Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

DFL: A DOM sample generation oriented fuzzing framework for browser rendering engines[☆]

Guoyun Duan [a,b,*], Hai Zhao [a], Minjie Cai [b], Jianhua Sun [b], Hao Chen [b]

[a] School of Information Engineering, Hunan University of Science and Engineering, Yongzhou 425199, China
[b] College of Computer Science and Electronic Engineering, Hunan University, No. 2 Lushan South Road, Changsha 410082, China

ARTICLE INFO

Keywords:
DOM
Browser security
Gray-box fuzzing
Coverage guided fuzzing
Browser rendering engine

ABSTRACT

The security of web browsers, being fundamental to Internet access infrastructure, has garnered significant attention. Current approaches to identify browser vulnerabilities predominantly rely on code auditing and componentized unit testing. Fuzzing has emerged as an efficient technique for vulnerability discovery. However, adapting this method to browser security testing poses considerable challenges. Recent endeavors in browser vulnerability discovery primarily concentrate on the parsing engine, with limited solutions addressing the rendering engine. Moreover, coverage-guided mutation, a critical aspect, is not prevalent in existing fuzzing frameworks. In this paper, we present a coverage-guided fuzzing framework of DFL, which builds on Freedom and AFL to re-engineer various text generators based on DOM syntax and optimize the efficiency of sample generation. Additionally, serialization and deserialisation methods are developed for the implementation of generator text mutations and the seamless conversion between binary samples and the source DOM tree. When compared with three established DOM fuzzing frameworks in the latest Chromium kernel, DFL has demonstrated an ability to uncover 1.5–3 times more vulnerabilities within a short timeframe. Our research identifies potential avenues for further exploration in browser rendering engine security, specifically focusing on sample generation and path direction.

1. Introduction

Browsers are the most fundamental applications that users to access the Internet. The core functionality of browsers consists of a rendering engine, HTML parser, and a JavaScript (JS) engine [1]. The rendering engine implements the vast majority of functionality of the browser, and it is commonly referred to as the browser kernel (e.g., Chromium/Blink). It harbors some extremely subtle and complex binary vulnerabilities [2], posing serious threats to users security and privacy [3–5]. Discovering and fixing vulnerabilities in rendering engines can effectively prevent security attacks targeting browsers and greatly enhance security for users [6]. Therefore, rendering engine security has become one of the popular research topic in the community.

Nowadays, there are mainly two types of methods for addressing browser vulnerabilities: code auditing and software detection. Code auditing includes manual auditing based on source code and tool auditing based on code detection [7,8]. However, due to the large code base of browsers, manual auditing requires a significant amount of manpower. Code detection requires the construction of efficient query statements and heavily relies on the experience and knowledge of analyzers. Software detection includes functional and security testing, and there are three main testing methods: white-box, black-box, and gray-box. Fuzzing is the most influential testing technique for security testing, and has discovered most of the security vulnerabilities in recent years. Its testing effectiveness is highly dependent on the quality of the input samples and the execution path feedback from the target under test. Thus, generating high-quality samples for the test target with an efficient feedback mechanism is critical for browser security testing.

Currently, browser fuzzing targets are mostly the JS engines [9–18], and there are very few methods to fuzzing the rendering engine. Although AFL [19] can generate test samples to probe the path of the rendering engine, it is impractical to generate all the samples in this way considering the vast body of HTML syntax. Invalid sample inputs can trigger the rendering engine's validation checks, causing the browser to terminate upon detecting syntax errors. This prevents the exploration of deeper code paths, deviating from the original intent of testing [20]. Therefore, code auditing [7] and componentized unit

☆ This research was supported by the Hunan Provincial Natural Science Foundation of China under Grants 2022JJ20015 and 2023JJ50420.

* Corresponding author at: School of Information Engineering, Hunan University of Science and Engineering, Yongzhou 425199, China.
E-mail addresses: dguoyun@hnu.edu.cn (G. Duan), ha1vk@foxmail.com (H. Zhao), caiminjie@hnu.edu.cn (M. Cai), jhsun@hnu.edu.cn (J. Sun), haochen@hnu.edu.cn (H. Chen).

https://doi.org/10.1016/j.infsof.2024.107591
Received 25 January 2024; Received in revised form 26 September 2024; Accepted 27 September 2024
Available online 1 October 2024

testing [7,21–24] remain the primary methods for discovering browser vulnerabilities.

To the best of our knowledge, there are very few works using fuzzing methods specifically for rendering engine security [25–27], such as FreeDom [28] and DOMato [29]. FreeDom proposes a context-aware document generation approach. While the idea of coverage-guided mutation is realized to some extent, it has very little impact on the discovery of browser bugs. To perform security testing of the rendering engine, we identify four main technical challenges. First, there is no fuzzing framework specifically designed for the rendering engine. Secondly, the operations in the rendering engine follow a strict set of syntax rules, and there is a lack of sample generators that follow the DOM syntax. The structure of HTML test samples is quite different from other fuzzing scenarios (e.g., TEE [30]). A specialized sample generator needs to be designed for the test scenarios of the rendering engine. Third, coverage-guided fuzzing techniques have achieved good results in identifying vulnerabilities of OS kernels [31–33], but the technique has rarely been introduced in rendering engine fuzzing frameworks. Finally, popular fuzzing frameworks (e.g., AFL [19]) usually generate samples blindly, which is inefficient in producing valid HTML test cases following certain syntax rules.

To address the above challenges, this paper focuses on studying the principles of DOM parsing in the rendering engine and proposes a Document-object-model Fuzzy Loop (DFL) framework specifically designed for the rendering engine. First, we adopt the AFL [19] fuzzing framework, and integrate the ideas of DOM sample generator from FreeDom [28]. Second, a comprehensive HTML Intermediate Representation Generator (HIRG) for constructing HTML samples are designed based on DOM syntax. Third, AFL's coverage guide method and DFL's fork service are abstracted as instrumented functions, which provides feedback on path coverage information as samples are executed. Finally, a caching algorithm is used to optimize the speed of JS API calls, and a set of serialization and deserialization algorithms are designed to store and transform HTML samples.

DFL is different from existing rendering engine fuzzing methods in that we focus on the efficiency and quality of sample generation while ensuring the effectiveness of the fuzzing framework. At the same time, sample generation is integrated with AFL to form a complete testing framework. Experiments are conducted to compare DFL with existing DOM fuzzing tools using the latest version of Chromium. The experimental results show that DFL can discover more vulnerabilities in the rendering engine, including vulnerabilities not recognized by comparative frameworks such as FreeDom. One of the newly discovered vulnerability has been already been confirmed by Google. The experiments demonstrate that DFL solves the challenges of sample construction and coverage collection, and achieves better results for testing rendering engines.

In summary, our main contributions are as follows:

- A coverage-guided fuzzing framework, named as DFL, for the rendering engine is proposed. It is built on top of AFL to specifically fuzzing the rendering engine in the browser. A set of strategies is proposed to improve fuzzing performance.
- A HTML sample generator is designed by incorporating the idea of template-based sample generation and context-aware attribute generation.
- A coverage module is implemented and integrated into the browser through instrumentation to collect path coverage at runtime.
- Serialization and deserialization methods of the DOM tree are developed to address the issue in existing rendering engine fuzzing frameworks where HTML samples could not be parsed into syntax tree objects.

The rest of this paper is organized as follows. The commonly used sample construction techniques and fuzzing methods are described in Section 2. The motivations for developing DFL are elaborated in Section 3. In Section 4, we present the design and implementation of DFL in detail. In Section 5, we conduct performance evaluation on DFL. Section 6 discusses limitations of DFL and the future work in fuzzing rendering engine in browsers.

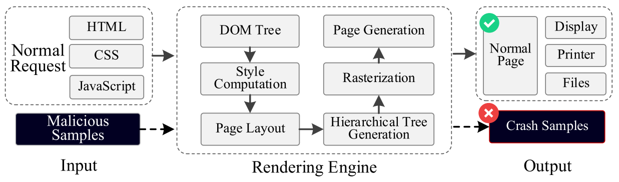


Fig. 1. The work flow of the rendering engine. Solid arrows indicate normal data flow and dashed arrows indicate attack data flow.

2. Background

In this section, we first introduce the security of DOM, followed by seed construction techniques and common types of vulnerabilities in browsers. Finally, we provide a detailed discussion on the popular vulnerability discovery method, fuzzing, currently employed in the field.

2.1. DOM security

Document Object Model (DOM) is an interface for scripting languages (e.g., JS) to manipulate HTML pages. It organizes the content of an HTML page using a tree-like structure and is independent of platforms and languages. At the end of each tree branch is a node containing a list of attributes. Each node in the tree is associated with an event handler. By manipulating node attributes, the content and style of the document can be modified, thereby triggering the execution of the business processing logic. The DOM tree is rendered by the rendering engine to produce beautifully crafted HTML pages.

