AN IR-BASED FUZZING APPROACH FOR FINDING CONTEXT-AWARE BUGS IN API-BASED SYSTEMS

A Dissertation
Presented to
The Academic Faculty

By

Wen Xu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computing
Department of Computer Science

Georgia Institute of Technology

May 2021

AN IR-BASED FUZZING APPROACH FOR FINDING CONTEXT-AWARE BUGS IN API-BASED SYSTEMS

Thesis committee:

Dr. Taesoo Kim (Advisor)
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Wenke Lee
School of Computer Science
Georgia Institute of Technology

Dr. Weidong Cui
Microsoft Research Lab - Redmond
Microsoft

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date approved: April 27, 2021

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Dr. Taesoo Kim. He is, indeed, a smart and hardworking person who has strongly motivated, guided, and supported me through my entire graduate study. I would also like to acknowledge my thesis committee, Dr. Wenke Lee, Dr. Alessandro Orso, Dr. Qirun Zhang and Dr. Weidong Cui, for their invaluable comments and suggestions on my dissertation.

I feel fortunate to work with many colleagues in the SSLab, including Dr. Sanidhya Kashyap, Dr. Changwoo Min, Dr. Sanghoo Lee, Dr. Byoungyoung Lee, Dr. Insu Yun, Dr. Hyungon Moon, Dr. Daehee Jang, Dr. Hong Hu, Dr. Meng Xu, Soyeon Park, Seulbae Kim, Ren Ding, Po-Ning Tseng, Fan Sang, and Jungyeon Yoon.

Last but not least, I would like to thank my parents, Dongnian Xu and Yiping Xiao, for their endless love and support during these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

API-based systems, a large group of security-critical software programs including web browser and OS kernels, accept program inputs being composed of API calls. Considering the scale and complexity of an API-based system, fuzzing proves to be the most effective approach for bug detection in practice. To effectively discover new bugs in an API-based system nowadays, a fuzzer needs to generate syntactically and semantically correct API calls, which are not declined at an early stage.

Grammar-based API fuzzers generate random API calls in various syntaxes described by context-free grammars. Nevertheless, context-free grammars are unable to deliver certain API semantics in an API program, especially how an API call interacts with the objects in the program. Therefore, the random API calls generated by such fuzzers largely have reference errors, type errors or state errors.

To effectively fuzz an API-based system, we present a context-aware fuzzing approach, which relies on RPG IR to generate random API calls. RPG IR is a formal and contextual representation that defines an object-based context for an API program and models not only the syntax but also the context-based semantics of every API call in the program. Hence, the generated API calls in RPG IR have reduced semantic errors and are more likely to trigger bugs in an API-based system. To evaluate the effectiveness of RPG IR in API fuzzing, we present FreeDom and Janus, two IR-based context-aware fuzzers targeting web browsers and file systems, respectively. In particular, FreeDom has revealed 24 previously unknown bugs in Apple Safari, Mozilla Firefox, and Google Chrome, 10 of which are assigned with CVEs. Meanwhile, FreeDom largely outperforms the grammar-based DOM fuzzer, Domato, with 3$\times$ more unique crashes. On the other hand, Janus visits at most 4.19$\times$ more code paths compared to the state-of-the-art system call fuzzer, Syzkaller, by generating context-aware file operations. More importantly, Janus has found 90 bugs in eight Linux file systems with 32 CVEs assigned.

We further present RPG (Random Program Generator), a more generic approach to conduct context-aware API fuzzing via RPG IR against different API-based systems. In particular, RPG accepts API description in ASL (API Specification Language), a formal language for developers to describe APIs that can be modeled by RPG IR. RPG manages to compile ASL files into a context-aware API fuzzer based on RPG IR specifically targeting the described APIs. We implement a prototype of RPG, which is evaluated by fuzzing WebKit with the ASL files that describe DOM and SVG specifications. As a domain-agnostic approach, RPG manages to discover a similar number of code blocks and unique crashes compared to FreeDom.

CHAPTER 1
INTRODUCTION

API (Application Programming Interface) based systems are a large group of security-critical software programs being accessed through a set of API calls, including but not limited to, OS kernels, web browsers, hypervisors, and databases. Common API-based systems expose numerous APIs via an implementation of extraordinary complexity, which creates significant difficulties for manual code inspection. Therefore, testing an API-based system automatically with random API calls, namely *fuzzing*, becomes the dominant approach for finding bugs in the system in practice [1, 2, 3, 4, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

An API-based system provides detailed API specifications for developers to make valid API calls. Meanwhile, an API call that does not meet the specification is considered syntactically or semantically incorrect. Such an API call is mostly declined by the system at an early stage where syntax parsing or semantic checking occurs and takes no effect. Since major API-based systems have been inspected for a long time, researchers have been discovering more contextual bugs that are located at deep code paths and triggered by a sequence of interdependent API calls without any syntax or semantic error.

A collection of recent fuzzers [1, 2, 4] invents various context-free grammars to describe the API format that meets the specifications. The expressiveness of a context-free grammar enables a fuzzer to generate API calls with correct syntax, namely the explicit structure of an API. Nevertheless, such a grammar proves unable to comprehensively deliver the semantics of an API, namely the implicit dependencies and effects of an API. The generated API calls thus naturally contain semantic issues and hardly trigger deep bugs in a target system.

We observe that a large portion of semantic issues are related to incorrect object operations performed by a randomly generated API call. More specifically, the API call may use

an invalid object, access an object of an unwanted type, or operate an object that has wrong status. To avoid those cases, we need to maintain a program context that records available objects and their states, which is queried for generating a new API call and updated based on the corresponding object state changes introduced by the generated API call. We call such an approach that relies on additional context information to reduce semantic errors during random API generation as context-aware API fuzzing.

To realize context-aware API fuzzing, we first propose RPG IR in this thesis. RPG IR is a formal and contextual abstraction of an API program. An API program in RPG IR contains an object-oriented context that records the lifetime, type and state information of every object created in the program. Furthermore, RPG IR relies on a context-based model to systematically describe the syntactic structure, object dependence and object state effects of every API call in the program. We then present context-aware API fuzzing based on RPG IR, including random generation of an API program in RPG IR with a consistent context and random mutation over an API program in RPG IR without corrupting the existing context. To evaluate the effectiveness of API fuzzing driven by RPG IR, we further design and implement two context-aware API fuzzers based on the API model described by RPG IR: FreeDom and Janus. FreeDom is a DOM fuzzer that uses RPG IR to describe the structure and context dependence of three distinct parts in an HTML document. Compared to the state-of-the-art DOM fuzzer, Domato, that is based on a context-free grammar, FreeDom discovers 3$\times$ more unique crashes in WebKit thanks to its context-awareness enabled by RPG IR. More importantly, FreeDom has successfully found 24 bugs in three mainstream browsers (*i.e.*, Apple Safari, Mozilla Firefox, and Google Chrome), to which 10 CVEs have been awarded. On the other hand, we also present Janus, a context-aware file system fuzzer that applies RPG IR to model Linux file operations and their effects to the file objects in an image. Compared to the grammar-based state-of-the-art system call fuzzer, Syzkaller, Janus achieves at most 4.19$\times$ more code coverage when fuzzing eight popular file systems for 12 hours. Moreover, Janus has found 90 bugs in the upstream Linux kernel, 32 of which have

been assigned with CVEs. The two applications based on RPG IR verifies the importance of context awareness in API fuzzing for reducing semantic errors.

To enlarge the research impact of RPG IR, we further present RPG (Random Program Generator), which leverages RPG IR for context-aware API fuzzing in a more generic manner. RPG works with ASL, a standard language for developers to program API syntax and semantics in the form RPG IR suggests. Given the ASL files that describe the concerned APIs, RPG automatically compile them into a context-aware fuzzer that generates random API calls in RPG IR. The generated API calls are free from the semantic issues related to object states previously mentioned and can be converted into any preferred format for testing. We implement a prototype of RPG, that contains the ASL description for DOM and SVG APIs. Our evaluation shows that when fuzzing either HTML or SVG documents in WebKit, RPG is always as effective as FreeDom by discovering a similar number of unique crashes and contextual crashes during a 24-hour run. Meanwhile, context awareness enables RPG to largely outperform Domato and Dharma two typical context-free grammar-based fuzzer.

In summary, this thesis makes the following contributions:

- We propose context-aware API fuzzing, where API programs are randomly generated in RPG IR. To reduce semantic errors caused by abusing the objects regardless of their states, RPG IR maintains an object-based context for an API program and explicitly describes the context-dependence of every API call in the program.
- We present two context-aware fuzzers, FreeDom and Janus, for finding bugs in web browsers and file systems, respectively. Both fuzzers model their concerned APIs as RPG IR and reflect the effectiveness of IR-driven context-aware fuzzing by discovering more than a hundred previously unknown bugs in real-world software with 42 CVEs assigned in total.
- We further present RPG, a generic API fuzzing approach that allows developers to program a context-aware fuzzer based on RPG IR by simply specifying APIs in ASL.

ASL is a formal language for describing API syntax and semantics based on the API model proposed by RPG IR.

The rest of the thesis is organized as follows. §2 presents an overview of context-aware API fuzzing. §3 proposes RPG IR in detail. §4 describes the details of context-aware API fuzzing via RPG IR. §5 presents FreeDom, a context-aware DOM fuzzer. The design and implementation of Janus, a context-aware file system fuzzer, is presented in §6. §7 proposes RPG and ASL. §8 discusses the limitations and future directions of context-aware fuzzing based on RPG IR. §9 introduces related work. Finally, §10 concludes the thesis.

CHAPTER 2

OVERVIEW

This chapter presents the research background, general ideas, and challenges of context-aware API fuzzing.

2.1 Background

In recent years, many API fuzzers have been proposed and show great success in discovering security bugs in real-world API-based systems represented by web browsers and OS kernels. We summarize the known API fuzzers in Table 2.1.

Fuzzer	Target	Year	Grammar	Context-awareness
Dharma [2]	Web browser	2015	✓	
Avalanche [4]		2016	✓	
Domato [1]		2017	✓	
FreeDom [6]		2020		✓
Syzkaller [7]	OS kernel	2015	✓	△
Janus [9]		2019		✓
Hydra [13]		2019		✓

Table 2.1: A list of recent API fuzzers. The **Grammar** column indicates whether the fuzzer is based on grammars. The **Context-awareness** column indicates whether the fuzzer is aware of the context when generating random API calls. Syzkaller is not completely context-aware and still suffers from semantic errors. FreeDom and Janus are two context-aware API fuzzer presented in this thesis.

