var HTMLFormElement> = <HTMLButtonElement>.form; <var bool> = <HTMLButtonElement>.checkValidity(); <var HTMLElement> = <HTMLButtonElement>; </pre>
	8. Construct calling statements for member functions based on interface declarations.	
	9. Add implicit conversions based on the inheritance of interfaces.	

that cannot be expanded, the fuzzers cannot generate a derivation tree. On the other hand, although W3C standards provide a comprehensive set of language features and constraints, they do not encompass all possible features that may exist in a browser implementation. Browsers may use custom keywords to enable non-standard features that can only be identified through the analysis of browser source code. For example, Safari performs image deblurring when the image-rendering property of an image element is set to the custom keyword *"-webkit-optimize-contrast"*. Additionally, different browsers may use distinct keywords for the same non-standard feature. For instance, Safari and Chrome use *"-webkit-font-smoothing"* to make a rendered font anti-aliased, while Firefox uses *"-moz-osx-font-smoothing"*.

To address these issues, we perform a data-flow analysis on the property-handling logic in different browsers' source code and identify keywords to complement our production rules. This analysis ensures that most nonterminals have at least one production rule to expand it. The only exceptions are five primitive nonterminals, namely $\langle int \rangle$, $\langle float \rangle$, $\langle string \rangle$, $\langle hex \rangle$, and $\langle url \rangle$, which cannot be further expanded. For these cases, SAGE randomly generates values based on their types during the fuzzing process. After this step, we make sure that every nonterminal can be expanded.

Remove Non-Terminating Expansions. Although some nonterminals have production rules to expand them, they may still be unable to be expanded to a string consisting solely of terminals if all expansion paths are loops. For example, considering this expansion process:

```
 $\langle var \ XRSession \rangle \rightarrow \langle XRSessionEvent \rangle.session,$ 
```

$\langle \text{var } XRSessionEvent \rangle \rightarrow \text{new } XRSessionEvent(\langle DOMString \rangle, \langle XRSessionEventInit \rangle),$
 $\langle \text{var } XRSessionEventInit \rangle \rightarrow \{ \text{session} : \langle XRSession \rangle \}.$

This is an expansion loop since the expansion of $\langle XRSession \rangle$ necessarily requires itself. If all the expansion paths of $\langle XRSession \rangle$ are loops, then $\langle XRSession \rangle$ will never be expanded to a string. To solve this issue, we adopt a depth-first search approach for every nonterminal. We start from a nonterminal and follow its production rules recursively until we reach a terminal or a nonterminal that has been previously visited. If all leaves of the search tree for a nonterminal are terminals, we consider it as a terminating nonterminal and keep it in the grammar. Otherwise, we remove the nonterminal. After this step, we obtain a CFG whose production rules satisfy both the expansion and termination requirements, enabling input generation for fuzzing.

3.2 Semantics Inference

The semantics inference module is designed to augment the CFG extracted in the previous module with semantics information to reduce the occurrence of semantic errors in generated inputs. As discussed in Section 2.2, compatibility errors and misuse errors arise from the incorrect selection of production rules during input generation. To mitigate these errors, our fuzzer needs to anticipate which production rules will lead to semantic errors under specific contexts during input generation, and then avoid selecting those rules.

This module transforms the CFG into a PCSG that includes semantic information in order to assist production rule selection during input generation. Formally, suppose our extracted CFG is denoted as $CFG = (N, T, P, S)$, where N is a nonterminal set, T is a terminal set disjoint from N , P is a set of context-free production rules of the form $\alpha \rightarrow \beta_1\beta_2 \dots \beta_n$, and P is a start symbol. This module transforms the CFG into a $PCSG = (\tilde{N}, \tilde{T}, \tilde{P}, \tilde{S})$, where \tilde{N} is identical to N , \tilde{T} is identical to T , \tilde{S} is identical to S , and \tilde{P} is a set of context-sensitive production rules. Each rule $\tilde{p} \in \tilde{P}$ is derived from a rule $p \in P$ but in the form $[C_{\tilde{p}}]\alpha \rightarrow \beta_1\beta_2 \dots \beta_n$, where $C_{\tilde{p}}$ is a context-checking function for production \tilde{p} . We define the context and the context-checking function as follows:

Definition 3.1 (Context of Production Rule). During derivation tree generation, suppose $\tilde{p} \in \tilde{P}$ is used to expand nonterminal $\tilde{n} \in \tilde{N}$ and \tilde{S} is the root node of the tree, the context ctx of \tilde{p} is defined as a sequence of production rules $[\tilde{p}_1, \tilde{p}_2, \dots, \tilde{p}_n]$ along the path $\tilde{S} \rightsquigarrow \tilde{n}$ in the tree.

Definition 3.2 (Context-Checking Function). During derivation tree generation, suppose ctx is the context of $\tilde{p} \in \tilde{P}$ and $C_{\tilde{p}}$ is the context-checking function for \tilde{p} , if selecting \tilde{p} as the next rule definitely leads to a semantic error, then $C_{\tilde{p}}(ctx) = \text{false}$; otherwise, $C_{\tilde{p}}(ctx) = \text{true}$.

The semantics inference module tries to infer the context-checking function C for every $\tilde{p} \in \tilde{P}$, enabling our fuzzer to avoid production rules that are likely to lead to semantic errors under specific contexts. To infer C , we utilize a trial-and-error approach, where we generate inputs using the extracted CFG and evaluate their semantic correctness in browsers. By analyzing the derivation trees and corresponding semantic correctness of inputs, we can infer if a production rule is likely to result in a semantic error under certain context. We provide details on how we generate inputs, collect semantic correctness, and build the PCSG in the following paragraphs.

Generate Inputs with CFG. In this step, our goal is to generate inputs with the extracted CFG to facilitate the subsequent PCSG building process. The CFG guarantees that every nonterminal is able to expand to a string consisting only of nonterminals, and therefore each expansion from the start symbol will result in a usable input statement. To generate inputs using the extracted CFG, we utilize the code from Domato [Google 2017]. However, we have modified its generation strategies to ensure that the probability of selecting every rule is equal. This approach guarantees that all production rules have an equal opportunity to contribute to the input generation process, which

eliminates any bias in the collected data. We also preserve the corresponding derivation trees of the generated input statements to use them later in the PCSG construction process.

Collect Semantic Correctness. In this step, our goal is to collect the semantic correctness of the generated input statements. To achieve this, we implement a runtime error monitor in the browsers. This monitor is built on top of the JavaScript interpreter without intrusively instrumenting the browsers, providing flexibility in detecting semantic errors. We detect semantic errors for JavaScript statements by catching exceptions, while for CSS statements we check if the CSS property exists, and for HTML statements we check if the HTML element ID exists. After implementing the error monitor, we execute the generated inputs using a browser and record the results of the execution. If an input statement causes a semantic error, we label it as semantically incorrect; otherwise, we label it as semantically correct. By collecting semantic correctness, we can identify which production rules may lead to semantic errors under specific contexts during the input generation process.

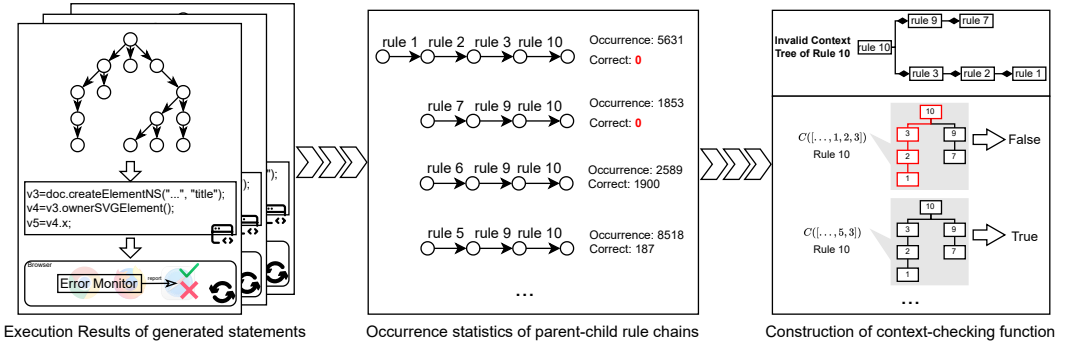


Fig. 6. Illustration of PCSG building.