The flexibility of the DOM structure gives the page designer room to maneuver. At the same time, it also facilitates attackers to construct DOM samples with various attack signature, such as DOM Clobbering [34], Scriptless attacks [35], DOM-XSS [36], and XS-Leaks samples generation [37]. It becomes easier for attackers to construct malicious samples, which is a significant source of browser crashes. Meanwhile, it also provides opportunities for the mutation of test cases in fuzzing. Submitting benign or malicious DOM samples to the browser is a common practice among security researchers for conducting security testing. In this paper, we study techniques of constructing DOM samples, which are combined with current vulnerability detection methods to investigate the security of rendering engine.

2.2. Rendering engine security

According to the statistics provided in [1], 67.4% of web browser vulnerabilities are discovered in the rendering engine. The rendering engine is responsible for interpreting the content of HTML sample, including the web page's structure, layout, and interactive behavior, and outputs the rendered web page. The work flow of the rendering engine is shown in Fig. 1.

A large amount of research has shown [38–40] that the effectiveness of fuzzing depends largely on the quality of the input samples. According to the characteristics of the test target, vulnerabilities can be triggered to a large extent by constructing and inputting potentially malicious samples. Therefore, the primary requirement for triggering rendering engine vulnerabilities is the construction of malicious HTML samples [41]. In this work, we focus on uncovering vulnerabilities in the rendering engine, by constructing samples specifically designed to

Table 1
Comparison of browser fuzzing frameworks.

Framework[a]	Year	Coverage guidance	Sample serialization	Elemental coverage	Context-sensitive	Sample generation method	Sample test method[b]	Framework integrity[c]	Development languages
Wadi [44]	2016	✗	✗	Incomplete	✗	Template	S	✗	JavaScript
DOMato [29]	2017	–	✗	Incomplete	✗	Template	S	✗	Python
FreeDom [28]	2020	✓	✓	Near complete	✓	Syntax	S	✗	Python
DFL	2024	✓	✓	Complete	✓	Syntax and Template	F	✓	C++

[a] Some of the results used for comparison in the table are from Favocado [10], SAGE [38] and the corresponding frameworks source code.
[b] S for semi-automatic or manual and F for automated testing using the ForkServer module.
[c] Whether the framework can perform automated testing on the browser after completing data collection and sample generation.

trigger rendering engine vulnerabilities or explore different execution paths.

Numerical vulnerability. When a browser parses HTML content, certain data fields of type integer may become array indices or the length of arrays. During the rendering process, it is possible that there is no index out-of-bounds check or integer overflow check for these fields. The generation of malicious samples for such vulnerabilities requires the use of a strategic integer generator, where the boundary values of the generated integers have higher weights. For float types, a similar approach is adopted to increase the occurring probability of boundary values and special values.

Heap vulnerability. There are a variety of heap vulnerabilities [42, 43] in operating system kernels, and these types of vulnerabilities also exist in the rendering engine, such as use-after-free (UAF). During the process of parsing HTML samples, the browser may allocate or release a significant amount of heap memory. There might be cases where a previously released heap memory is mistakenly released again, or the released heap memory is inadvertently reused, and such vulnerabilities often occur in JS APIs.

Logical vulnerability. Various types of logical errors can exist in various scenarios such as operating system kernels [45], web applications [46], desktop applications [47], and industrial control systems [48]. Browsers are no exception. Programmers may inadvertently introduce hidden vulnerabilities due to incorrect implementation of specific functions or poor logical design of algorithms. They may result in system crashes or be exploited by malicious actions.

2.3. Fuzzing

Fuzzing is an efficient and effective method for automated software testing. AFL [19] and Syzkaller [49] are two popular fuzzing frameworks in use today. The basic fuzzing process begins by randomly generating a large number of seeds. The seeds include HTML documents with Cascading Style Sheets (CSS) and JS codes for the rendering engine. These seeds are often filtered and mutated to generate high-quality test samples in a fuzzing framework. Then, the test samples are handed over to the target program for execution, while the program's execution status is simultaneously monitored. Once a crash event is detected, there is a high possibility of the program having a vulnerability. Finally, the samples causing crashes are retained for further analysis and vulnerability identification by researchers [50]. Fuzzing is increasingly receiving a lot of attention from both academia and industry, and has achieved remarkable results in vulnerability discovery.

There are three main ways of fuzzing: white-box fuzzing, black-box fuzzing, and gray-box fuzzing [51,52]. **White-box fuzzing** is used in scenarios where the source code of the target program is known. It is commonly used in lightweight testing due to problems such as path explosion. **Black-box fuzzing** focuses only on the inputs and outputs of the target program, whose internal structures are not considered.

Gray-box fuzzing is a testing methodology between white-box fuzzing and black-box fuzzing, where the tester possesses a small amount of internal information about the target program (e.g., source code, structure, or execution information), and is often used in the software integration testing phase. Unlike white-box testing, this technique is subject to various limitations during the testing process. It often relies on instrumentation techniques to collect coverage information of code branches during execution of the target program, and the coverage is further utilized to guide the fuzzing process [50]. A representative is the AFL fuzzing framework, released in 2013 [19].

Table 2
Percentage of effective samples generated.

Generator	#Sample	#Effective sample	Percentage
Wadi	500	349	69.80%
FreeDom	500	207	41.40%
DOMato	500	52	10.40%

3. Motivation

In recent years, many sample generating tools (e.g. FreeDom [28], DOMato [29], Wadi [44]) for browser security testing have been introduced (shown in Table 1). However, there are still challenges due to the complex code of the browser and the huge input space, making it difficult for the tool functionality to cover all the logic in the back-end of the browser. The generated samples are difficult to fully cover the HTML syntax, and the released frameworks rarely integrate advanced coverage-guided feedback techniques in the fuzzing domain, making vulnerability discovery inefficient (see Fig. 10). To tackle these challenges, we design a new framework DFL with three motivations of optimizing the syntax generator, introducing coverage technique, and improving the fuzzing efficiency, which are explained below.

Syntax generator. As we all know, native AFL [19] is difficult to use directly for browse fuzzing because it cannot generate HTML samples that fully conform to the rendering syntax. State-of-the-art browser fuzzing frameworks mainly employ (semi-)handwritten context-free grammars (CFG) (e.g., DOMato [29], Minerva [26], SAGE [38]) and context records (such as FreeDom [28]) to generate structured inputs. These approaches improved the syntactic quality of the samples to a large extent, but does not guarantee the quality of the samples. With an empirical study, we find that these generators still have much room for improvement (see Table 2). High-quality samples can trigger deep vulnerabilities, and there are two main construction methods, machine learning and manual rule collection. Machine learning methods [53, 54] rely on large-scale datasets and computational resources, and the generated outputs lack interpretability and are not tunable. Manually summarizing syntax rules (e.g., FreeDom [28]) is good at precisely controlling the structure and semantics of the generated code, but it is difficult to satisfy the generation of complex syntax due to the complexity and richness of the browser's input space. In this work, DFL tackles the problem of high-quality sample generation by designing language-specific generators which combines the ideas of template-based and grammar-based generation.

Coverage-guided fuzzing framework. The idea is widely used in OS kernel security testing [30,32,55], and has been applied to fuzzing scenarios in a variety of scenarios with very successful results. Such as the TriforceAFL and kAFL frameworks, the addition of Intel PT trace coverage feedback guidance to kAFL resulted in a 40x improvement in test efficiency over TriforceAFL [32,56]. However, there are few demonstration cases of coverage-based bootstrapping to improve the efficiency of rendering engine fuzzing in browser security testing scenarios. The advanced FreeDom implements coverage-guided mutation, however coverage has a very weak impact on the improvement of bug-mining effectiveness, with the coverage of basic blocks of code increased by only 2.62% at most before and after the use of the coverage-guided technique [28,57]. In DFL, we use the idea of AFL instrumentation to insert coverage collection functions during browser compilation (see Fig. 2), providing the framework with feedback on the exploration path during sample execution to optimize the generation of new samples and improve fuzzing efficiency.

Fuzzing efficiency. Comparative analysis of application performance developed in popular programming languages shows that C++ achieves extremely high performance [58,59]. Relative to C++ programs, Python and JavaScript are 8.01 and 29.5 times slower, respectively. Most of the current browser fuzzing tools are written in Python [24,28,29], JavaScript [44] and other scripting languages, which lowers the fuzzing efficiency. For example, under the same hardware and software environment to generate 5000 HTML test samples multiple times (10 times), FreeDom consumes an average of 0.428 s per sample, while the generator developed in C++ only takes 0.289 s. Therefore, we use C++ to implement DFL to significantly improve the fuzzing efficiency.

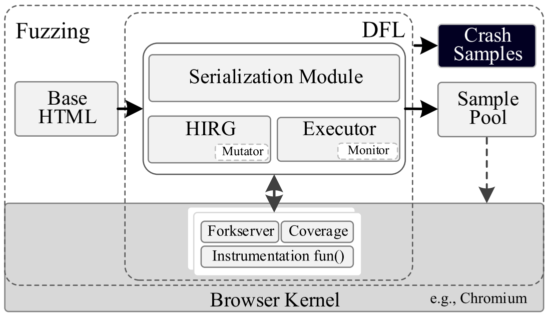


Fig. 2. DFL fuzzing framework functional module structure.