A majority of the existing API fuzzers, such as Dharma [2], Avalanche [4], and Domato [1], describe APIs in context-free grammars. Such a grammar normally splits an API into multiple syntactic units and defines random values for the non-constant units through grammar rules. Figure 2.1, Figure 2.2, and Figure 2.3 present the grammar rules used by Domato, Dharma, and Syzkaller to randomly generate CSS selectors, SVG `filter` attributes, and several system calls for file operations, respectively. The formality of the

```
<selector> = #<elementid>
<elementid> = htmlvar0000<int min=1 max=9>

<selector> = <element>
<element p=0.5> = <tagname>
<tagname> = a | abbr | acronym | address | applet | area | article | aside | ...
```

Figure 2.1: An example of reference issues. The context-free grammar rules used by Domato define CSS selectors with invalid elements.

```
filter_attributes :=
        filter="+filter_value+"

filter_value :=
        +uri+
        none
        inherit

uri :=
        url(#!element_id!)
        url(#xpointer(id('!element_id!')))
```

Figure 2.2: An example of type errors. The context-free grammar rules used by Dharma incorrectly define a `filter` attribute with any type of element while only a <`filter`> element is accepted.

grammar guarantees the syntactic correctness of generated API calls. Nevertheless, grammar-based API generation being unaware of the API context suffers from semantic issues, which can be categorized into three types.

Reference issues. If a grammar does not annotate object creation and destruction, its generated APIs may access non-existent objects or never have a chance to access certain valid ones. For instance, as shown in Figure 2.1, Domato predefines a fixed number of HTML elements and a list of HTML element tags to construct CSS selectors. However, the elements valid for access at a point are dynamically determined based on the API calls invoked (*i.e.*, generated) previously. In practice, a document randomly produced by Domato probably has more than 20 HTML elements or no <`address`> elements, which invalidates Domato's definition of CSS selectors.

Type errors. An API is designed to operate *specific* types of objects. Therefore, the API calls output from typeless grammars tend to misuse the objects of unmatched types. For example, Dharma accepts any type of SVG element for the value of a `filter` attribute

```
1 open(file ptr[in, filename], flags flags[open_flags], mode flags[open_mode]) fd
2 rename(old ptr[in, filename], new ptr[in, filename])
3 write(fd fd, buf buffer[in], count len[buf])
```

Figure 2.3: An example of state errors. Syzkaller's grammar does not evaluate the effects of a system call on the file name or the write property of a file descriptor. The affected file and file descriptor are likely to be inappropriately operated by another system call.

(see Figure 2.2). However, the attribute takes effect only when it is defined by a `<filter>` element. Therefore, the invocations of filtering APIs in an SVG document generated by Dharma are likely not functional.

State errors. The two aforementioned issues can be resolved by marking the lifetime and types of the objects used by APIs in the grammar, which is demonstrated by Syzkaller [7]. Nevertheless, an API may require an input object to have a valid state and meanwhile update the state of the object during its execution. However, existing context-free grammars that focus on describing API representation cannot deliver such underlying runtime semantics of an API. For instance, Figure 2.3 shows that Syzkaller's grammar is unable to present the effects of the system calls on the names (*i.e.*, state) of the files (*i.e.*, object) in a file system. As a result, an out-of-date file name caused by `rename()` can still be used by a system call afterward. Meanwhile, a new file name set through `open()` will never be referred to. Moreover, tracking whether or not a file descriptor (*i.e.*, object) is writable (*i.e.*, state) is also beyond the grammar's ability. Consequently, Syzkaller outputs invalid `write()` calls on the file descriptors that are read-only.

2.2 Approach

To overcome the common semantic issues we observe from the existing API fuzzers, we propose context-aware API fuzzing, which is based on a context-aware representation, namely RPG IR, to describe an API program for the purpose of fuzzing. An API program in RPG IR contains not only the API calls made in the program but also the program context that addresses the semantics of the API calls. The program context stores all the objects

created by the API calls in the program. As we summarize in §2.1, an API call incurs semantic errors normally because it tries to operate an object that is not in a valid state. Therefore, the program context intendedly maintains the lifetime and runtime properties of every created object. When a context-aware fuzzer generates a new API call in RPG IR, it interacts with the program context to fetch appropriate input objects for use and add the new objects created by the API call into the context. In addition, the fuzzer is responsible for emulating the changes of the API call to the state of any existing object. Meanwhile, a context-aware fuzzer performs mutation over an API program in RPG IR and meanwhile, ensures that the integrity of the program context is not corrupted after mutating any API call.

2.3 Challenges

To realize context-aware fuzzing driven by a formal representation for describing API calls, we need to address three main challenges.

Supporting API formats. The APIs in different domains being exposed by diverse programming languages have distinguishable formats with contrasting complexity. For example, system calls have a simple interface whose inputs and outputs are either integers or pointers to data buffers and structures. Meanwhile, an HTML document simultaneously involves three incompatible formats – JavaScript statements, DOM trees, and CSS rules – all of which have completely different syntax. Therefore, a context-aware representation for API description needs to be expressive enough to depict various API formats.

Delivering varied API semantics. The notable difference between individual APIs does not simply reside in their formats but, more importantly, in their specific semantics, namely how they interact with the context. Regarding an arbitrary API, there is no convention on (1) the number, types, and states of its input objects, (2) the number and types of its output objects, or (3) its changes to the state of the objects. Therefore, the API representation adopted by a context-aware fuzzer is expected to model all the API semantics related to object references, object type requirements and object states.

Managing complicated contexts. As we mention in §2.2, the key approach of a context-aware API fuzzer is to maintaining essential information about the objects in the context for fewer semantic errors in the generated API programs. We observe the complexity of the context, which is not straightforward to implement via RPG IR. When randomly generating an API at a given program location, a fuzzer relies on the context to retrieve the live objects of wanted types as the potential inputs of the API. To fully serve this purpose, the context basically needs to store the objects with their types, states, and lifetimes in the current program. In addition, the context should also tackle type inheritance, which is widely supported by modern API-based systems. For instance, the context is expected to provide all the `HTMLElements` for random selection when generating an API that operates an `Element` in the document. Furthermore, the context should also consider object scopes when managing object lifetime. For example, in an HTML document, the objects created by DOM APIs in an event handler can only be used locally. If object creation can be introduced at any level of nested scopes, the context becomes even more complicated.

CHAPTER 3

RPG IR

RPG IR is a formal and context-sensitive abstraction of API programs specifically designed for context-aware fuzzing. An RPG IR program basically consists of

- The API calls;
- The context information about the existing objects in the program based on *the evaluation of the effects of the API calls*.

The API calls are the main parts of a program, which are executed for finding bugs and also mutated for new programs. Meanwhile, the context information is also crucial for fewer semantic errors in the program. In this chapter, we present the detailed design of RPG IR.

3.1 Scope

RPG IR organizes the API calls and context information in a program into various scopes. Equation 3.1 describes a scope, which includes a sequence of API calls `APIS` and the objects defined in the scope `OBJS`. Note that a scope can be empty, which does not involve any API call. Also, no new objects can be created in the scope.

$$\texttt{SCOPE} = (\texttt{APIS}, \texttt{OBJS}) \tag{3.1}$$

$$\texttt{APIS} = \{\texttt{API}_1, \texttt{API}_2, ..., \texttt{API}_n\},\ n \geq 0 \tag{3.2}$$

$$\texttt{OBJS} = \{\texttt{OBJ}_1, \texttt{OBJ}_2, ..., \texttt{OBJ}_m\},\ m \geq 0 \tag{3.3}$$

The only global scope $\texttt{SCOPE}_\texttt{G}$ naturally represents an entire API program $\mathcal{P}$ in RPG IR.

$$\mathcal{P} = (\texttt{SCOPE}_\texttt{G}) \tag{3.4}$$

Except for the global scope, all the other scopes in the program are created by particular API calls. (see Equation 3.11 in §3.3).

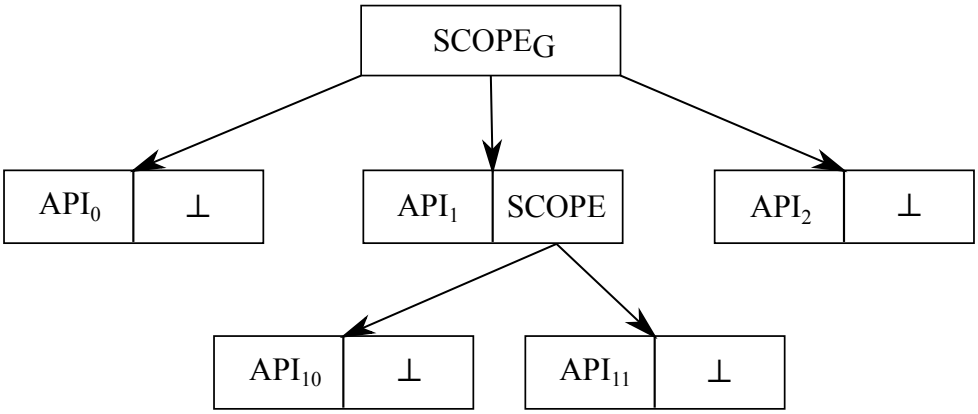


Figure 3.1: The scope structure of an example program in RPG IR.

Figure 3.1 illustrates the conventional scope tree adopted by RPG IR, which is widely applied to all sorts of API programs in practice. For example, in an HTML document, `document` and `window` are two predefined objects that can be freely accessed and are thus placed in the global scope. Meanwhile, nested scopes are required for event handlers in JavaScript, which have local objects. Note that the model does not support branches. In other words, an API call is not supposed to create two new scopes. However, branches are not considered meaningful in the context of API fuzzing. We expect all the API calls in a generated program to be executed by the target. Most importantly, none of the existing fuzzers mentioned in §2.1, including the context-aware ones, have a systematic support for object scopes as RPG IR does, which reflects the generality of RPG.

Program location. Based on the scope model, any location in an RPG IR program can be described as a two-element tuple, as shown in Equation 3.5.

$$\texttt{LOC} = (\texttt{SCOPE}, \texttt{INDEX}) \tag{3.5}$$

More specifically, `LOC` represents a program location at the `INDEX`th API call in the scope `SCOPE`. Based on this, we naturally define the ordering of two program locations in Equa-

tion 3.6.

$$\texttt{LOC}_\texttt{a} \prec_\texttt{s} \texttt{LOC}_\texttt{b} \iff \texttt{LOC}_\texttt{a} \text{ is visited before } \texttt{LOC}_\texttt{b} \text{ during the pre-order traversal of the sub-tree rooted by the scope s.} \tag{3.6}$$

RPG IR utilizes this ordering between two program locations to validate object references in Equation 3.10.

3.2 Object

RPG IR models an object created in a particular scope (*i.e.*, `OBJ` in Equation 3.1) as a tuple of its identifier `ID`, birth `BIRTH`, and death `DEATH` locations in the scope, type `TYPE`, and state `STATE`.

$$\texttt{OBJ} = (\texttt{ID}, \texttt{BIRTH}, \texttt{DEATH}, \texttt{TYPE}, \texttt{STATE}) \tag{3.7}$$

$$\texttt{STATE} = \{(\texttt{K}_1, \texttt{V}_1), (\texttt{K}_2, \texttt{V}_2), ..., (\texttt{K}_n, \texttt{V}_n)\},\ n \geq 0 \tag{3.8}$$

$$\texttt{STATE}[\texttt{K}_i] = \texttt{V}_i,\ 1 \leq i \leq n \tag{3.9}$$

Figure 3.2 shows two example objects described by RPG IR, including an `HTMLInputElement` object and a file descriptor object.