Build PCSG. In this final step, we aim to transform the extracted CFG into a PCSG by constructing the context-checking function C for each production rule, so that we can turn CFG into PCSG. The purpose of this function is to allow for a more semantically correct exploration during the input generation process. Fig. 6 illustrates how we construct the context-checking function C for each production rule. We first collect the execution results of the generated input statements as illustrated in the previous steps. Using this information, we analyze the derivation trees and execution results to deduce the correctness statistics of the contexts of each rule. The context of a rule is a sequence of production rules, as defined in Definition 3.1, and we use this information to construct a context-checking function for each rule.

We design an invalid context tree to embed observed semantic correctness information and efficiently check semantic correctness of an incoming context. The invalid context tree is a rooted tree where the root node is the rule, and a path from root to a leaf is an invalid context of the rule. For example, as shown in the middle figure of Fig. 6, the rule sequence $[\dots, 1, 2, 3]$ is a context of rule 10, and we observe that this context never leads to a semantically correct statement. We thus assume that selecting this rule will never lead to a semantically correct statement under this context. Therefore, we append the sequence to the invalid context tree of rule 10 to keep track of this invalid context.

The context-checking function is constructed based on the invalid context tree. Specifically, during the input generation process, the context-checking function returns true if the suffix of the current context matches a path in the invalid context tree for the rule. For instance, if the suffix of the current context is $[\dots, 1, 2, 3]$, then the $C([\dots, 1, 2, 3])$ for rule 10 will return false, as this

sequence matches a path in the invalid context tree for this rule. Algorithm 1 further illustrates the construction workflow of context-checking functions. The input of this algorithm is a list of tree-result pairs, which records the execution result (i.e., whether triggering semantic errors or not) of each derivation tree, and the output is a map of rules to context-checking functions. The algorithm begins by traversing each derivation tree to gather execution statistics for each rule chain (Line 3-5 and Line 18-24). After that, the map S keeps track of the total occurrences and semantically-correct occurrences of each rule chain. Next, the algorithm processes each rule-statistics pair, and constructs invalid trees for production rules (Line 6-13). An invalid tree is a rooted tree where its nodes denote production rules, its edges denote the predecessor relations between rules. A path from its root to one of its leaves is an invalid context of the rule. If a rule chain is found to be consistently incorrect, the algorithm updates the invalid context trees for the rules in this rule chain (Line 8-12). Finally, this algorithm constructs context-checking functions of production rules using the invalid context trees (Line 14-17). Given a context for a rule, the rule's context-checking function can verify if this context matches a root-to-leaf path in its invalid tree. If a match is found, it indicates an invalid context; otherwise, it is valid. In this way, the algorithm effectively works out the context-checking functions for production rules.

Algorithm 1: Construction of Context-Checking Functions

```

Input    : List of tree-result pairs  $L$ 
Output   : Map of rules to context-checking functions  $C$ 
1  $C \leftarrow \text{emptyMap}()$ 
2  $S \leftarrow \text{emptyMap}()$  // map of rule chains to occurrence statistics
3 for  $\text{tree\_root}, \text{result}$  in  $L$  do // derivation tree and its execution result
4   |  $\text{traverse}(\text{tree\_root}, \text{result}, S, [], 0)$  // traverse the derivation tree and update  $S$ 
5 end
6  $I \leftarrow \text{emptyMap}()$  // map of production rules to invalid trees
7 for  $\text{rule\_chain}, \text{stats}$  in  $\text{iteratels}(S)$  do
8   | if  $\text{stats.correct\_num}$  is 0 &&  $\text{stats.occurrence} > \text{MIN\_OCCURRENCE\_LIMIT}$  then
9     |   for  $\text{rule}, \text{prefix\_chain}$  in  $\text{rule\_chain}$  do
10      |   |  $\text{invalidTree} \leftarrow I[\text{rule}]$  // path from its root to leaf is the rule' invalid ctx
11      |   |  $\text{updateTree}(\text{invalidTree}, \text{prefix\_chain})$ 
12      |   end
13 end
14 for  $\text{rule}, \text{invalidTree}$  in  $\text{iteratels}(I)$  do
15   |  $C_{\text{rule}} \leftarrow \text{createCheckFunc}(\text{invalidTree})$  // The tree can check if a given ctx is invalid
16   |  $C[\text{rule}] \leftarrow C_{\text{rule}}$ 
17 end
18 Function  $\text{traverse}(\text{node}, \text{result}, S, \text{rule\_chain}, \text{depth})$ :
19   | if  $\text{depth} > \text{MAX\_RULE\_LIMIT}$  then
20     |   return
21   |  $\text{updateStatistics}(S, \text{rule\_chain}, \text{result})$  // record the execution result of this chain
22   | for  $\text{child}$  in  $\text{listChildren}(\text{node})$  do
23     |    $\text{traverse}(\text{child}, \text{result}, S, \text{rule\_chain} + [\text{node}], \text{depth} + 1)$ 
24   | end

```

In this way, we embed highly likely invalid sequence of parent rules for each production rule into an invalid tree, and leverage this tree to quickly check if selecting a rule may lead to a semantic error under a specific context. We take compatibility errors and misuse errors as examples to illustrate how the context-checking function mitigates semantic errors described in Section 2.2.

Example 1: Mitigate Compatibility Errors. Let's focus on a production rule " $\langle root \rangle \rightarrow \langle var Touch \rangle = \langle Document \rangle.createTouch()$ ". The `createTouch()` is a deprecated function, so every time we select this production rule, the corresponding generated statement will definitely lead to a semantic error. Therefore, after a number of executions, we can know that this rule occurs many times but none of the occurrence results in a semantically correct execution. As a result, the context-checking function $C([\])$ for this rule will always return false.

Example 2: Mitigate Misuse Errors. We take the Fig. 4b in Section 2.2 as an example. Let the production rule " $\langle root \rangle \rightarrow \langle Length \rangle = \langle SVGSVGElement \rangle.x$ " be p_1 , the production rule " $\langle varSVGSVGElement \rangle \rightarrow \langle SVGTitleElement \rangle.ownerSVGElement()$ " be p_2 , and the production rule " $\langle varSVGTitleElement \rangle \rightarrow document.createElementNS(..., "title")$ " be p_3 . Although these three production rules are semantically correct, they cannot be used together semantically. Therefore, after a number of executions, we can know that the rule chain $[\dots, p_1, p_2, p_3]$ does not result in a semantically correct execution even though it has been executed many times. As a result, the context-checking function $C_{p_3}([\dots, p_1, p_2])$ will always return false.