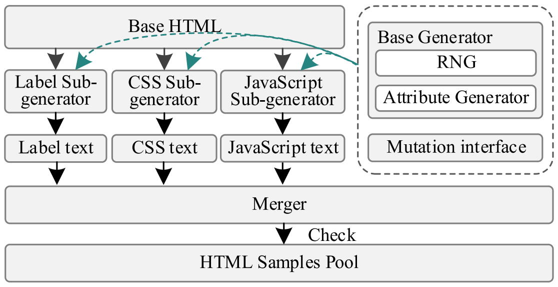


Fig. 3. HIRG sub-generator structure and samples merging process. Dash arrows indicate sub-generators calling base generators and mutation interfaces.

4. Design and implementation

This section comprehensively outlines the design and implementation of the DFL framework, encompassing four key components. Firstly, Section 4.1 provides an overview of the general framework of DFL. Secondly, Section 4.2 elaborates the design and implementation of multiple sub-generators. Thirdly, Section 4.3 details the design intricacies of serialization and deserialization algorithms. Lastly, Section 4.4 offers an in-depth exploration of the design of the executor, with detailed descriptions of the instrumentation, coverage collection, and memory monitoring.

4.1. Framework overview

The DFL consists of five parts: the executor, the HIR generator, the serialization module, the monitoring module, and the instrumentation module. The executor is the hub of all DFL modules and is responsible for connecting the functionalities of other components. The generator synthesizes HTML samples with different sub-generators. The serialization module is responsible for the conversion between the two states of the generated samples: objects in memory and files on disk. The monitoring module communicates with the instrumentation module to obtain the coverage information and the state of the target program. Finally, the instrumentation module collects interested information by instrumenting the browser. The overall architecture of the framework is shown in Fig. 2.

We generate HTML samples from a basic HTML file (with only HTML base tags) using the HIRG generator, which is repeatedly called during the generation process. The sample pool is empty at the beginning, and we gradually add legitimate HTML samples to the pool after blind testing. The executor reads the HTML samples and submits them to the fork server for execution. The monitoring module collects coverage information of the samples when they are executed. Samples are mutated to generate new samples based on coverage, and are stored by the serialization module when a crash is triggered. Compared with Freedom, which can use both generation mode and coverage-guided mode, the DFL framework first performs blind testing and then conducts subsequent tests based on legal samples combined with coverage. The legality of sample generation to reduce the number of triggered crashes.

4.2. HIR generator

The HTML sample input by the browser consists mainly of the initial DOM, CSS and JS. Therefore, the HIR generator in the DFL framework is designed with three sub-generators for Label, CSS, and JS, along with a basic generator. Label object nodes generated by the label sub-generator are added to the DOM tree, while the texts generated by the CSS and JS sub-generators are embedded in style and script tags respectively. The three sub-generators jointly maintain a syntax tree with the same context, and the individual sub-generators share with each other the constructed state of the current tree. Compared with FreeDom, different sub-generators in DFL generate text files, and then use sample construction modules according to DOM syntax standards to complete sample generation. In the randomization process, random numbers come from the basic generator, and different attribute data come from the attribute generator. The workflow of the HIRG generator is shown in Fig. 3.

4.2.1. Base generator

The base generator is responsible for generating random number and attributes that serve for the text generation of the label, CSS, and JS sub-generators.

Random number generator. In common programming languages, random functions (e.g., "rand" function in C) typically utilize mathematical calculations and a seed value to generate pseudo-random numbers. However, these functions have a limited period after which the sequence of generated numbers repeats, resulting in a lower degree of randomness in the obtained results. We augment the random functions in common programming languages by quantifying the operation noises (e.g., mouse, keyboard, and other device activities).

Attribute generator. The attribute object in an HTML text consists of a pair of `name` and `value`. The value may be a natural number, a string with a percent sign, or many other data types. We design attribute interfaces for each data type to realize the diversity of attributes. For each interface, we design the generation function for the corresponding attribute value separately, and then build a global mapping table for `name-value`.

4.2.2. Label sub-generator

The label sub-generator adds arbitrary label nodes to the DOM tree, which is the main component of HTML. Each label node contains a `tag` (label name), a number of attributes and child label nodes, creating a mapping table (e.g., `label_table`) between the three. We collect all the label names as the `Key` of the mapping table to provide candidate values for the tag variables (e.g., `name`). At the same time, we utilize the mapping table's `Key` to limit the range of values for the attributes of the label. In order to express complex label attribute mapping relationships, we design classes (e.g., `Element`) and add a parent pointer to point to the parent of the current node.

When generating `tag`, the integer value returned by RNG is used to select the candidate value from the label table. When generating attributes, the tag–attribute table is searched to obtain a list of attributes of the corresponding tag, and the attribute values are randomly selected. In addition, each node can randomly insert a child node to ensure its randomness. The key process of generating the text of a label node is shown in Algorithm 1.

Algorithm 1 Label text attribute generation method

```
Require: name and label_table
 1:                                        ▷ Label_table is relationship mapping table
 2: function Get_Str(name)
 3:     if name in label_table then
 4:         val ← Random(1,100000).to_string                ▷ Random number
 5:         val ← name + val
 6:         Str ← "<name id=val"
 7:         for attr in label_table do
 8:             Str ← Str + "  "                              ▷ Insert space
 9:             Str ← Str + attr.get_Value(name)
10:         if childens.size(name)==0 then
11:             Str ← Str + "/>"
12:         else
13:             Str ← Str + ">"
14:             for child in childens(name) do
15:                                                       ▷ Random selection
16:                 Str ← Str + child.get_Value(name)
17:                 Str ← Str + "\n"
18:             Str ← Str + "</name>"
19:         return Str                       ▷ Generate text for attribute name
20:     else
21:         exit()
```

4.2.3. CSS sub-generator

CSS syntax consists of a selector name and various attributes in a multinomial tree with a height of 1. There are multiple ways to represent a selector. The generation process uses a random functions to generate the name of a large number of CSS selector and attributes, which will become a node object in the DOM tree. When generating selector names, the CSS sub-generator limits rang the random selection of names to a list of objects (collecting objects that already exist in the DOM tree according to syntax rules). The parent class `CSS_Selector` is designed in the CSS child generator to manage all selector objects. Its subclass is designed to design constructors for all selector names and store the constructor names as a global table indexed by the selector name. The global table creates a mapping between the constructor and the attribute data, which has randomly selected values limited by the selector name. Finally, the member function instantiates the selector object to generate CSS text that conforms to CSS syntax.

4.2.4. JavaScript sub-generator

JS is an important scripting language in the construction of HTML pages, and is widely used for element control, interactive response, and data processing in the page. HTML syntax for many labeled objects, designed for a large number of functions to control the behavior of the label. To make the generated sample tests more comprehensive, we construct function mapping tables built from three elements: labels, functions and attributes. JS is used to encapsulate the labeled function node objects in the DOM tree into API call-like functions to extend the coverage of the generated samples. The structure of each line of JS API code is abstracted into three parts, which are used to receive the return value (the variable `Ret`), the API function name and the function parameters ("`obj.`" is the object prefix when called). The three parts are independently designed as subclasses, each of which is responsible for the generation and modification of the output values of the subclasses. Through the modification of the values of the incoming parameters, of the API can be tested to discover possible vulnerabilities. JS API code structure is shown in Table 3.

Table 3
JS API syntax structure.

Ret	Object	Api name	Args
var x=	obj.	func	arg1, arg2, arg3 ...

JS texts are generated based on objects that already exist in the current DOM tree. However, to ensure the relevance of the context objects, the objects that can be selected at the time of construction also include the Ret variable objects (return handlers) that are newly added during the JS API call. The number of API functions is obtained from the mapping table based on the object name and the weight. The weight is calculated based on the number of API functions and determines the selection probability.

4.2.5. Mutation interface

We propose a unified mutation interface (e.g., `mutation()`), and the three sub-generators mutate names and attribute values through this interface. The mutation function has different methods according to the characteristics of the sub-generator, using five operations: insertion, deletion, replacement, random merging of objects, and modification of object values to complete the mutation. The details of mutation of objects that each sub-generator operates on are slightly different. The CSS sub-generator completes the mutation operation by adding and subtracting selectors and attributes. Attribute objects for HTML (including SVG) nodes can be mutated by randomly selecting an attribute and a class of attributes in the node. The random merge operation is realized by merging all or some of the attributes of two node objects. When the sub-generator initiates a mutation operation, the mutation interface calls the deserialization module to convert the test data from the sample generator into state data in memory, and then serializes the samples to text or writes them to the DOM tree when the mutation is complete.

4.2.6. Sample structure

Three sub-generators were developed based on RNG. Different sub-generators are based on HTML syntax standards in order to accomplish the generation of corresponding texts, with no direct connection between the various types of texts. Each one HTML document consists of a DOM syntax tree, multiple CSS style, and multiple JS. We merge the text data generated by these sub-generators sequentially in the standard HTML document format so that they become input samples that can be recognized by the DFL fuzzing framework. When the sample is synthesized, the corresponding sub-generator is called according to the HTML grammar object (e.g., `Label` object calls label sub-generator), and the sub-generator then calls the mutation interface, which completes the mutation by modifying the information of the label nodes, CSS information, and the JS code structure, and returns the generated text. The text data generated by mutation ensures that synthetic HTML samples have diverse and potentially vulnerable properties that satisfy the input requirements of the fuzzing framework.