For a local object, RPG IR uses `BIRTH` and `DEATH` to specify the program locations of the API call that creates and deletes the object, respectively. RPG IR assumes that the life of a local object is automatically terminated at the end of the scope where it is created and uses this location as the default `DEATH` for the object. Before being deleted, a local object can be accessed by an API call located anywhere after its definition under the current scope. In contrast, an object in the global scope can be used anywhere in a program, whose `BIRTH` and `DEATH` are not specified by RPG IR. We summarize the restrictions on object references adopted by RPG IR into a formal form described by Equation 3.10. $\mathcal{U}(\texttt{OBJ}, \texttt{LOC})$ checks whether a given object `OBJ` can be used by the API call at the location `LOC` in an RPG IR

program.

$$\mathcal{U}(\texttt{OBJ}, \texttt{LOC}) = \begin{cases} \texttt{TRUE} & \texttt{SCOPE}_{\texttt{OBJ}} = \texttt{SCOPE}_{\texttt{G}} \; or \\ & \texttt{OBJ}[\texttt{BIRTH}] \prec_{\texttt{SCOPE}_{\texttt{OBJ}}} \texttt{LOC}, \; \texttt{LOC} \prec_{\texttt{SCOPE}_{\texttt{OBJ}}} \texttt{OBJ}[\texttt{DEATH}] \\ \texttt{FALSE} & \text{o.w.} \end{cases} \tag{3.10}$$

In addition, the state of an object is abstracted as a list of properties. A property is a key and value pair $(\texttt{K}, \texttt{V})$, where the key K is a string. For ease of presentation, we also index the state by K to represent the property value V, as shown in Equation 3.9.

ID	v3		
BIRTH	*S*, 0	**DEATH**	-
TYPE	HTMLInputElement		
STATE	*tag*	"input"	

(a)

ID	v5		
BIRTH	*S*, 2	**DEATH**	*S*, 5
TYPE	FD		
STATE	*W*	1	

(b)

Figure 3.2: Examples of object representation in an RPG IR program.

The state of an object can be set by the API calls that use the object and referenced in two ways in an RPG IR program. First, it can be used to define the argument of an API. For example, the `tag` value of an HTML element illustrated in Figure 3.2(a) can be used as a selector in certain DOM APIs. More importantly, it is used to check whether the object can be accessed by a particular API. For instance, a `write()` call should only accept a writable file descriptor (FD) whose W property equals 1, as shown in Figure 3.2(b).

Based on the context information recorded for every object, the three types of semantic issues mentioned in §2.1 are largely reduced in an RPG IR program. More specifically, the object scopes and definition locations are used to determine the visible and defined objects that can be accessed by an API call at any program location, which avoids reference issues. To get rid of type errors, the context marks the object types and thus enables an API call to operate the objects of specific types only. More importantly, the object states recorded

in the context are queried by RPG during API generation, which resolves the state errors. In general, the semantic correctness of an API program in RPG IR originates from the proposed object model.

3.3 API

RPG IR presents a context-based API model that formally describes the syntax and semantics of an arbitrary API. Equation 3.11 depicts the outline of the model. Basically, RPG IR formally defines an API call as a tuple of the return values `RETURN` (§3.3.1), a list of arguments `ARGS` that may be context-dependent (§3.3.2), the contextual effects `EFFECTS` (§3.3.3), a new scope `SCOPE` representing the API's body (§3.3.4), a print function `PRINT` that translates the API call in IR into plain text, and an optional print function `EPRINT` that returns the text to be displayed after the body (§3.3.5). Note that $\texttt{API}[\texttt{SCOPE}]$ here represents the scope created by the API call, which is different from the scope where the API is called (*i.e.*, $\texttt{SCOPE}_{\texttt{API}}$).

$$\texttt{API} = (\texttt{RETURN}, \texttt{ARGS}, \texttt{EFFECTS}, \texttt{SCOPE}, \texttt{PRINT}, \texttt{EPRINT}) \tag{3.11}$$

The API model built by RPG IR is prior to the existing ones mentioned in §2.1 which fail to fully address the object-related semantics of an API.

3.3.1 API Return

An API call may create zero or more objects as its return, as defined in Equation 3.12.

$$\texttt{RETURN} = \{\texttt{OBJ}_1, \texttt{OBJ}_2, ..., \texttt{OBJ}_n\},\ n \geq 0 \tag{3.12}$$

Object creation is a type of contextual effects of an API. More specifically, every returned object is added into its corresponding scope, which is formally described in §3.3.3. It is worth noting that a returned object usually belongs to the scope of the API call. However,

certain APIs even being invoked in a local scope create global objects. For example, an HTML element node defined anywhere in the XML tree of an HTML document is globally accessible.

3.3.2 API Arguments

ARGS in Equation 3.11 is naturally defined as a list of arguments. The length of the list can be zero, which indicates that the API call does not have any argument.

$$\texttt{ARGS} = \{\texttt{ARG}_1, \texttt{ARG}_2, ..., \texttt{ARG}_n\},\ n \geq 0 \tag{3.13}$$

Arguments are the most fundamental concept in the API model proposed by RPG IR. The syntax and semantics of an API call are completely addressed through the arguments. Meanwhile, generating or mutating an API call is actually conducted by generating or mutating its arguments. Equation 3.14 presents the definition of an argument, namely ARG, in RPG IR. Supposing that $\texttt{ARG} \in \texttt{API[ARGS]}$,

$$\texttt{ARG} = (\texttt{TYPE}, \texttt{IMPORTS}, \texttt{ASSERTS}, \texttt{ARGS}, \texttt{VALUE}) \tag{3.14}$$

First, TYPE is literally the type of argument.

Next, IMPORTS is a list of objects that used by ARG, which implies the context dependence of the argument (see Equation 3.15).

$$\texttt{IMPORTS} = \{\texttt{OBJ}_1, \texttt{OBJ}_2, ..., \texttt{OBJ}_n\},\ n \geq 0, \tag{3.15}$$

$$\forall 1 \leq i \leq n,\ \mathcal{U}(\texttt{OBJ}_i, \texttt{LOC}_{\texttt{API}}) = \texttt{TRUE} \tag{3.16}$$

To avoid reference errors, all the imported objects are required to be accessible from the owner API of the argument (see Equation 3.16). Figure 3.3(b) shows a context-dependent argument that presents a CSS ID selector in an HTML document. The CSS ID selector uses

the identifier of an HTMLInputElement object in its value. A context-free argument is not based on any imported object, which can be randomly generated without context.

Furthermore, ASSERTS enforces specific state requirements on the imported objects, which prevent state errors in a generated program, as mentioned in §2.1.

$$\texttt{ASSERTS} = \{\texttt{ASSERT}_1, \texttt{ASSERT}_2, ..., \texttt{ASSERT}_n\},\ n \geq 0 \tag{3.17}$$

$$\texttt{ASSERT} = \texttt{HAS\_PROPERTY}(\texttt{OBJ}, \texttt{K}) \tag{3.18}$$

$$\mid \texttt{PROPERTY\_EQ}(\texttt{OBJ}, \texttt{K}, \texttt{V}) \tag{3.19}$$

$$\mid \texttt{PROPERTY\_NEQ}(\texttt{OBJ}, \texttt{K}, \texttt{V}),\ \texttt{OBJ} \in \texttt{IMPORTS},\ \texttt{K} \text{ is a key string;} \tag{3.20}$$

$$\texttt{HAS\_PROPERTY}(\texttt{OBJ}, \texttt{K}) \Rightarrow \exists \texttt{V},\ \langle \texttt{K}, \texttt{V} \rangle \in \texttt{OBJ}[\texttt{STATE}] \tag{3.21}$$

$$\texttt{PROPERTY\_EQ}(\texttt{OBJ}, \texttt{K}, \texttt{V}) \Rightarrow \langle \texttt{K}, \texttt{V} \rangle \in \texttt{OBJ}[\texttt{STATE}] \tag{3.22}$$

$$\texttt{PROPERTY\_NEQ}(\texttt{OBJ}, \texttt{K}, \texttt{V}) \Rightarrow \langle \texttt{K}, \texttt{V} \rangle \notin \texttt{OBJ}[\texttt{STATE}] \tag{3.23}$$

RPG IR currently supports three types of state checks on an imported object.

- HAS_PROPERTY checks whether the object has a specific property in its state (Equation 3.21).
- PROPERTY_EQ checks whether the value of a specific property of the object equals a given value (Equation 3.22).
- PROPERTY_NEQ checks whether the value of a specific property of the object does not equal a given value (Equation 3.23).

For example, Figure 3.4 illustrates that RPG IR applies PROPERTY_EQ to check whether or not the file descriptor imported for writing has its W property set to 1.

More importantly, an argument has a recursive definition in RPG IR. ARGS in Equation 3.14 specifies a list of sub-arguments being used to construct the argument. As shown in Figure 3.3, a :focus CSS pseudo-class in RPG IR is built upon one sub-argument, which is the aforementioned CSS ID selector.