3.3 Input Generation

After constructing $PCSG = (\bar{N}, \bar{T}, \bar{P}, \bar{S})$ from the semantics inference module, SAGE is able to generate inputs that effectively explore browsers' semantics. Algorithm 2 shows the algorithmic sketch of how SAGE generate inputs using PCSG. For each language component, we repeatedly generate statements by expanding its start symbol \bar{S} until reach user-specific maximum length (Line 3-7). During the expansion of a symbol S , SAGE first retrieve the candidate expansion rule set \bar{P}_S of the symbol S (Line 12). Next, SAGE randomly chooses a \bar{p} that satisfies the condition

Algorithm 2: Input Generation with PCSG

Input : Production-context sensitive grammar $PCSG$
Output : HTML document H

```

1  $H \leftarrow \text{emptyString}()$ 
2 for  $Lang$  in  $[css, html, js]$  do
3   while  $\text{len}(H, Lang) < MAX\_LEN$  do
4      $\bar{S} \leftarrow \text{startSymbol}(PCSG, Lang)$  // start symbol of current language
5      $C \leftarrow PCSG.checkFunc$  // context-checking function
6      $CTX \leftarrow \text{initialization}()$  // context of this derivation tree
7      $\text{expand}(H, \bar{S}, CTX, C)$ 
8   end
9 end
10 return  $H$ 
11 Function  $\text{expand}(H, S, CTX, C)$ : // to expand nonterminal symbol  $S$ 
12    $\bar{P}_S \leftarrow \text{expansionRules}(S)$  // candidate expansion rule set for  $S$ 
13    $\bar{p} \leftarrow \text{randomChooseOne}(\bar{P}_S)$ 
14   while  $C_{\bar{p}}(CTX)$  is false do // check semantic correctness of  $p$  under  $CTX$ 
15      $\bar{p} \leftarrow \text{randomChooseOne}(\bar{P}_S)$ 
16   end
17    $\text{update}(\bar{p}, CTX)$  // update the current context
18   for  $symbol$  in  $\bar{p}$  do
19     if  $symbol$  is nonterminal then
20        $\text{expand}(H, symbol, CTX, C)$ 
21     else
22        $H += symbol$ 
23   end

```

$C_{\bar{p}}(CTX) = \text{true}$ (Line 13-16). In this way, we can avoid production rules that are likely to lead to semantic errors under this context CTX . Finally, SAGE traverses all nonterminals in \bar{p} and recursively expands these nonterminals (Line 18-23).

The advantages of SAGE over existing browser fuzzers are twofold. The first is that semantics exploration is extensive, enabling the fuzzer to cover a wider range of semantics of browsers. As shows in line 12 in Algorithm 2, every time we expand a symbol S , we need to randomly choose an expansion production rule from the candidate set \bar{P}_S . Since the production rules used by SAGE are extracted from W3C standards, we can ensure that the production rules contain as much semantics of browsers as possible. We prove that our input generation preserves the semantics of extracted CFG in Proposition 1. The second advantage is the less occurrence of semantic errors in generated inputs, enabling a more efficient semantics exploration. Different from other browser fuzzers, we infer a context-checking function for every production rule, telling the input generation module which production rules are highly probable to cause semantic errors under a specific derivation tree context. This helps SAGE generate less semantic errors. We prove that PCSG-based input generation is more likely to generate semantically correct derivation trees in Proposition 2.

PROPOSITION 1. *PCSG-based input generation preserves the semantics of extracted CFG.*

PROOF. Let our extracted CFG be $CFG = (N, T, P, S)$ and the transformed PCSG be $PCSG = (\bar{N}, \bar{T}, \bar{P}, \bar{S})$. According to the transformation rule described in Section 3.2, \bar{N} is identical to N , \bar{T} is identical to T , \bar{S} is identical to S , and every $\bar{p} \in \bar{P}$ is derived from $p \in P$ but in the form $[C_{\bar{p}}]\alpha \rightarrow \beta_1\beta_2 \dots \beta_n$, where $C_{\bar{p}}$ is a context-checking function. Suppose we are expanding a nonterminal $n \in N$ when generating a derivation tree, and the production rules to expand n are denoted as $P_n = \{n \rightarrow E_i | i \in [1, \dots, g]\} \subset P$, where g is the number of expansion production rules of n . CFG-based input generation will randomly select one rule from P_n . In contrast, PCSG-based input generation will filter out semantically-incorrect production rules under the current context according to the context-checking function C (Line 14-16 in Algorithm 2), and randomly select one rule from the remaining rule set P'_n . According to the construction of context-checking functions, C only filters out the rules that are never semantically correct under this derivation tree context on the basis of our observed occurrence statistics. Therefore, every filtered-out production rule $p \in P_n - P'_n$ will be highly probable to cause a semantic error under the current context. As a result, P'_n preserves the semantics of P_n without losing semantically correct production rules. \square

PROPOSITION 2. *PCSG-based input generation is more likely to generate semantically correct derivation trees than CFG-based input generation.*

PROOF. Suppose $N_t = [n_1, n_2, n_3, \dots]$ is the list of expanded nonterminals in a derivation tree t . Let P_t denote the probability that the tree t is correct, and $P(n)$ denote the probability that expanding nonterminal n will not lead to an incorrect generated derivation tree. According to the expansion process detailed in Section 2.1, P_t can be presented as

$$P_t = P(n_1)P(n_2|[n_1])P(n_3|[n_1, n_2])P(n_4|[n_1, n_2, n_3]), \dots$$

Without loss of generality, we investigate the expansion of $n_i \in N_t$. To expand n_i , CFG-based input generation will randomly select one of the expansion production rules of n_i . Suppose the number of the expansion rules is k , and w of the rules lead to a semantic error under the current derivation tree context, then $P^{CFG}(n_i|[n_1, \dots, n_{i-1}]) = 1 - \frac{w}{k}$. Our PCSG-based input generation filters out semantically-incorrect production rules according to context-checking functions (Line 14-16 in Algorithm 2), so it will randomly select one rule from fewer production rules, i.e.,

$P^{PCSG}(n_i|[n_1, \dots, n_{i-1}]) = 1 - \frac{w-f}{k-f}$, where f is the number of filtered-out rules and $f \geq 0$. Therefore, $P^{PCSG}(n_i|[n_1, \dots, n_{i-1}]) \geq P^{CFG}(n_i|[n_1, \dots, n_{i-1}])$ holds true for any $n_i \in N_t$. As a result, we can conclude $P_t^{PCSG} \geq P_t^{CFG}$. \square

4 IMPLEMENTATION

We implemented SAGE in around 5k lines of Python code, consisting of three modules, i.e., grammar extraction, semantics inference, and input generation. The grammar extraction module is an end-to-end tool, which is able to update itself as soon as the W3C standards update. This module leverages Webref [W3C 2022c] to obtain the latest stable W3C standards of HTML, CSS, and JavaScript. It parses CSS value syntax (e.g., "none | [`<number>` | `<percentage>`]{2,3}" in Table 1) with the assistance of the CSSTree project [csstree 2022]. It first uses the CSSTree project to convert the syntax into an abstract syntax tree, and then traverses the tree to form production rules of our grammar. As of JavaScript, the module collects WebIDLs from the standards and the source code of browsers, and parses them to form production rules. The semantics inference is built on the top of Domato [Google 2017], with our extracted CFG made compatible to Domato, enabling it to generate inputs. We developed a plugin for Domato to track the derivation trees during its generation. To collect semantic correctness of each statement, we wrote a JavaScript runtime library inside browsers, which catches the exit status of each JavaScript statement. To obtain the semantic correctness of CSS and HTML, the library checks if the corresponding CSS class name and HTML element ID exist. With the derivation tree and semantic correctness, the semantics inference module concludes context-checking functions for every production rules. In our implementation, we regard a context is invalid for a production rule if it occurs more than 10 times in our statistics while none of the occurrences is correct. We append such a context to the invalid context tree of the corresponding production rule, as shown in Fig. 6 The input generation module is also implemented as a plugin on the top of Domato.

5 EVALUATION

In this section, we conduct experiments to evaluate the effectiveness of SAGE. Our evaluation addresses the following research questions:

- **RQ1:** Can SaGe uncover critical bugs in production-level browsers? (Section 5.1)
- **RQ2:** How well does SaGe perform compared to other state-of-the-art browser fuzzers? (Section 5.2)
- **RQ3:** How many additional production rules does the grammar extraction module extract from browser standards? (Section 5.3)
- **RQ4:** How do the grammar extraction and semantics inference modules individually improve input generation performance? (Section 5.4)
- **RQ5:** Will the input generation module introduce additional overhead? (Section 5.5)

Experiment Setup. All experiments run on a machine equipped with an i9-10900K with 10 cores, running Ubuntu 20.04 LTS. We utilized X virtual frame bffer (Xvfb) to enable browsers to run in headless mode on separate virtual display hardware. We selected the recent versions of three mainstream browsers, namely Safari, Chrome, and Firefox, as our fuzzing targets. However, since Safari cannot be run on a Linux system, we use WebKitGTK, a full-featured port of Safari's rendering engine, as an alternative. Note that SAGE only focuses on the browser engines of Safari, Chrome, and Firefox, namely WebKit, Blink, and Gecko, respectively. Other components, such as bookmark or extension managers, are outside the scope of our research. We ran grammar inference modules for 24 hours with 4 cores to obtain PCSG, which is the grammar SAGE uses in our experiments.