4.3. Serialization module

The serialization module contains two methods: serialization and deserialization. They enable the conversion between the text file and

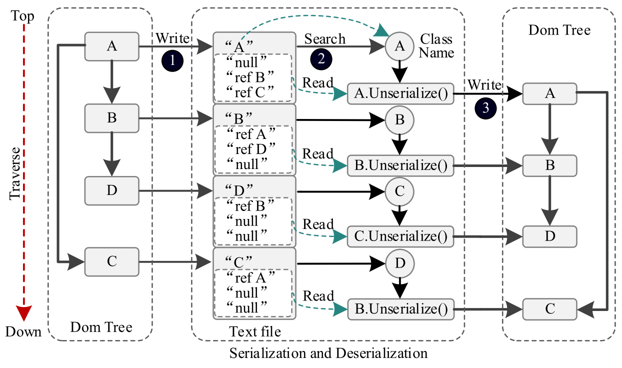

Top
Traverse
Down
Dom Tree
Write
Search
Class Name
Read
Write
Text file
Serialization and Deserialization
A.Unserialize()
B.Unserialize()
C.Unserialize()
B.Unserialize()
"A" "null" "ref B" "ref C"
"B" "ref A" "ref D" "null"
"D" "ref B" "null" "null"
"C" "ref A" "null" "null"

Fig. 4. DOM tree serialization and deserialization conversion methods. ❶ Write serialized memory data to hard disk, ❸ is the deserialization of text data in the hard disk and then written to memory. ❷ Deserialization algorithm, node index lookup.

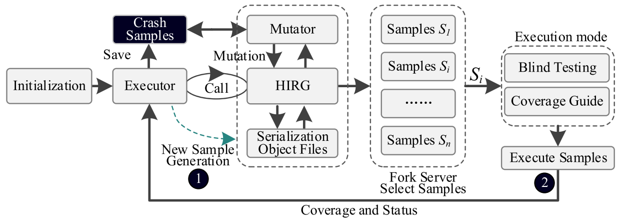


Fig. 5. Sample execution flow. Where ❶ is the generation of a new sample after selecting the Coverage Guide method. ❷ is the feedback information after executing the sample in the rendering engine.

in-memory data of the same DOM tree. During serialization, the module traverses the object tree in a top-down manner. It stores the data of the tree's child nodes and leaf nodes in a text file format, using the class name as an index. The class name serves as a guide for the deserialization algorithm. The deserialization algorithm uses the class name to determine the next object and its type to operate on. It sequentially reads the data from the serialized file stream and restores the objects accordingly. This process can be seen as a Deterministic Finite Automaton (DFA), where the state machine starts with an empty initial state and reads the data from the file stream in the order it was written. Each string is used to create the corresponding object. Fig. 4 shows the serialization transformation process.

In Fig. 4, the DOM tree and the serialized text file are stored in memory and on disk, respectively. When a mutation request is initiated by the sample generator, the serialization module performs deserialization on the text file and loads the resulting data into memory. Restores the DOM tree to a generator object operable by the mutation function. After the deserialization, the sample generator invokes the mutation interface `mutation()`, to perform mutations on the DOM syntax tree. For example, when the deserialization program reads the string "A", it calls the constructor of object A to create an instance of object A. It then invokes the object's functions to assign values to the member variables of the object. By integrating the serialization and deserialization processes with the mutation interface, we can effectively mutate the DOM syntax tree and generate new HTML samples.

4.4. Executor

The executor of the DFL framework continuously calls the HIR generator to make HTML samples and sends them to the fork server for execution. It also monitors the status of the browser and collects information about coverage. The workflow of the executor is depicted in Fig. 5.

There are two execution modes of the executor: blind testing and coverage-guided mutation, which run automatically in an alternating manner. By default, blind testing is performed which involves the generation, testing, and saving of random samples. If the coverage does not increase after a certain period of time (e.g., threshold of 6 min), it is switched to the coverage-guided mutation mode. In the coverage-guided mutation mode, the mutation process is triggered based on a crash sample. In case of a browser crash, the executor saves the current test sample S_i and initiates the sample mutation process. The mutation process generates a new sample S_i' from the crash sample S_i based on a genetic algorithm. The new sample S_i' is executed and monitored in a similar way. If there is an update of coverage, the new sample S_i' is retained, otherwise, it is discarded.

4.5. Monitoring and communications

The monitoring modules are responsible for communication with the *instrumentation fun()*. The instrumentation function is injected into the browser during compilation. Its objective is to facilitate the execution of test samples, monitor the kernel's state during execution, and collect coverage path information of the samples.

Fork server. We use the *Fork Server* from AFL [19], which excels in its communication functionality between the executor and the instrumentation program. The fork program and the coverage collection code are simultaneously instrumented into the basic block of the target program during compilation. A call instruction is inserted into each basic block for calling the business functions of the fork program. Before the call, the setting of a global variable is checked. If it is not set (indicating that the *Fork Server* is not running), the business function of the fork program is executed; otherwise, the current instrumentation check is skipped. During runtime, when the *Fork Server* receives control commands from the executor, it forks a new process using the fork instruction, creating a separate copy that shares memory with the original process. This allows for faster response times when there are no data write operations. This feature of using fork eliminates the need for dynamic loading and linking of the program, thus saving time in parsing and loading files.

Coverage collection. The coverage collection feature incorporates the coverage feedback code called instrumentation code block (*ICB*) from the AFL [19] framework. It achieves coverage collection by inserting *ICB* code into the intermediate code through a customized LLVM Pass module. The *ICB* code is placed at the beginning of each basic block. In our code, we define two identifiers: one represents the current basic block number (*cBn*), and the other represents the previous basic block number in the execution path (*preBn*). We insert a random number *cBn* as the first line of the *ICB*. Perform an XOR operation between *cBn* and *preBn* to obtain the index of the code block. Then, accumulate the coverage data at the corresponding position in shared memory. We create shared memory using *shmat()* (*SHM API* is used in Linux) and then the executor reads the shared memory data using *shmget()*, thereby retrieving the coverage feedback from the instrumentation functions.

Memory monitor. The purpose of memory monitoring is to detect vulnerabilities that may not trigger kernel crashes. These types of vulnerabilities are difficult to detect as they do not exhibit any obvious symptoms when triggered by a sample, such as small-scale array out-of-bounds accesses. To achieve this, we have added the AddressSanitizer (ASan) [60] tool during the instrumentation process. ASan is a fast memory error detection tool that consists of a compiler instrumentation module and a runtime library, which includes functions like *malloc()* and *free()*. This is a memory error detection plugin based on LLVM. It is instrumented into the program during compilation to check for memory vulnerabilities. It can detect memory vulnerabilities such as heap overflow, use-after-free, stack overflow, and memory leaks. When these issues occur in the program, ASan aborts the program's execution and prints information about the problematic location and the type of vulnerability. The program termination signal will be captured by the

Table 4
Experimental hardware and software environment.

Classification	Configuration	Parameters
Hardware	CPU	Intel(R) Xeon(R) CPU E7-4850 V2, 2.30 GHz, 4 Socket, Supports Intel VT-x
	Memory	ECC DDR3 1600 MHz, 32 GB, 8 Pieces
	Hard Drive	SATA3 2TB, 2 Pieces
Software	Operating system	Ubuntu 21.04 LTS Desktop-amd64, kernel version 5.15.0
	Browser kernel	Chromium 101.0.4951.8 (Developer build) (64-bit)
	Compilers	Clang 14.0.0
	Fuzzing framework	AFL v2.57b

monitoring program in the DFL framework, identified as a crash state, and submitted to the executor.

In the process execution monitoring module, we have designed a timer with the goal of setting a time limit for each sample test execution. If a process fails to produce a result within the specified time, it indicates that the current test may be running a sample with an infinite loop or other meaningless operations. To conserve system resources, the timed-out tested process is forcefully terminated.

5. Experiments

In this section, a detailed evaluation of our framework is provided. In Section 5.2, the speed and quality of HTML sample generation are evaluated. Additionally, in Section 5.3, the performance of our framework and other popular fuzzing frameworks are compared, including coverage-based metrics. Finally, in Section 5.4, the crash discovery capability of our framework and the discovered vulnerabilities are reported.

5.1. Experimental setting

Environment setup. DFL is implemented based on the AFL fuzzing framework, as shown in Fig. 2. The framework is developed and tested on Ubuntu 21.04 operating system with Clang 14 as the compilation environment. The target for testing was the 64-bit browser kernel of Google Chromium. The versions of the browser kernel and AFL framework used in the experiments were the latest ones available at the time of writing this document. The details of hardware and software configuration are shown in Table 4.

Compared frameworks. We select three generator or fuzzing framework (collectively referred to as frameworks) for comparison, which are DOMato [29], Wadi [44] and FreeDom [28]. However, these frameworks have robust sample generation capabilities but do not provide testing, data collection and automated testing capabilities. To address this, we embedded the sample generation modules of the three frameworks as different generators in our DFL framework to form three testers to ensure fair comparison.