The last element in the definition of an argument is VALUE, representing the argument

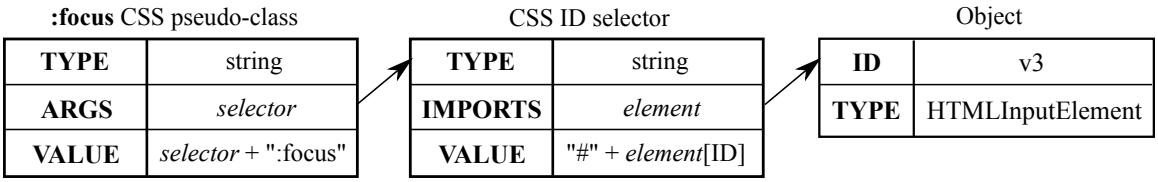


Figure 3.3: Examples of argument definitions in RPG IR. (b) describes a CSS ID selector, which is a context-dependent argument and has a value of `#v3`. (a) describes a `:focus` CSS pseudo-class built upon another argument, namely the CSS ID selector, whose value is `#v3:focus`.

value being passed into an API call. `VALUE` is resolved by an expression comprising the imported objects and sub-arguments that outputs a value of the specified type `TYPE`. Equation 3.24 describes `VALUE`, which can be an imported object, a primary expression, or a string concatenation of primary expressions.

$$\texttt{ARG}[\texttt{VALUE}] = \texttt{OBJ},\ \texttt{OBJ} \in \texttt{ARGS}[\texttt{IMPORTS}] \tag{3.24}$$

$$\mid \texttt{EXP}(\texttt{ARG}) \tag{3.25}$$

$$\mid \texttt{EXP}_1(\texttt{ARG}) + \texttt{EXP}_2(\texttt{ARG}) + ... + \texttt{EXP}_n(\texttt{ARG}),\ 2 \leq i \leq n \tag{3.26}$$

First, if the argument value is an imported object, the argument is supposed to be an object reference. A primary expression has various formats, including a string or number literal (Equation 3.27), the value of a sub-argument (Equation 3.28), the identifier of an imported object (Equation 3.29), or the value of a specific property of an imported object (Equation 3.30).

$$\texttt{EXP}(\texttt{ARG}) = \textit{literal} \tag{3.27}$$

$$\mid \texttt{ARG}'[\texttt{VALUE}],\ \texttt{ARG}' \in \texttt{ARG}[\texttt{ARGS}] \tag{3.28}$$

$$\mid \texttt{OBJ}[\texttt{ID}],\ \texttt{OBJ} \in \texttt{ARGS}[\texttt{IMPORTS}] \tag{3.29}$$

$$\mid \texttt{OBJ}[\texttt{STATE}][\texttt{K}],\ \texttt{OBJ} \in \texttt{ARGS}[\texttt{IMPORTS}] \tag{3.30}$$

Considering the API call `Document.createElement()` illustrated in Figure 7.1, both of its arguments define their values in primary expressions. The value of *document* is the identifier

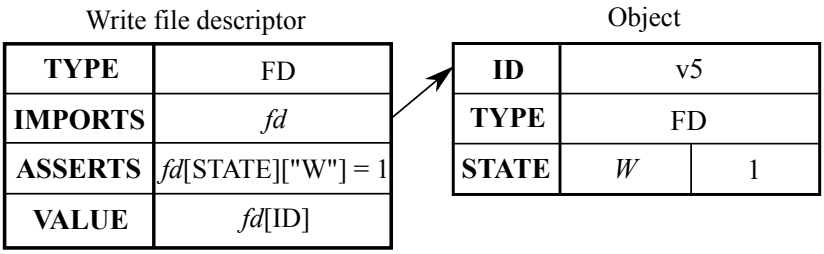


Figure 3.4: Another example of argument definition that involves an assertion on the object state. The argument represents a reference to a write file descriptor, whose `W` property is expected to be 1.

of a `Document` object. Meanwhile, the value of *tag* is simply a string literal – "`input`." In Figure 3.3, the values of both CSS pseudo-class and CSS ID selector are defined as string concatenations, which involve the value of a sub-argument (*i.e.*, *selector*) and the property value of an imported `HTMLInputElement` object, respectively.

Unlike the domain-specific API fuzzers that use custom methods to describe the API arguments in restricted formats, the recursive argument definition with flexible value representation enables RPG IR to formally express all kinds of API designs. Similar to RPG IR, the grammars adopted by existing grammar-based fuzzers are also expressive, which allow one to recursively define complicated values with primary values (see Figure 2.1 and Figure 2.2 for examples). Nevertheless, arguments in RPG IR present their semantics through object imports and state assertions, which are not delivered by context-free grammars.

3.3.3 API Effects

As is known, API calls operate objects. Therefore, RPG IR formally models the contextual effects of an API call, called API, into four types of object operations.

$$\texttt{EFFECTS} = \{\texttt{EFFECT}_1, \texttt{EFFECT}_2, ..., \texttt{EFFECT}_n\},\ n \geq 0 \tag{3.31}$$

$$\texttt{EFFECT} = \texttt{NEW\_OBJECT}(\texttt{API}, \texttt{OBJ}) \mid \texttt{NEW\_GOBJECT}(\texttt{OBJ}) \tag{3.32}$$

$$\begin{aligned} &\text{only for } \texttt{OBJ} \in \texttt{API}[\texttt{RETURN}], \\ &\texttt{DEL\_OBJECT}(\texttt{API}, \texttt{OBJ}) \mid \texttt{SET\_STATE}(\texttt{OBJ}, \texttt{K}, \texttt{V}), \\ &\text{for } \texttt{OBJ} \in \texttt{API}[\texttt{RETURN}]\ \textit{or} \\ &\texttt{OBJ} \in \texttt{API}[\texttt{ARGS}_{i_1}][\texttt{ARGS}_{i_2}]...[\texttt{ARGS}_{i_m}][\texttt{IMPORTS}], \exists i_1, i_2, ..., i_m, m \geq 1 \end{aligned} \tag{3.33}$$

Creating a local object. As defined in Equation 3.34, the operation NEW_OBJECT adds a local object returned by an API call into the scope where the API call is located. The definition location BIRTH of the object is the same as that of the API call API.

$$\begin{aligned} \texttt{NEW\_OBJECT}(\texttt{API}, \texttt{OBJ}) \Rightarrow &\texttt{SCOPE}_{\texttt{API}}[\texttt{OBJS}] = \texttt{SCOPE}_{\texttt{API}}[\texttt{OBJS}] \cup \{\texttt{OBJ}\}, \\ &\texttt{OBJ} \in \texttt{API}[\texttt{RETURN}] \end{aligned} \tag{3.34}$$

Creating a global object. Slightly different from NEW_OBJECT, NEW_GOBJECT adds a global object returned from an API call into the global scope.

$$\begin{aligned} \texttt{NEW\_GOBJECT}(\texttt{OBJ}) \Rightarrow &\texttt{SCOPE}_{\texttt{G}}[\texttt{OBJS}] = \texttt{SCOPE}_{\texttt{G}}[\texttt{OBJS}] \cup \{\texttt{OBJ}\}, \\ &\texttt{OBJ} \in \texttt{API}[\texttt{RETURN}] \end{aligned} \tag{3.35}$$

Deleting an existing object. An API call can delete an existing object from the scope. RPG IR proposes DEL_OBJECT to address this operation, which sets the DEATH location

of the object to the location of the API call. Note that the deleted object should be imported by an argument or its underlying sub-arguments of the API call. `DEL_OBJECT` has a wide variety of use cases for real-world APIs. For example, a file descriptor after being closed by `close()` should not be accessed afterward in a system call program. Also, `WebGLRenderingContext.deleteBuffer()`, a WebGL API, invalidates a `WebGLBuffer` object for any future use. The semantics of both APIs can be described by `DEL_OBJECT`.

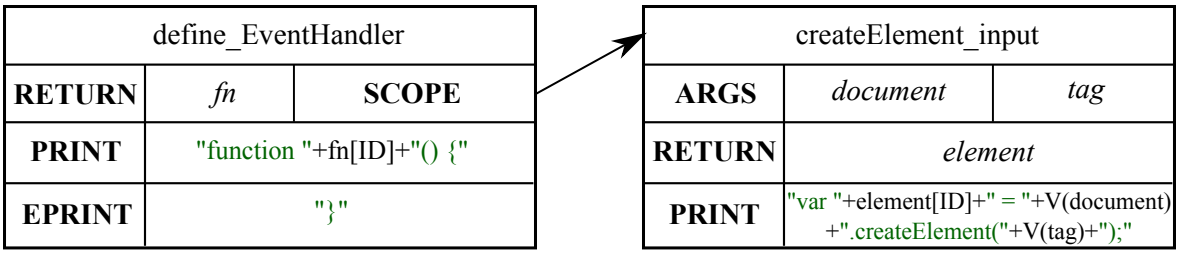
$$\begin{aligned}\texttt{DEL\_OBJECT}(\texttt{API}, \texttt{OBJ}) \Rightarrow \texttt{SCOPE}_{\texttt{OBJ}}[\texttt{OBJS}] = \texttt{SCOPE}_{\texttt{OBJ}}[\texttt{OBJS}] - \{\texttt{OBJ}\},\\ \texttt{OBJ} \in \texttt{API}[\texttt{ARGS}_{i_1}][\texttt{ARGS}_{i_2}]...[\texttt{ARGS}_{i_m}][\texttt{IMPORTS}], \exists i_1, i_2, ..., i_m, m \geq 1 \quad (3.36)\end{aligned}$$

Updating object state. The last operation `SET_STATE` is used to modify the state of an object for an API call. RPG IR supports the operation to be performed on both the returned objects and the imported objects of the API call. As defined by Equation 3.37, `SET_STATE` manages to set an arbitrary value for a specified property of an object.

$$\begin{aligned}\texttt{SET\_STATE}(\texttt{OBJ}, \texttt{K}, \texttt{V}) \Rightarrow \texttt{OBJ}[\texttt{STATE}][\texttt{K}] = \texttt{V},\\ \texttt{OBJ} \in \texttt{API}[\texttt{RETURN}] \ or\\ \texttt{OBJ} \in \texttt{API}[\texttt{ARGS}_{i_1}][\texttt{ARGS}_{i_2}]...[\texttt{ARGS}_{i_m}][\texttt{IMPORTS}], \exists i_1, i_2, ..., i_m, m \geq 1 \quad (3.37)\end{aligned}$$

For example, the `W` property of a file descriptor created by `open()` with a write-only or read and write flag is set to be 1. The effect of `WebGLRenderingContext.bindBuffer()` is described by setting the value of the `isBound` property of an imported `WebGLBuffer` object to 1 through `SET_STATE`. The `isBound` property indicates the usability of the `WebGLBuffer` objects, which is verified by most WebGL APIs operating `WebGLBuffer`. Based on `SET_STATE`, RPG IR is able to maintain object states for avoiding state errors during API generation.

In general, RPG IR relies on those contextual operations to formally evaluate the changes brought by an API call to the objects, which also lays a foundation for reduced semantic errors in a generated program. By contrast, none of the existing API fuzzers manage to



(a) (b)

Figure 3.5: Examples of converting two DOM API calls in RPG IR into plain text. The text of `createElement_input` in (b) is simply defined by its `PRINT` function. By contrast, the text of `define_EventHandler` in (a) starts with the string returned from `PRINT` followed by the text of its body, namely `createElement_input`, and ends with the string output by `EPRINT`. The final text varies by the identifiers of the involved objects. A valid example can be `function f3() { var v3 = document.createElement("input"); }`.

express such semantics of various APIs, which reveals the advantage of RPG IR.

3.3.4 API Body

An API call may create a new scope that contains other API calls (see Figure 3.1). The contained API calls form the body of the original API call. For instance, RPG IR treats an element node in the XML tree of an HTML document as an API call. The body of such a node is thus a list of other element nodes. In practice, RPG IR also considers a JavaScript function defined under `<script>` in an HTML document as an API call, whose body encloses a list of DOM API calls. Compared to the previous fuzzers like Syzkaller, which simply treats a program as a linear sequence of API calls that do not contain each other, RPG IR is more expressive in describing API programs in different formats.

3.3.5 API Display

At the end, we discuss how RPG IR uses print functions to lower an API call in RPG IR into plain text for testing and human reading. The definition of an API call involves two print functions: `PRINT` and `EPRINT` (see Equation 3.11).