5.1 Bug Finding

We spent around 10*30 CPU-days applying our prototype of SAGE to test the latest versions of the three mainstream browsers with AddressSanitizer (ASan) [Serebryany et al. 2012] compiled. Table 2 shows the detail of bugs found by SAGE. To sum up, SAGE found 62 unique bugs, 40 of which were confirmed and 27 fixed. Ten of the fixed bugs are assigned to CVE IDs because they are security-critical. 13 of the reported bugs are marked as duplicated because the bugs were found by other fuzzers before we reported them. Since all the mainstream browsers have been continuously tested by vendors using existing fuzzers for several years, we can regard the ones that are not marked as duplicated are uniquely found by SAGE.

Table 2. Bugs detected by SAGE.

ID	Browser	Bug Type	Bug Location	Status
1	Safari (WebKit)	Use After Free (9 bugs)	WebCore::RenderLayer	CVE-2023-25361
2			WebCore::RenderLayer	CVE-2023-25358
3			WTF::TypeCastTraits	CVE-2023-25359
4			WebCore::RenderLayer	CVE-2023-25362
5			WebCore::RenderLayer	CVE-2023-25363
6			WebCore::AXObjectCache	CVE-2022-26710
7			WebCore::IDBServer::UniqueIDBDatabase	CVE-2022-26709
8			WebCore::TextureMapperLayer	CVE-2022-30294
9			WebCore::RenderLayer	CVE-2023-25360
10	Buffer Overflow (1 bug)	WebCore::TextureMapperLayer	CVE-2022-30293	
11	Null Dereference (6 bugs)	WebCore::RenderLayerCompositor	Confirmed	
12		WebCore::RenderLayerCompositor	Confirmed	
13		WTF::Atomic	Fixed	
14		WebCore::WebGLRenderingContextBase	Fixed	
15		WebCore::RenderTreeBuilder	Fixed	
16		WebCore::Node	Fixed	
17	Abnormal Crash (1 bug)	WebCore::AccessibilityObject	Fixed	
18	Out Of Memory (1 bug)	gin::V8Initializer	Confirmed	
19	Chrome (Blink)	Null Dereference (2 bugs)	mojom::MojoAudioOutputIPC	Fixed
20			blink::RenderAudioOutputStreamFactory	Fixed
21		SIGILL ILL_ILLOPN (1 bug)	blink::NGPhysicalLineBoxFragment	Fixed
22		SEGV MAPERR (1 bug)	blink::ViewTransitionStyleTracker	Duplicated
23		Assertion Failure (27 bugs)	blink::EventHandlerRegistry	Confirmed
24			blink::ClampScrollbarToContentBox	Confirmed
25			blink::LayoutBox	Duplicated
26			blink::ComputeContentSize	Reported
..		
49			blink::LayoutFlowThread	Confirmed
50		Abnormal Crash (3 bug)	webrender::picture	Confirmed
51	nsCSSFrameConstructor		Confirmed	
52	mozilla::ipc		Reported	
53	Null Dereference (1 bug)	mozilla::gfx	Confirmed	
54	Firefox (Gecko)	Assertion Failure (9 bugs)	mozilla::SVGUtils	Confirmed
55			mozilla::dom	Duplicated
56			mozilla::nsLineLayout	Confirmed
57			mozilla::nsDisplayItem	Reported
..		
62		mozilla::nsFieldSetFrame	Confirmed	
Total 62 bugs; 13 were duplicated with others; 40 were confirmed, out of which 27 fixed with 10 CVE				

In terms of bug type, SAGE found 9 use-after-free bugs, 1 buffer-overflow bug, 6 null-dereference bugs, and 1 abnormal crash in Safari; 1 out of memory bug, 2 null-dereference bugs, 1 SIGILL bug, 1 SEGV bug, and 27 assertion failures in Chromium; 3 abnormal crashes, 1 null-dereference bug, and 9 assertion failures in Firefox. Besides, SAGE is capable of triggering bugs located in various locations. The “Bug Location” column in the Table 2 lists the namespace of the root cause line of each bug in the source code. We can see that in addition to all kinds of HTML-based rendering

Listing 1. Minimized code snippet of bug sample 1. It triggers a Heap Use-After-Free bug on Safari when parsing the HTML body. It was assigned CVE-2022-26710 and has been acknowledged by Apple Inc.

```

1  <html>
2  <head>
3    <style>
4      .class2 {
5        -webkit-text-security: square; content: url(#htmlvar00001);
6      }
7      #htmlvar00003 {
8        position: fixed;
9      }
10   </style>
11   <script>
12   </script>
13 </head>
14 <body>
15   <input id="htmlvar00003"></input>
16   <ul id="htmlvar00009">
17     <svg id="svgvar00001">
18       <text id="svgvar00012" visibility="hidden" class="class2">e1;k=g9</text>
19     </svg>
20     <li id="htmlvar00018">
21       <canvas id="htmlvar00022">Xj/e5)^x7\&quot;!sBXhU^;3</canvas>
22     </li>
23   </ul>
24 </body>
25 </html>

```

Listing 2. Root cause in Safari responsible for bug sample 1.

```

1 void AXObjectCache::textChanged(AccessibilityObject* object) { //input a freed pointer
2   if (!object) // object is a freed pointer instead of a NULL pointer
3     return;
4   + Ref<AccessibilityObject> protectedObject(*object); // patch
5
6   for (auto* parent = object; parent; parent = parent->parentObject()) {
7     ...
8     postNotification(object, object->document(), AXTextChanged); //trigger the UAF bug
9     ...
10  }
11 }

```

logic, SAGE also explores bugs of other locations, including web template framework (#3 and #13), WebGL (#14), IDBServer (#6), and IPC (#19 and #52).

Bug Sample 1: A Ten-Year-Old Bug. SAGE detected a use-after-free bug in the accessibility backend logic of Safari (ID 6 in Table 2), which was introduced ten years ago. As shown in Listing 1, the minimized code snippet contains only a few common HTML elements like *input*, *svg*, and *canvas*, along with some visualization-related CSS styles like *-webkit-text-security*, *position*, and *content*. The root cause of the bug, as shown in Listing 2, is a wrong manipulation of the accessibility tree. Web browsers maintain an accessibility tree that corresponds to the DOM tree to enable assistive technologies like screen readers to read web page contents. Whenever a text is changed on the page, the browser manipulates the accessibility tree to align it with the DOM tree. In this case, the *text* element is first deleted from the accessibility tree by the browser due to the *visibility* attribute and layout arranged by other elements. Next, because of the URL-based *content* CSS style, the *text* element has content again, and the browser calls the `AXObjectCache::textChanged(AccessibilityObject*)` function (Line 1 in Listing 2) with a freed object pointer, resulting in a use-after-free bug (Line

8). Although the function has a sanity NULL checker for the input (Line 2-3), a freed pointer can still bypass the checker.

By using `git blame` to track the history of the buggy code, we found that this bug was introduced in 2012. After we reported the bug to Safari, Apple developers added a patch to check the validity of the input pointer (Line 4 in Listing 2). This bug was assigned CVE-2022-26710 due to its severe security consequences and has been acknowledged by Apple Inc. Triggering this bug requires a fuzzer to generate highly structured inputs. First, the fuzzer's grammar must include a full set of HTML tags, HTML attributes, and CSS styles, enabling it to create inputs with highly semantic logic. Most importantly, the fuzzer should generate a high proportion of semantically-correct inputs, so that it can generate such a nested input within a reasonable time. SAGE satisfies the two requirements and was able to trigger this bug within 24 hours. This is particularly impressive given that the vendor had previously leveraged other browser fuzzers to continuously test Safari.