Evaluation metrics. We use sample generation speed, sample validity, path coverage, and crash discovery capability (quality and quantity) as the evaluation metrics. Sample generation speed is a count of the time consumed by each sample generator to generate a given number of samples, and can effectively reflect the efficiency of the generator components in the framework. Sample validity refers to the number of valid samples that can be parsed by the kernel syntax and trigger execution along valid paths. It reflects the quality of the generated samples. Path coverage refers to the number of valid execution paths covering the target under test per unit of time. Higher path coverage indicates a more effective exploration of program behavior. Crash discovery capability refers to the number of crashes that triggered a crash of the target under test, with high-quality crashes indicating potential vulnerabilities in the program.

5.2. Performance of sample generation

5.2.1. Generation speed

The vulnerability discovering ability of the fuzzing framework relies on the sample generator's sample generation speed efficiency. To compare the sample generation speed, we compare the generators in three popular frameworks (Domato, Wadi, FreeDOM) and the HIRG in our DFL. We compare the time taken by each generator to generate a fixed number of samples. Ten batches of sample requests are submitted for each generator, with the number of samples in each batch being 10, 100, 200, 400, 500, 1000, 1500, 2000, 3000, and 5000 respectively. The time taken by each generator to complete each batch task is recorded. Each task is repeated for 10 times, and the average time is reported. The results of sample generation time are shown in Fig. 6.

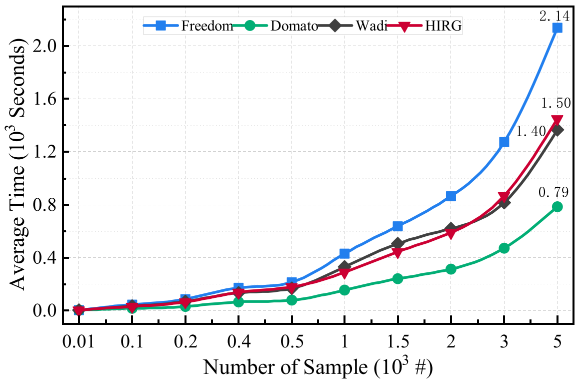


Fig. 6. Comparison of time-consuming generation of sample per unit number. The horizontal axis identifies the number of samples per unit, and the vertical axis refers to the length of time consumed by different generators to generate the number of samples per unit.

The data shown in Fig. 6 indicates that within the range of 100–500 sample generations, the four generators have similar time consumption. However, starting from generating 500 samples, there is a significant difference in the time taken as the sample quantity increases. Overall, FreeDom, developed using Python and with significant optimization potential in its internal algorithms, has the slowest sample generation speed. DOMato, on the other hand, is the fastest, but as a template-based sample generator that does not require complex computations, its implementation principle differs from the other three generators, making direct comparisons difficult. When comparing sample generators based on the same principle, HIRG and Wadi have comparable speeds. HIRG has optimized its algorithm and language, achieving a speed nearly twice as fast as FreeDom but slightly slower (7%) compared to DOMato.

5.2.2. Sample quality

Samples that can be parsed by the browser engine are considered valid samples. In fuzzing, it is expected to achieve a high rate of valid sample generation to enhance the testing efficiency of the target program. To evaluate the effectiveness of sample generation, we conducted a comparative experiment using four different fuzzer. The experiment involved grouping the samples based on the quantity: 10, 100, 200, 400, 500, and 1000. Each group of samples was inputted into the target browser, while monitoring the runtime status of the browser engine. The validity of the samples was determined by analyzing the output information for specific markers. In the monitoring module of

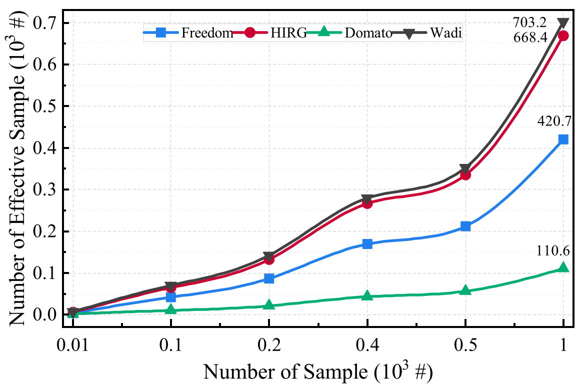


Fig. 7. Comparison of generator samples generation quality. The horizontal axis identifies the number of unit samples, and the vertical axis refers to the number of valid samples in the unit samples of different generators. The value marked is the maximum value of the valid sample.

the DFL framework, we introduced a feature field called "Console". If this feature field was found in the output log message, the current input sample was considered invalid. Conversely, if the feature field was not present, the sample was deemed valid. We repeat the experiment 10 times to take the average of the results, and the results are shown in Fig. 7.

Comparing these generators, it can be seen that Wadi and HIRG have a significant advantage in generating valid samples. The number of valid samples generated by HIRG was 1.67 times that of FreeDom, and its performance is nearly on par with Wadi (less than 5% difference). The lower validity of samples generated by DOMato can be attributed to its use of a deformity strategy, which disregards the contextual relationships within HTML. Wadi generates samples based on a generative technique after initializing the generator using syntax templates, which gives it a slight advantage over HIRG in terms of speed and quality of generation. However, the HIRG has a large advantage over Wadi in terms of the number of unique crashes found in the rendering engine, which is 200% higher than the Wadi generator (in Fig. 10).

5.3. *Coverage evaluation*

To evaluate the coverage collection performance of different frameworks, we compared the number of discovered paths by each framework within specified working duration. DFL-G/M represents the complete DFL testing functionality, while DFL-G signifies the mode without coverage-guided. Each framework was subjected to work for 1, 6, 12, 24, 36, 48, 60, and 72 h, respectively, while collecting the number of coverage paths during the runtime of the engine. The coverage comparison data is shown in Fig. 8.

In the experimental data, it was observed that DFL-G/M explored a large number of paths (524 286) in the kernel within just one hour and maintained a relatively stable trend in subsequent time intervals. FreeDom, DFL-G, and DOMato required 12 h to achieve the same coverage as DFL-G/M, while Wadi took 48 h. In terms of coverage metrics, DFL-G/M performed the best, Wadi showed moderate performance, and the other three frameworks performed similarly. The experiments provide strong evidence that the coverage guided technique and sample mutation strategy employed in our DFL can rapidly explore more paths and thus improve sample coverage.

5.4. *Crash discovery*

By designing a multitude of sub-generators to generate samples guided by the coverage information of execution path, the goal is to enhance the capability of the DFL framework to discover vulnerabilities of the rendering engine under test. We continue to compare the crash detection effectiveness among the five frameworks (5.3). Each framework operates continuously for 72 h, and we sample data at eight time points: 1, 6, 12, 24, 36, 48, 60, and 72 h. During the runtime, we collect the number of browser kernel crashes discovered. The number of detected crashes is shown in Fig. 9. It reveals that FreeDom triggers the highest number of crashes, followed by Wadi and DFL-G, while DOMato exhibits relatively poor performance. This experiment demonstrates that the samples generated by HIRG can induce crashes in the tested browser kernel, and the DFL framework successfully captures the crash states of the test targets.

While the fuzzing frameworks can capture crashes, the crash information alone does not necessarily represent the presence of vulnerabilities in the test targets. If the crashes triggered in the rendering engine share the same function or call stack, they would be treated as the same crash. Therefore, we further count the number of unique crashes that may be considered vulnerabilities by comparing the functions and call stack information of the crash logs. The results of the number of unique crashes along with the spent time is shown in Fig. 10. It can be seen that Wadi and DOMato have relatively lower crash discovery rates (one crash each), but Wadi had slightly higher efficiency in crash discovery compared to DOMato. FreeDom and DFL-G had an equal number of crash discoveries. DFL-G/M exhibited the fastest crash discovery rate, with two crashes discovered within two hours and three crashes discovered within four hours. The coverage collection methodology and added features used in the DFL framework excel in both efficiency and number of valid crash detections, further highlighting the superior testing capabilities of the DFL-G/M. The combination of coverage-guided sample generation, mutation, and timer mode switching methods partially addressed the issue of fuzzing frameworks being unable to automatically adjust detection paths, thereby enhancing the exploration capability.

DFL is built on FreeDom's design philosophy, and DFL-G/M can find one more bug in addition to all the unique Crash that FreeDom can find. The comparison between DFL-G/M and it proves that the added features can improve the quantity and quality of Crash discovery. Additionally, we submitted the newly discovered crash information to the official Google Chrome team for verification [61]. This further validates the effectiveness and advantages of DFL in browser vulnerability discovery.

6. Discussion

In this section, we discuss the limitations in the implementation of DFL and the future work that needs to be done.

6.1. *Limitations of implementation*

In terms of implementation, we delve into the potential limitations of the framework from three perspectives: design, validation, and usage.