The string representation s of an API call starts with the text returned from `PRINT`. RPG IR defines `PRINT` as a string concatenation of string literals and the argument values and

Algorithm 1: Translating an API call in RPG IR into human-readable text.

Input: An API call `API` in RPG IR
Result: A string representing the API call in plain text

```
Function STR(API)
    s ← PRINT(API)
    if API[SCOPE] ≠ ∅ then
        for each API' ∈ API[SCOPE][APIS] do
            s ← s + PRINT(API')
        end
        s ← s + EPRINT(API)
    end
    return s
```

returned objects of the API call.

$$\texttt{PRINT}(\texttt{API}) = \texttt{EXP}_1(\texttt{API}) + \texttt{EXP}_2(\texttt{API}) + ... + \texttt{EXP}_n(\texttt{API}),\ n \geq 1 \tag{3.38}$$

$$\texttt{EXP}(\texttt{API}) = \textit{literal} \tag{3.39}$$

$$\mid \texttt{ARG}[\texttt{VALUE}],\ \texttt{ARG} \in \texttt{API}[\texttt{ARGS}] \tag{3.40}$$

$$\mid \texttt{OBJ}[\texttt{ID}],\ \texttt{OBJ} \in \texttt{API}[\texttt{RETURNS}] \tag{3.41}$$

If the API call creates a new scope containing other API calls, the string representations of those sub-calls need to be appended to s. The tailing text of the API call is returned by `EPRINT`, which has exactly the same definition as `PRINT`.

3.4 Summary

RPG IR is a formal and context-aware representation of an API program that describes API syntax and semantics in a self-contained way. We highlight its important design choices, including (1) the recursive argument definition and argument value expressions enable RPG IR to flexibly express various API formats, (2) the scope and object models introduce a object-based context to an API program in RPG IR, (3) the argument imports, object state assertions, and API effect model manage to comprehensively deliver the object-dependence of an API call, which helps a fuzzer to reduce semantic issues in a generated API program.

CHAPTER 4
CONTEXT-AWARE API FUZZING VIA RPG IR

In this chapter, we present context-aware API fuzzing based on RPG IR.

4.1 Context-aware API Generation

A context-aware API fuzzer generates random API calls with reduced semantic errors based on the context information recored by RPG IR including scopes and objects. Figure 4.1 presents the pseudocode of this generation process (lines 9-31). Generating an API call first requires the specification of the API, notated as `api_spec`. The context information is delivered by `scope` and `index` which represents the scope and the location in the scope where the API is called. The arguments of the API call are generated first (lines 13-16) through a separate process called `generate_arg()`. The returned object is then constructed if it exists (lines 19-20). After that, the specified effects related to object lifetime management and object runtime properties are applied to the returned object and the objects imported by the arguments (lines 23-24). If the targeted API is specified with certain child APIs, the fuzzer creates a new scope associated with the API call. A sequence of random calls of the child APIs are recursively generated in the newly created scope afterward based on the procedure described at lines 1-7.

Figure 4.2 presents the pseudocode for generating a random argument of an API call. The pseudocode first imports the objects required by the argument (line 3-10). More specifically, the scope is queried for the live objects that have a matched type. Those objects are filtered by the object state assertions. An imported object is randomly selected from the objects that are left at the end. Next, the sub-arguments of the argument are recursively generated (line 13-18). Finally, the argument value is resolved (line 21). Note that argument generation may fail when there is no valid object for importing which can further lead to a failure of the API

```
def generate_apis(scope, api_specs):
    scope.apis = []
    for _ in range(scope.limit):
        api_spec = random.choice(api_specs)
        api = generate_api(api_spec, scope, len(scope.apis))
        if api is not None:
            scope.apis.append(api)

def generate_api(api_spec, scope, index):
    api = API(api_spec)

    # Generate the arguments
    for arg in api.args:
        ok = generate_arg(arg, scope, index)
        if not ok:
            return None

    # Generate the optional returned object
    if api.return_type is not None:
        api.ret = object_factory.create(api.return_type)

    # Apply the effects
    for effect in api.effects:
        api.effect()

    # Generate the optional body
    if len(api.child_api_specs) > 0:
        api.scope = create_scope(api)
        api.scope.apis = generate_apis(api.scope, api.children_api_specs)

    return api
```

Figure 4.1: Pseudocode for generating a random API call at a specific program location (*i.e.*, `scope` and `index`) based on the API specification (*i.e.*, `api_spec`).

generation and is thus particularly handled by the fuzzer.

Once the fuzzer manages to generate a random API call in a particular scope, generating a random API program in RPG IR becomes straightforward as presented in Figure 4.3. Note that a program in RPG IR is represented by the global scope (see §3.3). Therefore, the fuzzer needs to create a global scope for the program. After that, the fuzzer determines the APIs to be invoked in the global scope and generates random calls for them through the aforementioned process (*i.e.*, `generate_api` in Figure 4.1).

```
def generate_arg(arg, scope, index):
    # Generate the imported objects
    for obj_type in arg.object_types:
        objects = scope.get_objects(obj_type, index)
        for cond in arg.asserts:
            objects = filter(cond, objects)
        if len(objects) > 0:
            arg.imports.append(random.choice(objects))
        else:
            return None

    # Generate the sub-arguments
    for sub_arg in arg.args:
        ok = generate_arg(sub_arg, scope, index)
        if ok is not None:
            arg.args.append(sub_arg)
        else:
            return None

    # Calculate the value of the argument
    arg.value = arg.VALUE()

    return arg
```

Figure 4.2: Pseudocode for generating a random argument for the API call at a specific program location (*i.e.*, `scope` and `index`).

```
def generate_program():
    program = context_manager.create_global_scope()
    program.apis = []
    for api_spec in range(global_api_specs):
        api = generate_api(api_spec, program, len(program.apis))
        if api is not None:
            program.apis.append(api)
```

Figure 4.3: Pseudocode for generating a random RPG IR program.

4.2 Context-aware API Mutation

A context-aware API fuzzer also supports several types of mutations over an existing program in RPG IR without corrupting the maintained program context. The mutation algorithms mostly rely on the generation process of APIs and arguments described in §4.1.

API insertion. The fuzzer first randomly selects an existing scope in the program. Then, a random API call is generated at a random location in the scope, which is implemented by `generate_api()`. Note that the birth and death location of all the objects created in the

scope needs to be updated afterward.

API deletion and replacement. First, the fuzzer can randomly remove an existing API call in the program. To avoid semantic errors in the mutated program, the API calls whose effects have been applied to the context are excluded. Furthermore, the fuzzer can replace an existing API call with a new one by removing the API call and generating a new one at the same program location.

Argument mutation. The fuzzer randomly selects an argument of an existing API and regenerates the argument itself or any of its underlying sub-arguments via `generate_arg()`. To avoid semantic errors, the fuzzer should not consider regenerating the arguments whose imported objects are operated by an API call (*i.e.*, affected by API effects).

CHAPTER 5

FREEDOM: CONTEXT-AWARE DOM FUZZING VIA RPG IR

In this chapter, we present how we apply RPG IR to fuzz one of the most widely used API-based systems – web browsers – in a context-aware manner. More specifically, we propose FreeDom, a DOM fuzzer that strictly follows the API model based on RPG IR.

5.1 Background

5.1.1 DOM: An HTML Document Representation

A web browser accepts HTML documents as its input, which follows the Document Object Model (DOM) standardized by W3C. Figure 5.1 presents an example of a document that consists of the following parts:

The initial DOM Tree. The DOM logically treats a document as a tree structure, and each tree node represents an object to be rendered. An HTML document file specifies the initial object tree. Most notable nodes represent *elements*. An element is identified by its tag and has its own semantics. A leaf node of an element can be another element or a text node. Moreover, each element owns a list of *attribute* nodes. The attributes control various aspects of the rendering behavior of an element. Note that for each element, the DOM standard specifies exactly what child elements and attributes it owns and whether it can have text in its content. For example, the DOM tree presented in Figure 5.1 includes a `<form>` element that owns two attributes and two child elements.

CSS Rules. Cascading Style Sheets (CSS) are used to specify in which style the elements in the document are rendered. Contained by `<style>`, a CSS rule consists of (1) a group of CSS selectors, which determines the elements to be styled, and (2) a group of CSS properties, each of which styles a particular aspect of the selected elements. For instance,

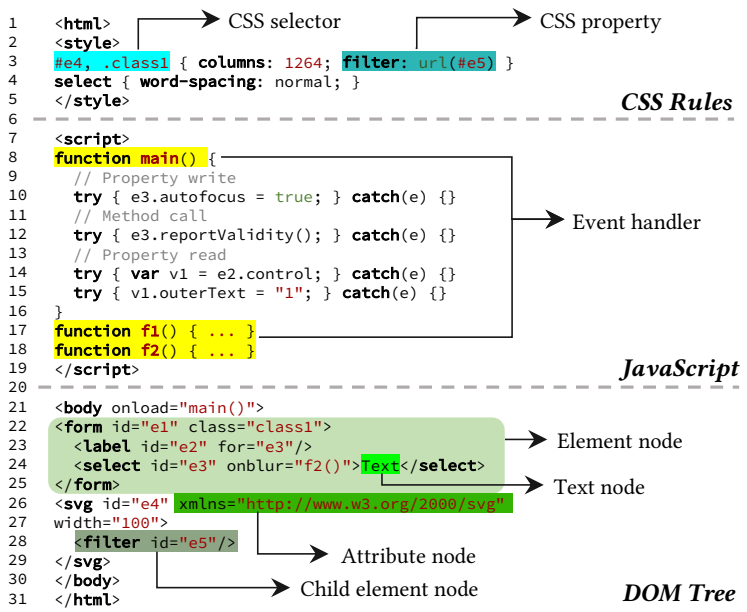


Figure 5.1: An example of an HTML document. The document is composed of three main parts that have distinct syntax and semantics: (1) A DOM tree specifies objects to be displayed at the very beginning. (2) A list of CSS rules further decorates the objects in the tree. (3) The JavaScript codes modify the layout and effect of the objects at runtime.

Line 3 in Figure 5.1 requires the <form> selected by `.class1` to split its content into 1,264 columns.

Event Handlers. To provide the interactivity of a web page, the DOM standard defines various events being triggered at specific timing points or user inputs. Event handlers can be registered in <script> so as to programmatically access or control the objects in the tree at runtime. For example, in Figure 5.1, `main()` and `f2()` are executed when a document is loaded and the <select> element loses focus, respectively. An event handler calls DOM APIs declared by the specification to manipulate DOM objects. Typical DOM APIs include object property accesses and object method invocations. Currently, all the popular browsers expose DOM APIs in JavaScript.