Bug Sample 2: An Invalid-Character Caused Bug. SAGE detected a null pointer dereference bug of Chrome (ID 20 in Table 2). As shown in Listing 1, the minimized code snippet is quite simple: the HTML part creates an *audio* element, and the JavaScript part accesses this element, and calls the *setSinkId* member function of this element with a random *String* input. When executing the *setSinkId* function, Chrome crashes. The root cause of the bug, shown in Listing 4, is the lack of invalid character checking in the inter-process communication (IPC) of the browser. The main process of Chrome is supposed to send the sink ID to the rendering process (Line 8 in Listing 4), but

Listing 3. Minimized code snippet of bug sample 2. It triggers a null pointer dereference bug on Chrome.

```

1  <html>
2  <head>
3    <style></style>
4    <script>
5      var var00015 = document.getElementById("htmlvar00015");
6      var var00022 = var00015.setSinkId(String.fromCharCode(391438, 34928, 730218,
7        790627, 696715, 922786, 667983, 501049, 602320, 1058675, 499878, 204162, 683030,
8        1091862, 677148, 194695, 6097, 565610, 57127, 488118)); // crash here
9    </script>
10 </head>
11 <body>
12   <audio id="htmlvar00015"> ... </audio>
13 </body>
14 </html>

```

Listing 4. Root cause in Chrome responsible for bug sample 2.

```

1  // the sending logic of main process
2  void SetSinkIdResolver::Start() {
3    + if (sink_id_.Utf8(WTF::kStrictUTF8Conversion).empty() != sink_id_.empty()) { ... } // patch
4  }
5  if (sink_id_ == HTMLMediaElementAudioOutputDevice::sinkId(*element_))
6    Resolve();
7  else
8    StartAsync(); // send RPC
9  }
10
11 // the receiving logic of rendering process
12 void MojoAudioOutputIPC::DoRequestDeviceAuthorization(..., const std::string& device_id,
13   ...) {
14   RequestDeviceAuthorization(..., String::FromUTF8(device_id)); // trigger the crash
15 }

```

the sink ID we set is a string with random characters that is not a valid UTF-8 string. Consequently, the rendering process fails to deserialize the message and crashes (Line 13 in Listing 4).

By using `git blame` to track the history of the buggy code, we found that this bug was introduced in 2019. After we reported the bug to Chrome, Google developers added a patch to ensure that the input sink ID should be a UTF-8 string (Line 3 in Listing 2). Most existing browser fuzzers cannot generate such a code, as the grammars of Domato [Google 2017] and FreeDom [Xu et al. 2020] do not include the `setSinkId` member function of the `audio` element due to their handwritten nature. Although the grammars of Favocado [Dinh et al. 2021] and Minerva [Zhou et al. 2022] include the `setSinkId` function, their generation strategies cannot efficiently generate semantically-correct inputs. Favocado failed to trigger this bug within 24 hours, and Minerva took much longer time (3.43 hours) to trigger this bug. In contrast, SAGE can trigger this bug within six minutes.

5.2 Comparison to Other Browser Fuzzers

We evaluate SAGE compare to four state-of-the-art browser fuzzers, i.e., Domato [Google 2017], FreeDom [Xu et al. 2020], Favocado [Dinh et al. 2021], Minerva [Zhou et al. 2022]. Table 3 demonstrates their characteristics. SAGE differs these approaches in several aspects. Firstly, its grammar is automatically extracted from W3C standards, enabling the fuzzer to explore a wider range of semantics. Secondly, it takes derivation context into account, allowing the fuzzer to mitigate compatibility errors, misuse errors, and reference errors. Other characteristics, i.e., variable-context awareness and type error mitigation, are derivate from our baseline tool Domato.

Table 3. Characteristics of existing browser fuzzers.

Fuzzer	Year	Grammar Origin	Variable Context	Derivation-Tree Context	Type Error Mitigation	Compatibility, Misuse, and Reference Error Mitigation
Domato	2017	handwritten	✓	-	✓	-
FreeDom	2020	handwritten	✓✓	-	✓	-
Favocado	2021	automatic	✗	-	✓	-
Minerva	2022	semi-auto	✓	-	✓	-
SAGE	2023	automatic	✓	✓	✓	✓✓✓

We used three different versions of each of the three mainstream browsers as our fuzzing targets. Our evaluation metrics are edge coverage, semantic correctness rate, time to trigger the same coverage, and time to trigger unique bugs. The edge coverage is collected using SanitizeCoverage (SanCov) [llvm 2022]. Specifically, we append `-fsanitize-coverage=trace-pc-guard` in compilation flags when building the three mainstream browsers. The semantic correctness rate is calculated as the fraction of the number of semantically-correct statements to the total input statements. Unique bugs are triaged based on the root cause lines of call stacks. Each experiment runs in 24 hours and repeated five times. Note that all of these fuzzers are generation-based, so they do not need initial seeds. Table 4 shows average time taken by browser fuzzers to trigger unique bugs. Favocado does not trigger any bugs, so we omit it in the table. Table 5 presents the comparison of semantic correctness rate, and Table 6 presents the coverage improvements of SAGE compared to other fuzzers. Fig. 7 shows the time required by SAGE to reach the same coverage as other fuzzers' runs of 1, 3, 6, 12, 24 hours.

v.s. Baseline (Domato). Domato is the most well-known browser fuzzer so far. SAGE is built on the top of Domato, so comparing the performance of SAGE to Domato demonstrates how incorporating our CSG can improve the efficiency of browser fuzzing. Table 4 shows that SAGE can detect all bugs that Domato detects. Additionally, SAGE detects 6 bugs that Domato cannot

Table 4. Average time taken by browser fuzzers to trigger unique bugs in 24 hours over five runs. "# campaign" refers to the number of campaigns that trigger the bug out of the five repeated campaigns. "∞" means that the fuzzer did not trigger the bug within 24 hours.

Browser	Unique Bug	Domato time # campaign	FreeDom time # campaign	Minerva time # campaign	SAGE time # campaign
WebKitGTK-2.36	Assertion Failure 1	24.28min 5/5	2.81h 5/5	3.57h 5/5	30.61min 5/5
	Use After Free 1	1.82h 2/5	3.55h 5/5	∞ 0/5	1.28h 5/5
	Use After Free 2	∞ 0/5	∞ 0/5	∞ 0/5	22.43h 1/5
WebKitGTK-2.37	Null Deref 1	∞ 0/5	14.50h 1/5	20.84h 1/5	1.81h 3/5
WebKitGTK-2.38	Assertion Failure 2	4.56min 5/5	17.66min 5/5	12.2min 5/5	7.55min 5/5
	Null Deref 2	8.29h 2/5	4.69h 5/5	4.24h 5/5	2.51h 5/5
Chrome-98	SEGV MAPERR 1	∞ 0/5	∞ 0/5	∞ 0/5	2.14h 5/5
	SEGV MAPERR 2	∞ 0/5	3.81h 1/5	∞ 0/5	11.72h 1/5
	Null Deref 3	∞ 0/5	∞ 0/5	3.43h 2/5	5.68min 5/5
Chrome-111	SIGILL ILL_ILLOPN 1	∞ 0/5	∞ 0/5	∞ 0/5	19.99h 1/5

detect. Furthermore, SAGE achieves higher coverage and semantic correctness. Table 5 and Table 6 demonstrates that SAGE improves edge coverage and semantic correctness by an average of 20.32% and 3.56%, respectively. Figure 7 illustrates that SAGE requires significantly less time to achieve the same coverage as Domato. Specifically, SAGE takes 1.06 hours to reach the same coverage as Domato's 24-hour run. These improvements are consistent across all versions and browsers. The reason why SAGE outperforms Domato is that the CSG we built indeed helps Domato more efficiently explore the semantics of browsers.

Table 5. Semantic correctness rate comparison of browser fuzzers in 24 hours over five runs.