Design limitations. The multiple sub-generators designed in the DFL framework are mainly oriented towards the generation of samples for most API calls in HTML, and do not target the JS interpreter's own syntax and API implementation for sample generation. The framework treats the browser as a large program to be tested, and compared to testing individual modules, the entire testing space is too vast, making it challenging to efficiently explore program paths. The sample generators designed in our framework are not yet well coupled with AFL, and some of the excellent features of AFL are not yet fully utilized.

Limitations of experiment. The results of multiple experiments (In Section 5) demonstrate the advantages of DFL. However, DFL is constructed based on the Blink kernel under the X86 architecture, and it has achieved good results in testing the DOM module of this kernel. However, it has not been fully constructed and tested on other

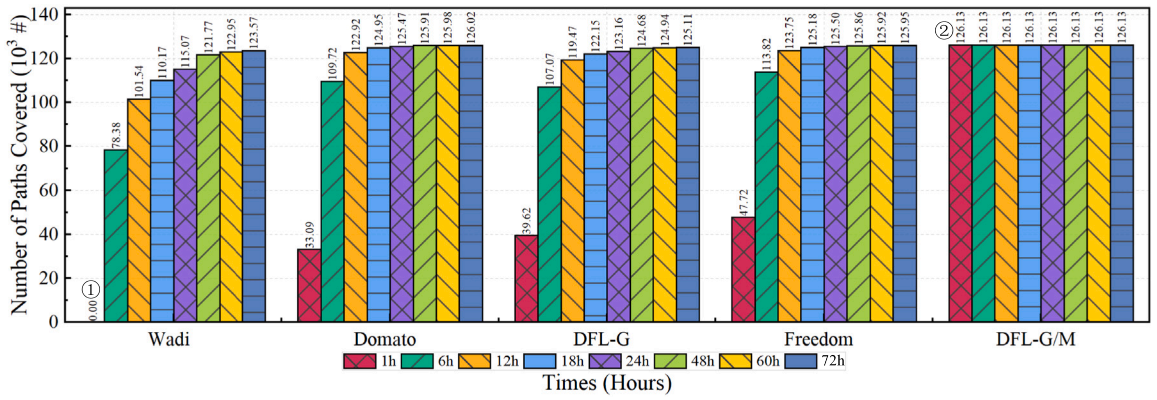


Fig. 8. Comparison of sample path coverage efficiency generated by sample generator. The figure is the coverage path collected by each framework minus the minimum value of the experimental results plotted. The point in the figure ① is the minimum value of the coverage path in this experimental result 398158. Position ② is the maximum value of the coverage path in the experiment is 524 286.

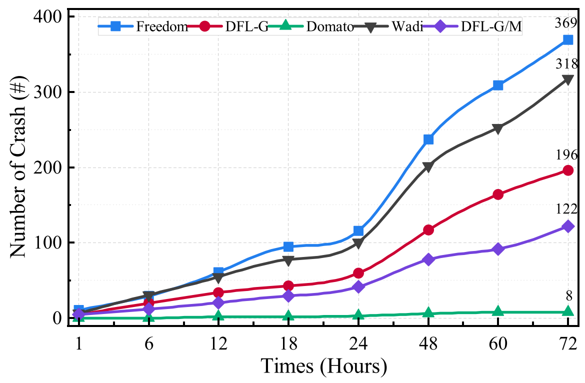


Fig. 9. Comparison of the number of crashes of the fuzzing framework at different sampling time points. The horizontal axis identifies the sampling time point, and the vertical axis indicates the number of crashes captured.

browser kernels and architecture environments (such as ARM). This limitation is acknowledged and is included in our plans for future improvement. Additionally, the instrumentation and compilation of browser programs result in increased file size and slower startup times, which can consume significant resources and lower testing efficiency.

Limitations of usage. The framework designed in the paper was developed using C++ in the Ubuntu environment (Table 4). It has limited consideration for the dependency on operating system and compiler versions. If there is a need to switch to other operating system environments such as Windows, Android, or iOS, please consider potential limitations in terms of compilation dependencies and runtime. Additionally, in the design of the sample generators, we have incorporated various syntax constraints to ensure that the generated samples adhere to the HTML grammar as much as possible. However, when conducting research using DFL, it is necessary to strike a balance between the threshold of sample effectiveness and ineffectiveness. By including samples with unexpected syntax formats to some extent and combining coverage-guided techniques, it is possible to increase coverage and explore more code branches.

6.2. Future work

We have designed numerous sample generators based on this syntax-constrained mechanism, and their effectiveness in fuzzing has been demonstrated through experiments. However, there is still a need for further research to be conducted in order to explore and advance this area. **High-quality fuzzing framework** remains our next goal. Currently, our fuzzing of rendering engine only focuses on the DOM objects, while the browser itself is a complex system software. There are other components in the rendering engine, such as `pdfium`, `omnibox`, and `mojo rpc`, which are also prone to frequent bug occurrences. Therefore, our goal is to analyze the characteristics of these modules and design a greater variety of targeted sample generators. This will further enhance test coverage and improve the ability to discover deep-level vulnerabilities. **Directed fuzzing** is an important branch in the field of fuzzing. The results shown in Figs. 9 and 10 demonstrate that many crash reports point to the same bug, indicating that the framework conducted a significant amount of repetitive testing on the paths covered by the same functional module. A well-designed path management strategy during the testing process is helpful for the testing engine to avoid exploring duplicate paths. Our further research goal is to develop new path strategies based on the testing objectives and achieve path-guided fuzzing. Furthermore, the process from vulnerability discovery to fixing often takes a considerable amount of time. During this period, users face unprecedented risks. Therefore, an interesting and worthwhile topic for further exploration is whether we can design protective frameworks based on path strategies to temporarily address (path avoidance) newly discovered vulnerabilities.

7. Related work

Code auditing and software testing are two approaches for discovering vulnerabilities in browser kernels. Manual code auditing is a time-consuming and labor-intensive process, often aided by tools like CodeQL [62] for code query assistance to reduce the workload of auditors. Software testing has evolved from black-box random testing to gray-box testing based on feedback from path coverage [63]. This approach increases the coverage of program paths by samples, reducing the randomness of test samples to some extent and improving the success rate of testing.

7.1. Code audit

There are certain differences in the approach to code auditing for open-source and closed-source systems. For open-source systems, security researchers can download the source code of the open-source browser kernel (e.g., Chromium, Webkit) and use IDE tools (e.g., VS Code) to read and examine the code. This method primarily relies on the experience of auditors to understand the code logic and manually discover vulnerabilities from the source code. In the case of closed-source browser kernels (e.g., Internet Explorer), security researchers use disassembly tools to disassemble the binary files and then search for calls to sensitive functions based on their experience. They analyze pseudocode and perform dynamic debugging using a debugger to verify the presence of vulnerabilities. This vulnerability discovery method has

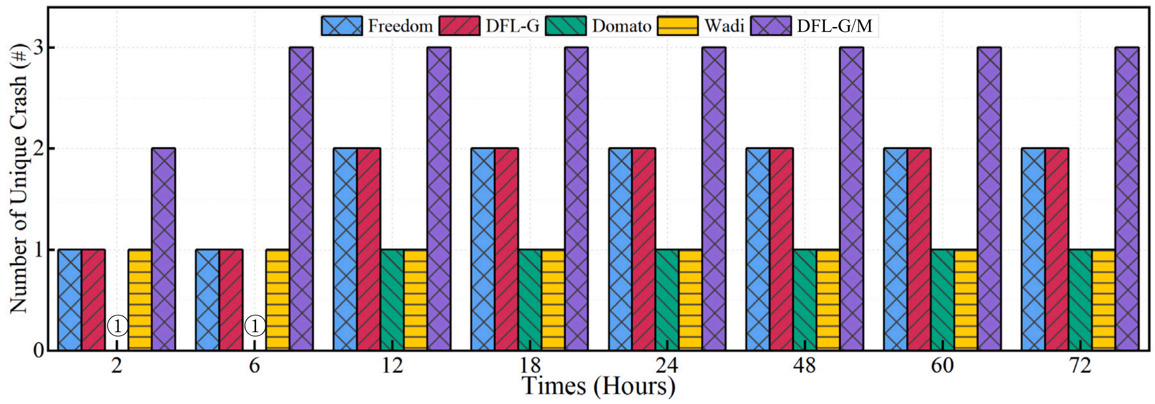


Fig. 10. Comparison of the number of unique crash messages found by fuzzing frameworks. The horizontal axis refers to the continuous time distribution, and the vertical axis identifies the number of crash messages. The point in the figure ① is the number of unique crashes discovered by the Domato framework at the 2nd and 6th hours of sampling, which the number is 0.

the advantage of avoiding false positives, but it requires a significant amount of manual effort, resulting in limited findings and slow efficiency. With the development of code auditing techniques, automated auditing tools have emerged, which can save security researchers a considerable amount of time.