DOM fuzzer	Year	Type	Method	Str.	Ctx.	Cov.	Active
domfuzz [26]	2008	D	G	-	-		
Bf3 [5]	2010	S	G				
cross_fuzz [27]	2011	D	G	-	-		
Dharma [2]	2015	S	G	✓			✓
Avalanche [4]	2016	S	G	✓			✓
Wadi [3]	2017	S	G	✓			
Domato [1]	2017	S	G	✓			✓
FreeDom	2020	S	G/M	✓	✓	✓	✓

Str.: Structure-aware, **Ctx.**: Context-aware, **Cov.**: Coverage-guided
D: Dynamic, **S**: Static, **G**: Generative, **M**: Mutational

Table 5.1: The classification of existing DOM fuzzers. Dynamic fuzzers themselves are web pages executed by the target browser, while static fuzzers generate documents first and then test them. FreeDom is a static DOM fuzzer, which supports both blackbox generation and coverage-guided mutation in a context-aware manner.

5.1.2 DOM Engine Bugs

A browser runs a DOM engine (*e.g.*, WebKit in Apple Safari and Blink in Google Chrome), which literally implements the DOM specification so as to interpret an HTML document. We aim to use FreeDom to find memory errors triggered by a DOM engine when operating malformed documents. Such client-side bugs result in data breaches or even remote code execution in the context of a renderer process and therefore have always been considered one of the most significant security threats to end users over the past decade. Though browser vendors exert endless efforts to eliminate DOM engine bugs [18, 19, 20, 21, 22], there still have been quite a few full browser exploit chains that target DOM bugs in recent years [23, 24, 25], including one developed by us based on a bug found by FreeDom in Safari.

5.1.3 A Primer on DOM Fuzzing

The giant and rapidly growing DOM specification describes an extremely complex format for an HTML document. Hence, fuzzing, which requires minimal knowledge about the internals of the target software, becomes the most preferable approach for finding DOM engine bugs in practice. Over the last decade, researchers have proposed numerous DOM fuzzers,

which are summarized in Table 5.1. The earliest DOM fuzzers, such as domfuzz [26] and cross_fuzz [27], ran the fuzzer code in JavaScript together with a seed document in the same window. At runtime, it crawled the available elements on the page and invoked random DOM API calls to manipulate them on-the-fly until the browser crashed. The popularity of such *dynamic* fuzzers has declined because a target browser instance ages after a long run, which results in unstable executions and irreproducible crashes. By contrast, most recent fuzzers are *static* [5, 2, 3, 1] and generate syntactically correct documents from scratch based on static rules or grammars that describe the specification and execute every document with a fresh browser instance for a limited amount of time. Since the rules and grammars used by those fuzzers are not fully context-sensitive, the generated documents suffer from semantic errors. As typical blackbox fuzzers, they do not utilize existing testcases and feedback information for input generation.

5.2 Motivation

In this section, we systematically analyze the defects of conventional generation-based DOM fuzzers. Their common approach fails to construct inputs with complex semantics and, more importantly, restricts the exploration of coverage-guided mutation-based DOM fuzzing, which can be resolved by RPG IR.

5.2.1 On the Ineffectiveness of Static Grammars

Previous research has pointed out that one crux of fuzzing complex software effectively is to avoid semantic errors [28], not excepting DOM engines. Nevertheless, recent DOM fuzzers like Domato use various context-free grammars to describe an HTML document. Such a static grammar largely guarantees the syntactic correctness of a generated input, but it is unable to describe every data dependence throughout the input. As motivating examples, we summarize the typical context-dependent values (CDVs) in a document that Domato's grammar cannot correctly describe.

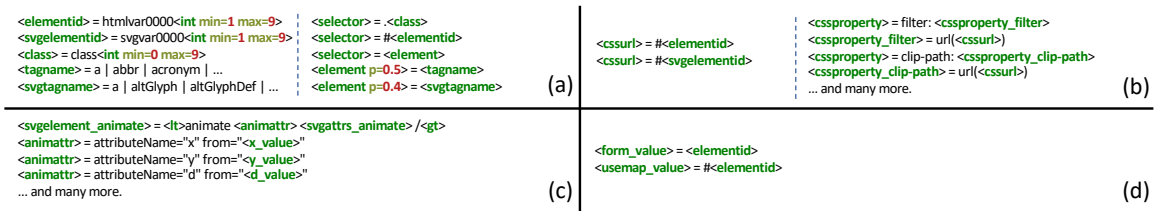


Figure 5.2: The grammar rules used by Domato that incorrectly generate four types of context-dependent values, including (a) CSS selectors, (b) CSS property values, (c) attribute names, and (d) attribute values.

CDV1: CSS selectors. CSS selectors explicitly refer to one or more elements to be styled by `id`, `class`, or tag. Domato expresses such references as shown in Figure 5.2(a). Basically, Domato only considers a fixed number of HTML and SVG elements, a fixed number of predefined class names, and all the HTML and SVG tags for styling. The contradicted fact is that the number of elements and the tags or `class` names available for reference in a document randomly generated by Domato are *undetermined* before generation. In practice, a document output by Domato probably has more than 30 HTML elements, only five SVG elements, or simply no `<abbr>` elements, which are the counterexamples to the grammar rules listed in Figure 5.2(a). In such cases, reference errors may occur; meanwhile, particular valid elements are never utilized.

CDV2: CSS property. Certain CSS property values also refer to existing elements. Normally, an element being referred to is required to not only be live but also have a particular type according to the DOM standard. Nevertheless, Domato incorrectly describes this data dependence as shown in Figure 5.2(b). Instead of any type of element, the standard only allows an SVG `<clipPath>` element and an SVG `<filter>` element to be used in the value of a `clip-path` CSS property and a `filter` CSS property, respectively. Thus, those CSS properties generated by Domato are likely not functional.

CDV3: Attribute name values. The attributes of an element are referenced by their names for manipulating them. The validity of such an attribute reference depends on the owner element of the attribute, which is not fully presented by Domato's grammar. For example, an SVG `<animate>` element uses its `attributeName` attribute to indicate a particular attribute

of its parent element to be animated [29]. Being unaware of what element the <animate> element exactly serves, Domato randomly sets attributeName to any animatable SVG attribute, as described in Figure 5.2(c). For instance, Domato probably generates a worthless <animate> element that tries to change the non-existent x attribute of a <path> element.

CDV4: Attribute values. Similar to CSS selectors and property values, the value of a particular attribute (*e.g.*, form and usemap) involves a reference to an element of a specific type (*e.g.*, <form> and <map>), which is again not correctly described by Domato, as shown in Figure 5.2(d).

Generally, the dilemma of an existing DOM fuzzer based on a context-free grammar is that the grammar predefines a random approach to generate every possible unit of an HTML document but cannot anticipate the exact unit values eventually concretized in a document. Unfortunately, avoiding semantic errors during generation requires being aware of those concrete values. By contrast, RPG IR always memorizes the values that have been generated in its object-based context in the current document for generating new values afterwards.

5.2.2 Exploring Coverage-guided DOM Fuzzing

Most emerging fuzzers adopt a coverage-driven mutation-based approach, which is proven to be effective in practice [30, 31, 7]. Nevertheless, as of now, pure generation-based fuzzing with no runtime feedback is still the dominant approach for finding DOM engine bugs. Meanwhile, no public research aims to understand whether or not coverage-guided mutation-based fuzzing outperforms blackbox fuzzing against DOM engines. In other words, the optimal design of a DOM fuzzer is still an open problem.

However, existing DOM fuzzers cannot be directly utilized to determine the answer. Those fuzzers output final documents in plaintext for one-time testing. Due to the lack of the detailed context that originates from the use of static grammars, it is difficult to extend such fuzzers to comprehensively mutate the documents they generate. For instance, though the author of Domato implements an extension that enables mutation [32], the only supported

type of mutation is to append new data to an existing document. Meanwhile, all the other known mutation strategies, such as flipping and splicing existing data [30, 31] and merging two or more existing inputs [33, 28], are not feasible over plaintext. The extension is thus incapable of achieving the full potential of mutation. By contrast, FreeDom uses RPG IR to present a document with stateful structures. RPG IR carries detailed context information to ensure the semantic correctness of a document after all sorts of mutations.

In summary, building a mutation-based DOM fuzzer requires describing documents in mutable structures rather than in text. FreeDom achieves this by adopting RPG IR as its context-aware representation for HTML documents.

5.3 Design

In this section, we introduce the design of FreeDom based on RPG IR.

5.3.1 Document Representation in RPG IR

An HTML document consists of three different parts as mentioned in §5.1.1, which have distinct formats. Nevertheless, RPG IR still enables FreeDom to describe both the syntax and semantics of an HTML document comprehensively.

Document context. An HTML document in RPG IR contains two types of scopes. The global scope records all the elements contained in the initial DOM tree for reference. The state of an element includes its tag and attributes, which are referred to by many DOM API calls. Moreover, every JavaScript function creates a local scope, which contains DOM API calls. Each local scope contains the DOM objects locally defined by a particular API call. The recorded elements and DOM objects in the scopes serve as a basis for constructing four types of context-dependent values, summarized in §5.2.1, in a document.

CSS rules. FreeDom treats every CSS rule as an API call, which contains two types of arguments, including CSS selectors and a block of CSS properties. Both selectors and properties may contain references to an element and thus require importing objects from the

global scope.

Event handlers. An event handler contains a list of DOM API calls in JavaScript. As mentioned in §5.1.1, all the three types of DOM API calls involve an optional return object and zero or more arguments, which exactly fits the API model in RPG IR.

DOM tree. A DOM tree is an XML-based structure wherein each node represents an element in the document. To describe a DOM tree in RPG IR, FreeDom considers each element node as an API call. The arguments of such an API call are the attributes of the corresponding element. In addition, FreeDom supposes that a new scope is created by the API call, which contains the child nodes of the element. The outermost API call, namely the root of the DOM tree, is a `<body>` element.

5.3.2 Context-aware DOM Fuzzing

FreeDom systematically generate and mutate random HTML documents in RPG IR in a context-aware manner.

Document Generation

To generate a random input, FreeDom always starts with a blank document in RPG IR, which only has a `<body>` element, an empty `main` event handler, and a list of empty event handlers. Then, FreeDom uses various methods to construct the document content in the order of a DOM tree, CSS rules, and event handlers, which involves heavy queries and updates on the objects stored in the context.

DOM tree generation. Generating the DOM tree in a document has the highest priority because the tree determines the available nodes to be styled and manipulated by CSS rules and event handlers, respectively. In particular, FreeDom builds a DOM tree by repeatedly invoking the following three methods.

(1) **G_{t1}:** *Insert an element.* FreeDom creates a new element and inserts it as the child of an existing element in the tree. The index of the new element among all its

siblings is random. Depending on the type of parent, FreeDom randomly decides the corresponding element type of the new child by specification. FreeDom finally adds the element into the global context.

(2) $\mathbf{G_{t2}}$**:** *Append an attribute*. FreeDom creates a new attribute that is owned by an existing element yet is not set. FreeDom relies on the global scope to generate the initial value of the attribute, which is defined as an argument in RPG IR.