Browser	Domato	FreeDom	Favocado	Minerva	SAGE
WebKitGTK-2.36	81.35%	62.21%	31.92%	71.24%	84.33%
WebKitGTK-2.37	81.31%	62.59%	31.88%	70.93%	84.31%
WebKitGTK-2.38	80.96%	63.09%	32.93%	69.13%	84.22%
Chrome-98	79.75%	62.54%	32.47%	72.86%	84.66%
Chrome-105	82.19%	63.92%	32.51%	72.00%	84.49%
Chrome-111	81.89%	63.13%	32.45%	71.78%	84.42%
Firefox-101	82.01%	63.35%	32.38%	72.02%	84.36%
Firefox-103	81.98%	63.47%	32.46%	69.81%	84.37%
Firefox-105	81.95%	61.11%	32.31%	72.25%	84.26%
Avg Impr	↑ 3.56%	↑ 33.85%	↑ 160.71%	↑ 18.31%	-

v.s. Context-Aware Fuzzer (FreeDom). FreeDom is a variable-context aware browser fuzzer, which leverages a custom intermediate representation (named FD-IR) to store context information of both locally and globally generated variables. The variables are stored in a tree structure to simulate the DOM tree of browsers. Different from FreeDom, SAGE focuses more on the context of derivation trees during production rule selection in order to generate more semantically-correct inputs. Table 4 shows that SAGE can detect all bugs that FreeDom detects, but does so in less time. Additionally, SAGE detects 4 bugs that FreeDom cannot detect. Furthermore, SAGE achieves higher coverage and semantic correctness. Table 5 and Table 6 demonstrates that SAGE improves edge coverage and semantic correctness by an average of 24.77% and 33.85%, respectively. Figure 7 illustrates that SAGE requires significantly less time to achieve the same coverage as FreeDom. Specifically, SAGE takes 0.60 hours to reach the same coverage as FreeDom's 24-hour run. These improvements are consistent across all versions and browsers. There are two reasons why SAGE outperforms FreeDom. Firstly, the grammar used in SAGE covers a wider range of browser functionalities, better than the handwritten grammar of FreeDom. Secondly, FreeDom does not make use of interactive

information from browsers to refine its FD-IR, while SAGE leverages this information to improve its extracted grammar.

Table 6. Coverage improvements of SAGE compared to other browser fuzzers in 24 hours over five runs.

Browser	v.s. Domato			v.s. FreeDom			v.s. Favocado			v.s. Minerva		
	max-impr	avg-impr	min-impr	max-impr	avg-impr	min-impr	max-impr	avg-impr	min-impr	max-impr	avg-impr	min-impr
WebKitGTK-2.36	14.37%	13.88%	13.13%	18.17%	17.88%	17.66%	523.33%	444.83%	264.68%	7.48%	6.72%	6.01%
WebKitGTK-2.37	14.51%	13.87%	12.89%	17.56%	16.84%	15.17%	502.90%	437.85%	277.01%	7.67%	6.96%	6.37%
WebKitGTK-2.38	14.50%	13.24%	11.18%	17.49%	16.67%	15.05%	497.68%	393.04%	258.80%	7.74%	6.88%	5.67%
Chrome-98	32.78%	31.12%	27.08%	34.75%	33.62%	31.17%	417.42%	406.18%	369.30%	6.74%	5.46%	3.24%
Chrome-105	35.32%	34.25%	33.71%	40.59%	39.17%	38.39%	172.22%	167.27%	163.65%	10.73%	9.76%	8.94%
Chrome-111	32.29%	31.56%	30.95%	44.77%	40.72%	36.94%	334.80%	332.37%	329.81%	7.43%	6.19%	5.16%
Firefox-101	15.83%	14.83%	13.44%	26.96%	21.23%	18.61%	105.94%	105.05%	102.55%	5.14%	4.68%	3.56%
Firefox-103	14.05%	13.72%	13.51%	18.10%	17.00%	15.59%	105.90%	105.90%	105.90%	2.45%	2.33%	2.19%
Firefox-105	17.20%	16.45%	14.16%	21.19%	19.80%	15.53%	110.72%	107.68%	103.33%	6.84%	5.25%	2.63%
Avg Impr	↑ 20.32%			↑ 24.77%			↑ 277.80%			↑ 6.03%		

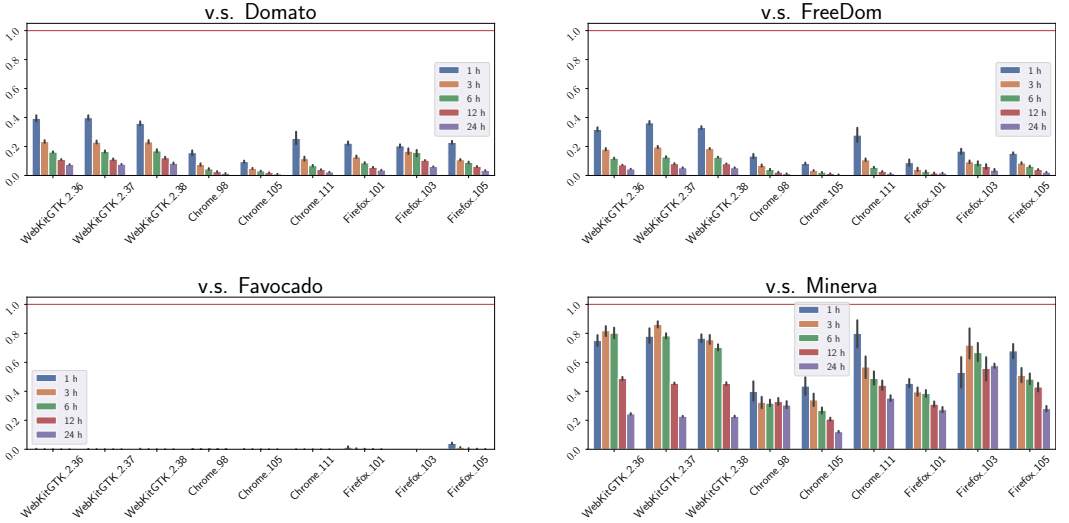


Fig. 7. Normalized execution time required by SAGE to reach the same coverage as other fuzzers' runs of 1, 3, 6, 12, and 24 hours. The X axis is browsers, the Y axis is the ratio between the execution times required by SAGE and the compared fuzzer for reaching the same coverage. A ratio of less than 1.0 indicates that SAGE is faster at reaching the target coverage.

v.s. Semantics-Aware Fuzzer (Favocado). Favocado is a semantics-aware browser fuzzer, focusing on improving type correctness of its generated inputs. In contrast, SAGE goes beyond type correctness and strives to ensure the correctness of other semantic types as well. Table 4 shows that Favocado does not detect any bugs within 24 hours, while SAGE detect all 10 bugs. Furthermore, SAGE achieves higher coverage and semantic correctness. Table 5 and Table 6 demonstrates that SAGE improves edge coverage and semantic correctness by an average of 277.80% and 160.71%, respectively. Figure 7 illustrates that SAGE requires significantly less time to achieve the same coverage as Favocado. These improvements are consistent across all versions and browsers. The reason why SAGE outperforms Favocado is that, although Favocado also extracts semantic information from browsers' source code, it does not make good use of them. Its semantic information is maintained in a JSON-like structure instead of CFG, which make it difficult to effectively generate

highly-structured inputs. Additionally, its flawed implementation makes it challenging to bypass syntactic checks. Although we made our best effort to fix the flaws in its implementation, the inputs generated by Favocado are still prone to containing syntactic errors, which are easily rejected by browsers. Consequently, Favocado can only explore a limited set of browser backend logic.

v.s. API-Oriented Fuzzer (Minerva). Minerva is a browser fuzzer which tries to analyze the relations of JavaScript APIs in order to generate highly-relevant API invocations. In contrast, SAGE infers semantics of each production rule to explore the semantics of browsers during the fuzzing process more efficiently. These two fuzzers target at total different fuzzing goals: Minerva aims at generating memory-related API invocations in a test case so that it has higher possibility to trigger memory bugs, i.e., focusing on depth of exploration; while SAGE focuses more on the breadth of exploration. Table 4 shows that SAGE can detect all bugs that Minerva detects, but does so in less time. Additionally, SAGE detects 5 bugs that Minerva cannot detect. Furthermore, SAGE achieves higher coverage and semantic correctness. Table 5 and Table 6 demonstrates that SAGE improves edge coverage and semantic correctness by an average of 6.03% and 18.31%, respectively. Figure 7 illustrates that SAGE requires significantly less time to achieve the same coverage as Minerva. Specifically, SAGE takes 9.59 hours to reach the same coverage as Minerva's 24-hour run. These improvements are consistent across all versions and browsers.