CodeQL [62] is an excellent automated code auditing tool released by GitHub. It scans the target codebase, identifies vulnerabilities, and provides improvement suggestions. It treats code as a database and retrieves specific code structures using a query language similar to Structured Query Language (SQL). Researchers can leverage their vulnerability discovery experience to construct specific code structures as inputs for CodeQL, greatly enhancing the efficiency and convenience of auditing. Researchers from 360 Companys Chromium security team presented a method at Black Hat 2021 [64] for constructing samples. By leveraging the characteristics of existing vulnerabilities, they construct high-quality input samples to discover more specific types of vulnerabilities. In recent years, researchers have proposed various excellent code auditing tools such as Sys [7] and VCCFinder [8]. These methods have the advantage of being able to identify more specific types of vulnerabilities with low false positives. However, they require a high level of expertise from researchers to construct efficient CodeQL query statements.

7.2. Browser fuzzing

Currently, fuzzing methods for browsers primarily focus on the JS engine [9–15], with fewer approaches targeting the rendering engine. For fuzzing of rendering engines we discuss two approaches: sample generator-based fuzzing and coverage-guided sample mutation.

Sample generator-based. This approach relies on a variety of targeted generators to produce high-quality test samples, such as DOMato [29], Dharma [65], and Wadi [44]. DOMato, developed by Google, is a DOM sample generator that uses an intermediate description language to generate multiple HTML samples based on templates. This tool has discovered over 30 browser vulnerabilities. However, its drawback is that it only provides sample generation. The generated samples need to be fed into the target program using other tools. For requirements that involve triggering deeper code paths, researchers' expertise is necessary.

Sample mutation methods with coverage feedback. These methods based on coverage feedback guide sample mutation to discover new code paths and improve testing efficiency, such as FreeDom [28] and RIFF [66]. FreeDom adopts a sample generation and mutation method based on DOM syntax trees, and proposes a coverage-guided fuzzing scheme. Sample mutation is achieved by randomly performing operations such as deletion, insertion, attribute mutation, or node merging in the syntax tree. However, in the open-source code, only the sample generation part is implemented, and the parsing of HTML samples into the internal representation of the DOM syntax tree is not implemented. Execution and monitoring functionalities are also not provided, making it necessary to integrate with other fuzzing frameworks to perform fuzzing.

In addition, there are also semantic and API-based approaches to browser fuzzing. Such as the Favocado [10] and SaGe [38] are semantics-aware browser fuzzing solutions that focus on improving the correctness of the their generate inputs. Favocado extracts semantic information from browser source code and saves it in JSON structure. SaGe extracts semantics from W3C standards and save it in CFG. In comparison, JSON has a lower utilization rate. The CFG approach facilitates the generation of structured inputs, and to some extent solves the problem of low browser coverage due to handwritten CFG-generated inputs. Minerva [26] improves coverage with a approach that analyses API interference graphs and uses API interference relations to reduce the search space, and has been successful in browser API fuzzing, mainly for browser memory error discovery scenarios. In comparison, these fuzzing approaches are more focused on JS engines, while DFL specializes in sample generator's and framework design, and focuses on fuzzing for rendering engines.

8. Conclusion

In this article, we have developed a rendering engine fuzzing framework called DFL using the efficient programming language C++. The framework uses coverage-guided techniques to guide sample generation and mutation, which significantly improves test performance. DFL is designed with a modular approach, redesigning the generator based on FreeDom and introducing some of the best features from AFL to automate browser fuzzing. Additionally, DFL introduces HIRG generator and sample serialization/deserialization method to overcome the issue of AFL's inability to generate samples that conform to HTML syntax rules. Experimental results demonstrate that DFL outperforms other existing rendering engine fuzzing frameworks in terms of sample generation speed, coverage collection efficiency, and crash discovery capabilities. Moreover, compared to popular rendering engine fuzzing frameworks, DFL discovered an additional bug within a specified timeframe. This bug was certified by the official Google Chrome team [61], confirming that the triggered bug from our submitted sample is a genuine vulnerability. Nevertheless, we expect more researchers to conduct further studies based on DFL. Our goal is to address its limitations and enhance testing coverage and the ability to discover deep-level vulnerabilities. Specifically, in terms of devising coverage path management strategies, we plan to focus on designing research methodologies for directed fuzzing and intensify research efforts in mock patch fixes targeting known vulnerabilities.

CRediT authorship contribution statement

Guoyun Duan: Writing – review & editing, Writing – original draft, Validation, Resources, Project administration, Methodology, Funding acquisition, Formal analysis. **Hai Zhao:** Writing – original draft, Validation, Software, Methodology, Investigation, Data curation. **Minjie Cai:** Writing – review & editing, Funding acquisition. **Jianhua Sun:** Writing – review & editing, Supervision, Resources, Methodology. **Hao Chen:** Writing – review & editing, Supervision, Resources, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We would like to express our gratitude to the anonymous reviewers for their thoughtful review and to our shepherd.

References

[1] A. Barth, C. Jackson, C. Reis, T. Team, et al., The Security Architecture of the Chromium Browser, Stanford University, 2008.
[2] Google, Chromium issues, 2023, https://bugs.chromium.org/p/chromium/issues/list?q=&can=1, Website.
[3] P. Snyder, C. Taylor, C. Kanich, Most websites don't need to vibrate: A cost-benefit approach to improving browser security, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 179–194.
[4] S. Oesch, S. Ruoti, That was then, this is now: A security evaluation of password generation, storage, and autofill in Browser-Based password managers, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 2165–2182.
[5] I. 1132218, Tesla.com: Color options not rendered until window resize when compositesvg is enabled, 2024, https://bugs.chromium.org/p/chromium/issues/detail?id=1132218, Website.
[6] C. Qian, H. Koo, C. Oh, T. Kim, W. Lee, Slimium: Debloating the chromium browser with feature subsetting, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 461–476.
[7] F. Brown, D. Stefan, D. Engler, Sys: A Static/Symbolic tool for finding good bugs in good (browser) code, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 199–216.
[8] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 426–437.
[9] C. Holler, K. Herzig, A. Zeller, Fuzzing with code fragments, in: 21st USENIX Security Symposium (USENIX Security 12), USENIX Association, 2012, pp. 445–458.
[10] S.T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, et al., Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases, in: 28th Annual Network and Distributed System Security Symposium, NDSS, February 21-25, 2021, The Internet Society, 2021.
[11] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, T. Holz, JIT-picking: Differential fuzzing of JavaScript engines, in: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 351–364.
[12] S. Lee, H. Han, S.K. Cha, S. Son, Montage: A neural network language model-guided JavaScript engine fuzzer, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2613–2630.
[13] G. Ye, Z. Tang, S.H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, Z. Wang, Automated conformance testing for JavaScript engines via deep compiler fuzzing, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, in: PLDI 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 435–450.
[14] H. Han, D. Oh, S.K. Cha, CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines, in: 26th Annual Network and Distributed System Security Symposium, NDSS, The Internet Society, 2019.
[15] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, M. Egele, HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing, 2020, arXiv preprint arXiv:2002.03416.
[16] J. Wang, B. Chen, L. Wei, Y. Liu, Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy, SP, 2017, pp. 579–594.
[17] S. Groß, S. Koch, L. Bernhard, T. Holz, M. Johns, FUZZILLI: Fuzzing for JavaScript JIT compiler vulnerabilities, in: Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, 2023.
[18] S. Wi, T.T. Nguyen, J. Kim, B. Stock, S. Son, DiffCSP: Finding browser bugs in content security policy enforcement through differential testing, in: 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023, The Internet Society, 2023.
[19] M. Zalewski, AFL, 2023, https://lcamtuf.coredump.cx/afl/, Website.
[20] S. Karamcheti, G. Mann, D. Rosenberg, Adaptive grey-box fuzz-testing with thompson sampling, in: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, AISec '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 37–47.
[21] A. Oest, Y. Safaei, A. Doupé, G.-J. Ahn, B. Wardman, K. Tyers, PhishFarm: A scalable framework for measuring the effectiveness of evasion techniques against browser phishing blacklists, in: 2019 IEEE Symposium on Security and Privacy, SP, 2019, pp. 1344–1361.
[22] U. Iqbal, S. Englehardt, Z. Shafiq, Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors, in: 2021 IEEE Symposium on Security and Privacy, SP, IEEE, 2021, pp. 1143–1161.
[23] S. Kim, Y.M. Kim, J. Hur, S. Song, G. Lee, B. Lee, $\{FuzzOrigin\}$: Detecting $\{UXSS\}$ vulnerabilities in browsers through origin fuzzing, in: 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1008–1023.
[24] G. Kwong, DOMFuzz, 2023, https://github.com/MozillaSecurity/domfuzz, Website.
[25] C. Shou, u.B. Kadron, Q. Su, T. Bultan, CorbFuzz: Checking browser security policies with fuzzing, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), ASE '21, IEEE Press, 2022, pp. 215–226.
[26] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, Y. Jiang, 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, in: ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1135–1147.
[27] R. Wang, G. Xu, X. Zeng, X. Li, Z. Feng, TT-XSS: A novel taint tracking based dynamic detection framework for DOM cross-site scripting, J. Parallel Distrib. Comput. 118 (2018) 100–106.
[28] W. Xu, S. Park, T. Kim, FREEDOM: Engineering a state-of-the-art DOM fuzzer, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 971–986.
[29] I. Fratric, A DOM fuzzer, 2023, https://github.com/googleprojectzero/domato, Website.
[30] G. Duan, Y. Fu, B. Zhang, P. Deng, J. Sun, H. Chen, Z. Chen, TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation, Future Gener. Comput. Syst. 144 (2023) 192–204.
[31] P. Chen, H. Chen, Angora: Efficient fuzzing by principled search, in: 2018 IEEE Symposium on Security and Privacy, SP, 2018, pp. 711–725.
[32] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, T. Holz, kAFL: Hardware-assisted feedback fuzzing for OS kernels, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 167–182.
[33] M. Cho, D. An, H. Jin, T. Kwon, BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing, in: 32nd USENIX Security Symposium (USENIX Security 23), USENIX Association, Anaheim, CA, 2023, pp. 4985–5002.
[34] S. Khodayari, G. Pellegrino, It's (DOM) clobbering time: Attack techniques, prevalence, and defenses, in: 2023 IEEE Symposium on Security and Privacy, SP, 2023, pp. 1041–1058.
[35] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, J. Schwenk, Scriptless attacks: Stealing the pie without touching the sill, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 760–771.
[36] S. Ninawe, R. Wajgi, Detection of DOM-based XSS attack on web application, in: S. Balaji, A. Rocha, Y.-N. Chung (Eds.), Intelligent Communication Technologies and Virtual Mobile Networks, Springer International Publishing, Cham, 2020, pp. 633–641.
[37] D.T. Noß, L. Knittel, C. Mainka, M. Niemietz, J. Schwenk, Finding all cross-site needles in the DOM stack: A comprehensive methodology for the automatic XS-leak detection in web browsers, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 2456–2470.
[38] C. Zhou, Q. Zhang, L. Guo, M. Wang, Y. Jiang, Q. Liao, Z. Wu, S. Li, B. Gu, Towards better semantics exploration for browser fuzzing, Proc. ACM Program. Lang. 7 (OOPSLA2) (2023).