(3) $\mathbf{G_{t3}}$**:** *Insert a text node*. FreeDom selects an existing element that is allowed to have text content, generates a random string, and inserts the string into the tree as a child of the selected element.

CSS Rule Generation. After having the DOM tree in a document, FreeDom further generates a list CSS rules. The generation algorithm of a CSS rule, called $\mathbf{G_{c1}}$, repeatedly invokes two sub-routines.

(1) $\mathbf{G_{c2}}$**:** *Append a CSS selector*. FreeDom generates a CSS selector and adds it into the rule.

(2) $\mathbf{G_{c3}}$**:** *Append a CSS property*. FreeDom constructs a CSS property and appends it into the rule.

Both CSS selectors and properties are the arguments generated based on the objects contained in the global scope.

Event Handler Generation. FreeDom fills every event handler in a document with a sequence of DOM API calls. In the procedure of appending a random API call to a particular event handler (notated as $\mathbf{G_f}$), FreeDom first queries both global and local scopes for available DOM objects that can be used as the arguments of an API call in the current event handler. Then, FreeDom chooses a satisfiable DOM API (*i.e.*, the types of all the required arguments of such an API are supported by the context) defined by the specification and generates a corresponding API call based on the context. If the API call returns a new object, the object along with the location of its definition is recorded by the current local scope.

#	Before	After	Wgt.	New
G_{t1}	`<select></select>`	`<select><option></option></select>`	M	
G_{t2}		`<select size="3"></select>`	M	
M_{t1}	`<select size="3"></select>`	`<select size="0"></select>`	H	✓
M_{t2}		`<select autofocus=""></select>`	H	✓
G_{t3}	`<option></option>`	`<option>A</option>`	L	✓
M_{t3}	`<option>A</option>`	`<option>CCCCC</option>`	L	✓
G_{c1}	`.class1 {font-size:15px;}`	`.class1 {font-size:15px;} div {color:red;}`	M	
M_{c1}		`div {color:red;}`	M	✓
G_{c2}		`.class1, div {font-size:15px;}`	M	
M_{c2}		`div {font-size:  15px;}`	H	✓
G_{c3}		`.class1 {font-size:15px; color:red;}`	M	
M_{c3}		`.class1 {font-size:1vmin;}`	H	✓
G_{f}	`var v1 = window.getSelection();`	`var v1 = window.getSelection();` `var v2 = v1.getRangeAt(0);`	M	
M_{f1}		`document.createElement("div");` `var v1 = window.getSelection();`	M	✓
M_{f2}		`document.createElement("div");`	H	✓
M_{f3}	`var v2 = v1.getRangeAt(0);`	`var v2 = v1.getRangeAt(16);`	H	✓

Wgt.: Weight, **H**: High, **M**: Medium, **L**: Low

Table 5.2: The examples of the mutation algorithms used by FreeDom for three different parts of a document. The Wgt. column indicates the preference of FreeDom to those algorithms. We mark the algorithms that are beyond simple appending and difficult to support by extending old DOM fuzzers.

Single Document Mutation

FreeDom aims to mutate three different parts of an existing document in RPG IR with various granularities, while maintaining context information during mutation. We present the detailed mutation algorithms adopted by FreeDom as follows.

DOM tree mutation. FreeDom may call the generation routines, namely G_{t1}, G_{t2} and and G_{t3}, to grow the DOM tree in a document. In addition, FreeDom mutates existing nodes in the tree in three ways.

(1) **M_{t1}:** *Mutate an attribute value*. FreeDom selects an existing attribute and regenerate its value as an RPG IR argument based on the global context.

(2) **M_{t2}:** *Replace an attribute*. FreeDom first selects an element and randomly removes one of its attributes. Then, FreeDom applies G_{t2} to append a new attribute to the element. Here, FreeDom never removes an attribute whose value is referred to by other attribute values (*e.g.*, `attributeName` of SVG `<animate>`).

(3) **M_{t3}:** *Mutate a text node*. FreeDom simply selects a text node and regenerates its string content.

CSS rule mutation. First, FreeDom may directly invokes G_{c1}, G_{c2}, or G_{c3} to enlarge CSS rules in a document. Meanwhile, the existing CSS rules can be mutated from the following three aspects.

(1) **M_{c1}:** *Replace a CSS rule*. FreeDom removes an existing CSS rule from the document and inserts a new one generated by G_{c1}.

(2) **M_{c2}:** *Mutate a CSS selector*. FreeDom selects and mutates a selector in an existing rule.

(3) **M_{c3}:** *Mutate a CSS property*. FreeDom selects a CSS property, and similarly mutates its value.

Event handler mutation. Furthermore, FreeDom is also able to mutate event handlers in JavaScript. In particular, FreeDom first randomly selects a target event handler in the document. Note that `main` event handler has a much higher probability to be selected, as it is triggered most of the time. Besides appending a new API call (*i.e.*, G_f) to the target handler, FreeDom may also run the following three mutation methods.

(1) **M_{f1}:** *Insert an API call*. FreeDom first chooses a particular line of the JavaScript function as the insertion point. After that, FreeDom generates a new API call similar to how FreeDom does in G_f. The only difference is that when FreeDom queries the context for available DOM objects, all the elements in the global scope are still usable. Meanwhile, only the local objects defined before the insertion point can serve as the arguments of the API call. The generated API call is eventually placed at the chosen line. In addition, the birth location of the definition of every DOM object created below the line is incremented by one.

(2) **M_{f2}:** *Replace an API call*. FreeDom first selects a random line within the event handler. The original API call at this line is removed. Then, FreeDom generates a new API call and inserts it into the event handler at the line in the exact same way M_{f1} does. Note that FreeDom avoids removing any API call at a particular line that returns an object, because the object may be used in the later API calls and removing

such a call introduces reference errors.

(3) **M_{f3}:** *Mutate API arguments*. FreeDom first randomly selects an existing API call in the event handler and regenerates any of the arguments of the API call in a random way based on both global and local scopes.

Table 5.2 summarizes the document mutation algorithms supported by FreeDom with examples. FreeDom assigns each algorithm a specific weight for random sampling at runtime. We empirically set text-related mutations with low priority, as the exact text content is generally not crucial to trigger a crash. In general, FreeDom prefers to modify existing document content instead of adding new data to fully explore the states of existing DOM objects and avoid a rapid increase in testcase sizes.

Document Merging

Besides mutating a single document, FreeDom also supports merging two or more documents in RPG IR into a new document without breaking the document context due to the effectiveness of combining existing seed inputs for testing proven by [34, 35, 28]. Given two documents D_a and D_b, we present a random algorithm to merge D_b into D_a part by part.

Merging initial DOM trees. First, algorithm 2 presents how FreeDom makes D_a consume every node of the DOM tree in D_b, which starts from the direct child elements of D_b's tree root. For such an element n_b belonging to D_b, FreeDom randomly selects an element node n_t in D_a that has the same type (*i.e.*, tag) and the same or smaller tree depth. Next, FreeDom copies every missing attribute and all the text content from n_b into n_t. In addition, FreeDom uses an object map (*i.e.*, `ObjectMap` in algorithm 2) to record the mapping from n_b to n_t. The child elements of n_b are then recursively merged with the offspring of n_t in the same way. In this case, n_b no longer exists in the new DOM tree. Sometimes, an element of the same type as that to be merged with does not exist in D_a. Then, FreeDom directly inserts n_b along with its offspring into a random location in the DOM tree of D_a, which has the same tree depth as n_b. At the end, FreeDom records every element that originates from D_b and is directly

Algorithm 2: Merging two DOM trees in FreeDom.

Input: Two DOM trees T_a and T_b in two documents, an object map
Result: T_a being enlarged by merging with the nodes in T_b
`// ObjectMap: a global object map used throughout merging.`

```
1  Procedure mergeElement(n_a, n_b, ObjectMap)
2      TargetSet ← ∅;
3      for each n ∈ getOffsprings(n_a) do
4          if getType(n) = getType(n_b) then
5              TargetSet ← TargetSet ∪ {n};
6          end
7      end
8      if TargetSet = ∅ then
           // Move the sub-tree rooted at n_b to be a child of n_a.
9          insertChild(n_a, n_b);
10     else
11         n_t ∼ TargetSet; // Randomly sample a node from the set.
12         mergeAttributesAndText(n_t, n_b);
13         ObjectMap[n_b] ← n_t;
14         for each n ∈ getChildren(n_b) do
15             mergeElement(n_t, n, ObjectMap);
16         end
17     end
18 Procedure mergeTree(T_a, T_b, ObjectMap)
19     r_a ← getRoot(T_a); r_b ← getRoot(T_b); // The tree root is the <body>
           element.
20     for each n_b ∈ getChildren(r_b) do
21         mergeElement(r_a, n_b, ObjectMap);
22     end
23     for each n_b ∈ T_b − {r_b} do
24         if ¬∃ ObjectMap[n_b] then
25             addElementIntoGlobalContext(n_b);
26         end
27     end
```

inserted into D_a without merging in D_a's global context.

Merging CSS rules. Second, FreeDom directly copies the CSS rules from D_b into D_a, which does not involve any merging conflicts.

Merging event handlers. Next, FreeDom merges the event handlers in D_a and D_b. As every document in FreeDom is initialized with a fixed number of event handlers, FreeDom simply shuffles two paired event handlers F_a and F_b by inserting every API call from F_b

into a random line in F_a (see M_{f1} mentioned previously). Note that the relative order of any two API calls from F_b is not changed.

Fixing references. FreeDom finally uses `ObjectMap` to fix every reference in the new D_a that points to an element that originates from D_b but vanishes when merging the DOM trees.

FreeDom's merging algorithm ensures that the resulting document takes on the characteristics of the two input documents such as their DOM tree hierarchies and API call sequences while not introducing semantic errors.

5.4 Evaluation

In this section, we evaluate FreeDom to reveal the effectiveness of RPG IR in DOM fuzzing. In particular, we answer the following questions:

- **Q1.** How effective is FreeDom in discovering new bugs in the mainstream browsers? (§5.4.1)
- **Q2.** Does FreeDom become a state-of-the-art DOM fuzzer due to its RPG-IR-based approach? (§5.4.2)
- **Q3.** How effective is coverage-driven mutation in fuzzing DOM engines? (§5.4.3)

Experimental Setup. We run FreeDom and other DOM fuzzers in a small fuzzing cluster that consists of five 24-core servers running Ubuntu 18.04 with AMD Ryzen 9 3900X (3.8GHz) processors and 64GB memory. To display browser windows on a server without a graphical device, we leverage X virtual frame buffer (Xvfb), which is the most common solution in DOM fuzzing. Each fuzzer instance owns a separated Xvfb session for running its generated inputs.