5.3 Statistics of Grammar Extraction

In total, our grammar extraction module extracted 41,031 production rules from W3C standards. The detailed statistics are listed in Table 7. Our extracted grammar is significantly richer in semantics compared to the original Domato's grammar. It covers a wider range of CSS properties, JS interfaces, member variables/functions, and HTML attributes. Moreover, our extracted grammar includes more value options, which allows fuzzers to generate a more diverse set of inputs. For instance, Although Domato includes *counter-increment* CSS property, it only provides one value option for this property, which only allows one possible CSS statement, i.e., "*counter-increment: c*". In contrast, our extracted grammar provides a more comprehensive syntax "<counter-name> <integer>?]+ | none", enabling fuzzers to generate statements like "*counter-increment: none*", "*counter-increment: c1 10*", and "*counter-increment: c1 10 c2 -20*". With the extracted grammar, SAGE is able to achieve better edge coverage than our baseline Domato as shown in Table 5, demonstrating the effectiveness of the grammar extraction module.

Table 7. Statistics of extracted grammar compared to Domato's grammar.

Statistics	Grammar of Domato	Grammar of SAGE
# Total Production Rule	20,045	43,555
# Covered CSS Property	505	882
# Covered JS Interface	1,657	4,853
# Covered JS Variable/Function	4,544	11,787
# Covered HTML Attribute	218	533

5.4 Effectiveness of Grammar Extraction and Semantics Inference

To evaluate the individual contribution of SAGE's grammar extraction module and semantics inference module, we conducted experiments for ablation study. The results are presented in Table 8. The grammar extraction module aims to enable our fuzzer to explore a more diverse semantics. A comparison between the SAGE^{wo-s} column and the SAGE^{wo-se} column reveals that the grammar extraction module greatly enhances the coverage of individual inputs. However, it inevitably results in a decrease in the rate of semantic correctness because of the increase in

semantic diversity. The semantics inference module complements this by improving the semantic correctness of our extracted grammar, thereby further enhancing the coverage of individual inputs. By comparing the SAGE column to the $SAGE^{wo-s}$ column, we can conclude that the semantics inference module can significantly improve semantics correctness rate.

Table 8. Performance comparison between SAGE w/wo grammar extraction module and semantic inference module.

Browser	Semantics Correctness Rate				Coverage of Individual Inputs			
	SAGE	$SAGE^{wo-s}$	$SAGE^{wo-e}$	$SAGE^{wo-se}$	SAGE	$SAGE^{wo-s}$	$SAGE^{wo-e}$	$SAGE^{wo-se}$
WebKitGTK-2.36	84.33%	58.59%	84.40%	81.35%	103185	84229	55967	46243
WebKitGTK-2.37	84.31%	57.20%	84.69%	81.31%	99679	78461	51382	43213
WebKitGTK-2.38	84.22%	57.68%	83.40%	80.96%	102236	83984	54327	46035
Chrome-98	84.66%	56.84%	84.42%	79.75%	183655	154071	118601	108883
Chrome-105	84.49%	57.18%	85.12%	82.19%	182519	146608	119791	112411
Chrome-111	84.42%	58.03%	84.11%	81.89%	183953	150287	122796	114639
Firefox-101	84.36%	57.57%	85.00%	82.01%	147759	125081	105524	95728
Firefox-103	84.37%	57.07%	84.60%	81.98%	155790	136373	116576	101619
Firefox-105	84.26%	57.83%	84.66%	81.95%	153999	138536	107598	103689

$SAGE^{wo-s}$ denotes SAGE without the semantics inference module; $SAGE^{wo-e}$ denotes SAGE without the grammar extraction module; $SAGE^{wo-se}$ denotes SAGE without both modules, which is identical to Domato.

5.5 Overhead of Input Generation

The main overhead introduced by the input generation module is the use of the context-checking function C to ensure that a selected rule is semantically correct under a given context (Line 14-16 in Algorithm 2). It is necessary to assess the average time required by SAGE to generate an input compared to our baseline, Domato. We use the two fuzzers to repeatedly generate 1000 inputs to evaluate the average time required for SAGE and Domato to generate an input, with all settings, e.g., the maximum number of generated lines, kept consistent with Domato's default setting. On average, SAGE takes 0.1736 seconds to generate an input, while Domato takes 0.1360 seconds. Comparatively, browsers take an average of 0.9149 seconds to execute an input. Thus, the majority of the time spent in a fuzzing loop is on the browser's execution. From this, we can conclude that the overhead of this module does not significantly impact the overall performance of our fuzzer.

6 DISCUSSION

More comprehensive modeling of context. SAGE currently models the context of a selected production rule as the parent chain in the generated derivation tree (see Definition 3.1). However, it does not consider the siblings of this rule, which could result in the overlooking of certain semantic collisions between rules and their siblings. A more comprehensive modeling of the context is therefore necessary. However, embedding the whole context of derivation trees into a data structure for efficient context checking remains an open problem. In SkyFire [Wang et al. 2017], the context of a production rule is modeled as a tuple of $\langle \text{great-grandparent, grandparent, parent, first sibling} \rangle$, also partially capturing the context. Some research studies [Yin and Neubig 2017; Zhang et al. 2019] have focused on embedding syntax trees into neural networks, which can be used in our modeling. However, this could introduce considerable overhead during input generation. For now, our design made a tradeoff between the efficiency of input generation and comprehensiveness of the considered context. We leave a better context modeling as a future work.

Semantics inference during input generation. In our current design, SAGE performs semantics inference before input generation to assist in generating semantically correct inputs. An

alternative approach is to perform semantics inference during input generation. We highlight that the current design has its advantages. As the number of newly-discovered invalid contexts decreases over time, performing semantics inference during every input generation may become unnecessary. Additionally, the two-phase design allows testers to reuse existing semantics in our CSG, which saves time and resources compared to inferring semantics from scratch. To evaluate the effectiveness of the two approaches, we conducted experiments and found no significant differences between them. The coverage differences percentages are 0.09% in WebKit, 0.69% in Chrome, and 0.12% in Firefox. Therefore, the choice between the two approaches may depend on the specific requirements of the testing process, and our released prototype supports both of the approaches.

False-positives and false-negatives of semantics inference. The semantics inference module is unlikely to introduce false-positives. False-positives refer to cases where the module mistakenly adding a valid context to the invalid context tree of a production rule. The only exception is if a valid context appears more than 10 times for a production rule during semantics inference and none of these occurrences are semantically valid. Such a specific corner case could potentially result in a false-positive being introduced during semantics inference. We did not observe such an exception during our experimental evaluation. Even if there is an exception, we can adjust our add-to-tree threshold (e.g., from 10 to 100) to mitigate these false-positives. On the other hand, the semantics inference module may introduce false-negatives. False-negatives refers to cases where the module overlooks invalid contexts. We observed that there are no more newly-discovered invalid contexts after a 20-hour semantics inference and SAGE have achieved more than 84% semantic correctness rate as shown in Table 5. Therefore, our fuzzer have considered as many invalid contexts as possible.