[39] H.L. Nguyen, L. Grunske, BEDIVFUZZ: Integrating behavioral diversity into generator-based fuzzing, in: 2022 IEEE/ACM 44th International Conference on Software Engineering, ICSE, 2022, pp. 249–261.

[40] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, L. Zhang, 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, in: ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 530–543.

[41] R. Valotta, Taking browsers fuzzing to the next (DOM) level, 2023, https://deepsec.net/docs/Slides/2012/DeepSec_2012_Rosario_Valotta_\protect\discretionary{\char\hyphenchar\font}{}{}_Taking_Browsers_Fuzzing_to_the_next_(DOM)_Level.pdf, Website.

[42] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, W. Zou, MAZE: Towards automated heap feng shui, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, 2021, pp. 1647–1664.

[43] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang, P. Su, HTFuzz: Heap operation sequence sensitive fuzzing, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22, Association for Computing Machinery, New York, NY, USA, 2023.

[44] Wadi, 2023, https://github.com/sensepost/wadi, Website.

[45] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, M.F. Kaashoek, Linux kernel vulnerabilities: State-of-the-art defenses and open problems, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys '11, Association for Computing Machinery, New York, NY, USA, 2011.

[46] F. Nabi, J. Yong, X. Tao, A taxonomy of logic attack vulnerabilities in component-based e-commerce system, Int. J. Inf. Secur. Res. 9 (2019) 898–905.

[47] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, DeepWukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Trans. Softw. Eng. Methodol. 30 (3) (2021).

[48] P.H.N. Rajput, C. Doumanidis, M. Maniatakos, ICSPatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs, in: Proc. 32nd USENIX Secur. Symp, USENIX Association, 2023.

[49] Syzkaller, 2023, https://github.com/google/syzkaller, Website.

[50] V.J. Manès, H. Han, C. Han, S.K. Cha, M. Egele, E.J. Schwartz, M. Woo, The art, science, and engineering of fuzzing: A survey, IEEE Trans. Softw. Eng. 47 (11) (2021) 2312–2331.

[51] P. Godefroid, M.Y. Levin, D. Molnar, SAGE: Whitebox fuzzing for security testing, Commun. ACM 55 (3) (2012) 40–44.

[52] C. Beaman, M. Redbourne, J.D. Mummery, S. Hakak, Fuzzing vulnerability discovery techniques: Survey, challenges and future directions, Comput. Secur. 120 (2022) 102813.

[53] P. Godefroid, H. Peleg, R. Singh, Learn&Fuzz: Machine learning for input fuzzing, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, 2017, pp. 50–59.

[54] H. Xu, Y. Wang, Z. Jiang, S. Fan, S. Fu, P. Xie, Fuzzing JavaScript engines with a syntax-aware neural program model, Comput. Secur. 144 (2024) 103947.

[55] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, K. Li, GREBE: Unveiling exploitation potential for linux kernel bugs, in: 2022 IEEE Symposium on Security and Privacy, SP, 2022, pp. 2078–2095.

[56] J. Li, B. Zhao, C. Zhang, Fuzzing: a survey, Cybersecurity 1 (2018) 1–13.

[57] W. Xu, S. Park, T. Kim, FreeDom source code, 2024, https://github.com/sslab-gatech/freedom, Website.

[58] D. Lion, A. Chiu, M. Stumm, D. Yuan, Investigating managed language runtime performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and go can be faster? in: 2022 USENIX Annual Technical Conference (USENIX ATC 22), USENIX Association, Carlsbad, CA, 2022, pp. 835–852.

[59] P. Diehl, M. Morris, S.R. Brandt, N. Gupta, H. Kaiser, Benchmarking the parallel 1D heat equation solver in Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, and Java, in: Euro-Par 2023: Parallel Processing Workshops, Springer Nature Switzerland, Cham, 2024, pp. 127–138.

[60] Google, AddressSanitizer, 2023, https://github.com/google/sanitizers/wiki/AddressSanitizer, Website.

[61] Haivk007, Security: Heap buffer overflow in mojo message, 2023, https://bugs.chromium.org/p/chromium/issues/detail?id=1321040, Website.

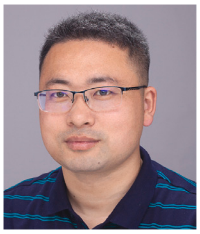
[62] GitHub, CodeQL, 2023, https://codeql.github.com/, Website.

[63] J. Wang, B. Chen, L. Wei, Y. Liu, Superion: Grammar-aware greybox fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, 2019, pp. 724–735.

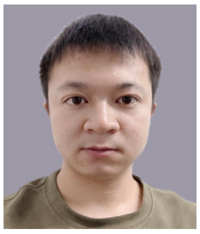
[64] G.G. Rong Jian, Put in one bug and pop out more: An effective way of bug hunting in chrome, 2023, https://www.classcentral.com/course/youtube-put-in-one-bug-and-pop-out-more-an-effective-way-of-bug-hunting-in-chrome-184998, Website.

[65] J. Schwartzentruber, Dharma, 2023, https://github.com/posidron/dharma, Website.

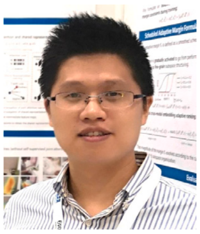
[66] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, J. Sun, RIFF: Reduced instruction footprint for Coverage-Guided fuzzing, in: 2021 USENIX Annual Technical Conference (USENIX ATC 21), USENIX Association, 2021, pp. 147–159.



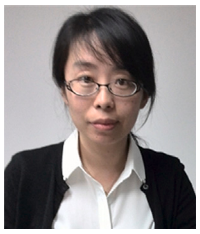
Guoyun Duan (Member, IEEE) is a Ph.D. student at the College of Computer Science and Electronic Engineering, Hunan University, China. He is now an Associate Professor at the School of Information Engineering, Hunan University of Science and Engineering, China. His research interests are systems security, heterogeneous systems, and privacy security. He is a member of the IEEE and CCF.



Hai Zhao received the B.E. degree in Computer Science from Hunan University of Science and Engineering, China. Binary security researcher at Nanjing Cyberpeace Tech Co., Ltd. His research interests include system security, binary vulnerability discovery, and IoT hardware security, and has given a speech at the Black Hat conference.



Minjie Cai received the B.S. and M.S. degrees in electronics and information engineering from Northwestern Polytechnical University, China, in 2008 and 2011, respectively, and the Ph.D. degree in information science and technology from The University of Tokyo, in 2016. He is currently an Associate Professor with the College of Computer Science and Electronic Engineering, Hunan University, China. He also serves as a cooperative research fellow at The University of Tokyo. His research interests include computer vision, multimedia analysis, and human–computer interaction.



Jianhua Sun is a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. She received the Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China in 2005. Her research interests are in systems security, parallel and distributed computing, and operating systems. She has published more than 70 papers in prestigious journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, FGCS, VLDB, PACT, ICPP, and IPDPS.

Hao Chen (Member, IEEE, and ACM) received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the Ph.D. degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now a professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He has publication in prestigious international journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, Future Generation Computer Systems, ASPLOS, SIGMOD, VLDB, USENIX ATC, PACT, ICS, IPDPS and ICPP. He is a member of the IEEE and the ACM.