7 RELATED WORK

7.1 Grammar-Based Fuzzing

Fuzzing has been proven to be a practical technique to explore unknown bugs. A large number of security researchers proposed optimizations from different angles, e.g. boosting coverage tracing [Nagy and Hicks 2019; Wang et al. 2021a, 2022; Zhou et al. 2020], and improving search effectiveness [Chen et al. 2019; Liang et al. 2022; Nguyen and Grunske 2022]. However, traditional fuzzers are ineffective when fuzzing programs requiring highly-structured inputs. To address this challenge, many grammar-based fuzzers [Aschermann et al. 2019; Godefroid et al. 2008; Srivastava and Payer 2021; Wang et al. 2019] are proposed to leverage grammar to make inputs syntactically correct. Some advanced approaches also utilize other information to enhance their grammar-based fuzzers. Skyfire [Wang et al. 2017] proposes a data-driven approach that learns from existing samples to generate well-distributed seed inputs. With a uniform intermediate representation, Poly-Glot [Chen et al. 2021] supports generating valid inputs for different testing targets. Zest [Padhye et al. 2019a] leverages coverage feedback to infer the type of values that are not well-defined in the grammar. These methods attempt to supplement semantic information that is not explicitly defined in the grammar, in an effort to increase the correctness of generated seeds. In addition to generic grammar-based fuzzers, there are also many custom fuzzers, enhanced with domain knowledge, for fuzzing specific domain such as C/C++ compilers [Livinskii et al. 2020; Yang et al. 2011], databases [Rigger and Su 2020; Wang et al. 2021b; Zhong et al. 2020], and deep learning frameworks [Liu et al. 2023]. Different from the generic fuzzers and other domain's fuzzers, SAGE only focuses on generating semantically correct browser inputs with browser domain knowledge.

7.2 Browser Fuzzing

Among all browser fuzzers, Domato [Google 2017] is recognized as the most successful fuzzers that is widely adopted in the industry. Relying on hand-written grammar, it generates extensive

syntactic correct inputs. Subsequently, FreeDom [Xu et al. 2020] maintains an FD-IR to store context information during input generation. Favocado [Dinh et al. 2021] mainly tests browsers' binding code with semantically correct inputs. Minerva [Zhou et al. 2022] tries to explore deeper paths by generating API-dependent invocations with mod-ref relations between APIs. Different from them, SAGE focuses on better explorations of browser's semantics. It extracts semantics from W3C standards, and makes sure that its generated inputs incur fewer semantic errors.

In addition to browser fuzzers, fuzzers for JavaScript engines [Bernhard et al. 2022; Groß et al. 2023; Han et al. 2019; Park et al. 2020] also gained significant traction. Their generation strategies focus more on generating nested JavaScript code, such as code with complex variable lifetimes or multiple loops, in order to explore the interpretation/optimization logic of JavaScript engines. In contrast, browser fuzzers, when generating JavaScript code, focus more on generating meaningful API invocations to test browsers' backend logic, such as DOM and WebGL. In particular, SAGE aims to generate inputs that cover a wide range of browser semantics with fewer semantic errors. Among all the JavaScript fuzzers, COMFORT [Ye et al. 2021] and JEST [Park et al. 2021] are more related to SAGE since they leveraged JavaScript standards. However, they utilize the standards to perform differential testing in order to uncover standard conformance bugs, i.e., inconsistencies of the implementation of JavaScript engine and JavaScript standards. In contrast, SAGE pursues different goals, and employs standards in a different manner. It leverages W3C standards to construct CFG, aiming to cover as many browsers' backend logic as possible.

7.3 Learn Program Semantics for Fuzzing

Learning program's semantics during fuzzing is a popular topic. GRIMOIRE [Blazytko et al. 2019], GLADE [Bastani et al. 2017], and pFuzzer [Mathis et al. 2019] intend to synthesize the structure of inputs when fuzzing a program without the guidance of grammar. In addition, Gopinath et al. introduce the concept of the grammar transformer, which enables developers to create a grammar for fuzzing flexibly. For now, the main challenge of this research topic is learning the input structure of an unknown program. Different from them, SAGE only focuses on browsers, so it can easily ensure its input conforms browser's input structure. It focuses more on semantic correctness.

8 CONCLUSION

This paper presents SAGE, a fuzzer that efficiently explores browser semantics by generating inputs that cover a wide range of browser semantics with fewer semantic errors. Its core idea is to extract a primitive CFG from W3C standards and then enhance it to a PCFG that includes semantic information. Our experimental results show that it achieves significant improvements compared to state-of-the-art fuzzers. During our testing period with SAGE, it has detected 62 bugs on mainstream browsers, out of which 10 are assigned CVE IDs. Our future work will focus on integrating a more comprehensive modeling of contexts to our work.

DATA-AVAILABILITY STATEMENT

The relevant evaluation artifact of this paper is available on Zenodo [Zhou et al. 2023]. We also release the prototype of SAGE at <https://github.com/ChijinZ/SaGe-Browser-Fuzzer>.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003).

REFERENCES

- Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.
- Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.
- Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (Barcelona, Spain) (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 95–110.
- 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, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 351–364.
- Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002.
- Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1967–1983.
- Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 642–658.
- csstree. 2022. CSSTree - A tool set for CSS including fast detailed parser, walker, generator and lexer based on W3C specs and browser implementations. <https://github.com/csstree/csstree>. (visited on September 20, 2022).
- Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.
- Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215.
- Google. 2017. Domato: A DOM fuzzer. <https://github.com/googleprojectzero/domato>. (visited on September 20, 2022).
- Google. 2019. ClusterFuzz. <https://google.github.io/clusterfuzz/>. (visited on September 20, 2022).
- Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. Input Algebras. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 699–710.
- Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.
- HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.
- Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 189–199.
- Igalia. 2021. WPE powers hundreds of millions of embedded devices. <https://wpewebkit.org/>. (visited on September 20, 2022).
- Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jianguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)(SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 154–170.
- Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 530–543.
- Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages.
- llvm. 2022. Clang 13 documentation: SANITIZERCOVERAGE. <https://clang.llvm.org/docs/SanitizerCoverage.html>. (visited on September 20, 2022).

- Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Hörschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 548–560.
- Mozilla. 2005. MDN Web Docs. <https://developer.mozilla.org/>. (visited on September 20, 2022).
- Mozilla. 2015. dharma: Generation-based, context-free grammar fuzzer. <https://github.com/MozillaSecurity/dharma>. (visited on September 20, 2022).
- mozilla. 2021. Web APIs. <https://developer.mozilla.org/en-US/docs/Web/API>. (visited on September 20, 2022).
- Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 787–802.
- Hoang Lam Nguyen and Lars Grunske. 2022. BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 249–261.
- Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019a. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 329–340.
- Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019b. FuzzFactory: domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 174:1–174:29.
- Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 13–24.
- 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 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1629–1642.
- Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682.
- Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 309–318.
- Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 244–256.
- Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nikolai Zeldovich (Eds.). ACM, 344–358.
- W3C. 2022a. W3C Standards. <https://www.w3.org/standards/>. (visited on September 20, 2022).
- W3C. 2022b. WebIDL Level 1. <https://webidl.spec.whatwg.org/>. (visited on September 20, 2022).
- W3C. 2022c. Webref - Machine-readable references of terms defined in web browser specifications. <https://github.com/w3c/webref>. (visited on September 20, 2022).
- 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)*. 579–594.
- Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735.
- Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021a. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 147–159.
- Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1010–1024.
- Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021b. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 328–337.
- Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis. In *ISSTA '22: 31th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, South Korea, July 18-22, 2022*.
- Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti,

- Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 971–986.
- 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* (San Jose, California, USA) (PLDI '11). Association for Computing Machinery, New York, NY, USA, 283–294.
- 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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 435–450.
- Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 440–450.
- Michał Zalewski. 2013. american fuzzy lop. <https://lcamtuf.coredump.cx/afl/>. (visited on September 20, 2022).
- Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*, Joanne M. Atlee, Tefvik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794.
- Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 955–970.
- Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21–25, 2020*. IEEE, 858–870.
- 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. Zenodo. <https://doi.org/10.5281/zenodo.8328742>
- 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, ESEC/FSE 2022, Singapore, Singapore, November 14–18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1135–1147.

Received 2023-04-14; accepted 2023-08-27