5.4.1 Discovering New DOM Engine Bugs

We have intermittently run FreeDom for finding zero-day vulnerabilities in the DOM engines of all the mainstream browsers for two months. By default, FreeDom fuzzes the HTML, CSS, and SVG standards together since their corresponding implementations

#	Browser	Report ID	Component	Summary	Status
1	Safari 12.0.2	705074056	SVG	Use-after-free	CVE-2019-6212
2	Safari 12.1.0	709777313	WebGL	Arbitrary memory access	CVE-2019-8596
3†	Safari 12.1.0	-	SVG	Heap overflow	CVE-2019-8609
4	Safari 13.0.1	710042930	SVG	Heap overflow	CVE-2019-8720
5	Safari 13.0.5	727800575	SVG	Null dereference	Patched
6	Safari 13.0.5	729340941	SVG	Race condition	Patched
7	Safari 13.0.5	729379682	SVG/CSS	Use-after-free	Patched
8	Safari 13.0.5	729429465	SVG	Use-after-free	CVE-2020-9803
9	Safari 13.0.5	-	CSS	Null dereference	Patched
10*	Safari 13.0.5	-	HTML/SVG	Use-after-free	Patched
11	Safari 13.0.5	730447379	HTML/SVG/CSS	Use-after-free	CVE-2020-9806
12	Safari 13.1.0	732608208	HTML/SVG/CSS	Use-after-free	CVE-2020-9807
13	Safari 13.1.0	734414767	HTML/CSS	Use-after-free	CVE-2020-9895
14‡	WebKitGTK 2.24.0	-	WebGL	Out-of-bound memory access	Patched
15‡ *	WebKitGTK 2.28.0	731291111	HTML	Use-after-free	Patched
16†	Chrome 73.0	943087	WebGL2	Integer overflow	CVE-2019-5806
17†	Chrome 73.0	943709	WebGL2	Heap overflow	CVE-2019-5817
18†	Chrome 74.0 (beta)	943424	WebGL2	Use-after-free	Patched
19†	Chrome 74.0 (beta)	943538	WebGL2	Use-after-free	Patched
20	Firefox 76.0	1625051	HTML/CSS	Out-of-bound access	Patched
21	Firefox 76.0	1625187	HTML/CSS	Rust assertion	Acknowledged
22	Firefox 76.0	1625252	HTML/SVG/CSS	Null dereference	Acknowledged
23	Firefox 76.0	1625369	HTML	Correctness issue	Patched
24	Firefox 76.0	1626152	SVG/CSS	Use-after-free	Patched

† The bugs which earn bug bounty rewards.
‡ The WebKit bugs that only affect WebKitGTK builds on Linux and do not affect Safari on macOS.
* The duplicated bugs which are also reported from internal efforts or other external researchers.

Table 5.3: The reported bugs found by FreeDom in Apple Safari (WebKit), Google Chrome, and Mozilla Firefox. We mark out the latest browser versions that are affected by the bugs. The **Component** column indicates what specific DOM components a document is required to contain for triggering the bugs. In particular, bugs #18, #19, and #24 only affect the beta version and are never shipped into a release; though they are security bugs, there are no CVEs assigned for them.

in a browser are closely related. FreeDom separately fuzzes WebGL on different OS platforms, whose implementation is independent of other browser components but involves platform-dependent code. Table 5.3 lists a total of 24 bugs found by FreeDom that have been confirmed by the browser vendors, including 14 bugs in Safari/WebKit, four bugs in Chrome, and five bugs in Firefox. Besides a few assertions, null dereferences, and correctness issues, the vast majority of the bugs are security-critical, which have helped us gain 10 CVEs and 65K USD bug bounty rewards so far. The fuzzing results reflect that FreeDom is effective in discovering new bugs in the latest DOM engines.

5.4.2 Effectiveness of Context-aware Fuzzing

To prove that FreeDom is state of the art with its context-aware fuzzing approach driven by RPG IR, we compare the fuzzing performance of FreeDom with that of Dharma [2] and Domato [1]. In this evaluation, we always run 100 instances of every fuzzer in study on five machines against an ASan build of WebKitGTK 2.28.0 on Linux for 24 hours. According to public record [1], Domato has found the most bugs in WebKit, which is thus selected as our evaluation target. To retrieve the code coverage of a document generated during the experiment, we re-run it with an instrumented WebKit that profiles the visited basic blocks of the DOM engine part (*i.e.*, `Source/WebCore/`).

Comparison with Dharma

We first evaluate FreeDom with Dharma, a generation-based fuzzer based on context-free grammars. Dharma officially only provides the grammar file for SVG documents with no support of HTML tags, CSS rules, and DOM APIs. For a fair comparison, we use the original Dharma and modify FreeDom to only generate the initial DOM tree with SVG tags and attributes and skip the other two parts in a document. We also configure the number of <svg> elements, SVG nodes rooted in an <svg> element, and attributes of a node in a document output by FreeDom to ensure that both fuzzers generate the inputs of similar size and complexity. Both fuzzers execute a document for at most 5 seconds. The experiment is repeated for three times.

Figure 5.3 presents the experimental result. FreeDom visits 13.96% more code blocks than Dharma on average. More importantly, FreeDom stably triggers at least 7 unique crashes during each fuzzing run and totally discovers 18 unique ones. Meanwhile, Dharma fails to find any crashes during the experiment. There is a higher chance that Dharma may trigger a few of FreeDom's crashes by adding more random options for generating certain constant values like integers and images into its grammar. Nevertheless, we observe that around 60% of the crashes triggered by FreeDom involve SVG animations. Similar

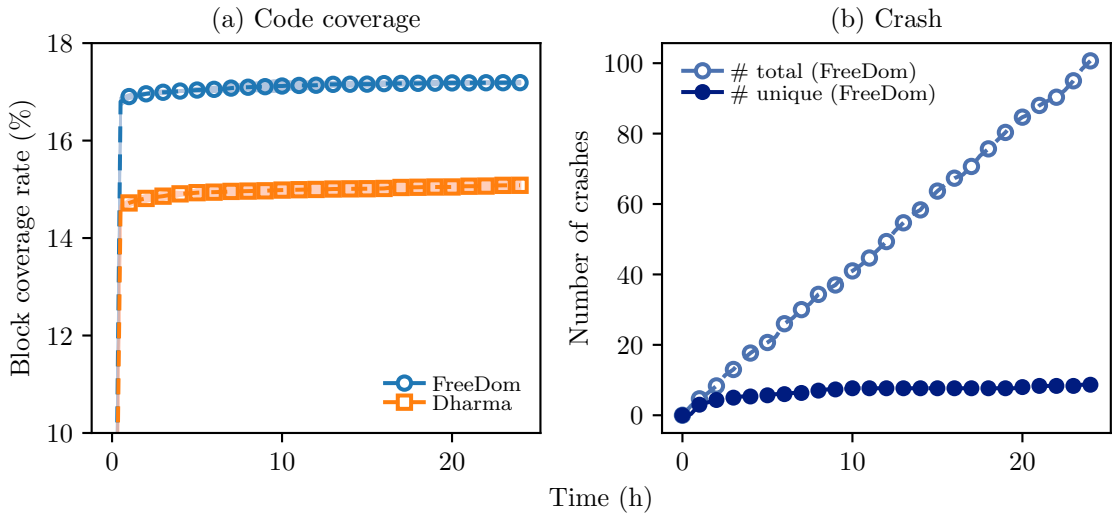


Figure 5.3: Achieved code coverage and triggered crashes of FreeDom and Dharma when fuzzing SVG documents in WebKit with 100 cores for 24 hours. Dharma fails to find any crash during three fuzzing runs. In (b), we differentiate unique crashes found by FreeDom based on their crashing PC values.

to Domato discussed in §5.2.1, Dharma, based on its context-free grammar, is not aware of the exact element whose attributes are to be animated and simply animates an attribute that is randomly selected from 30 candidates that are neither all animatable nor always owned by the element. Therefore, Dharma rarely constructs a valid SVG animation. By contrast, FreeDom is more likely to generate working animations and manages to trigger those crashes, which preliminarily reflects the effectiveness of context-aware generation in DOM fuzzing.

Comparison with Domato

We then evaluate FreeDom and Domato, both of which fuzz the HTML, SVG, and CSS specifications together by default. Although the two fuzzers have completely different designs, we do not introduce any change to Domato's fuzzing engine and take great effort to configure FreeDom to generate the documents that have complexity similar to those generated by Domato. In addition, both fuzzers run with a 5-second timeout and are evaluated three times.

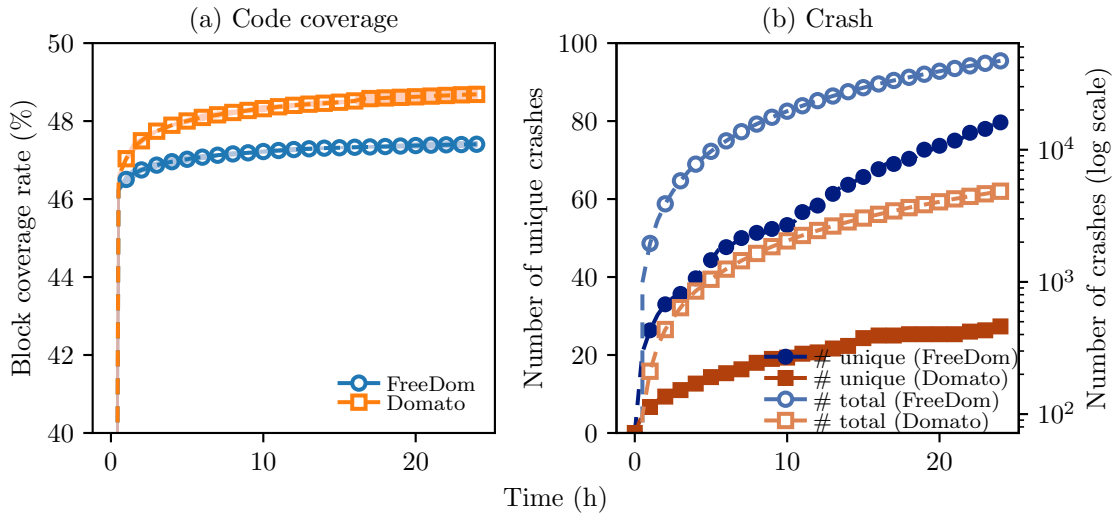


Figure 5.4: Achieved code coverage and triggered crashes of FreeDom and Domato for a 24-hour run with 100 cores. Note that we use a log scale on the right side to present the total number of crashes. We differentiate unique crashes based on their crashing `PC` values.

Figure 5.4 presents the evaluation results. With nearly 3% more executions, the overall code coverage of Domato is slightly higher than that of FreeDom by 2.69% on average. Nevertheless, FreeDom triggers around 9.7× more crashes and 3× more unique ones than Domato. In particular, FreeDom discovers 112 unique crashes in three runs, 11 of which have explicit security implications reported by ASan (*i.e.*, heap buffer overflow and use-after-free bugs rather than null pointer dereferences and infinite recursions). By contrast, Domato only finds a total of 39 unique crashes, three of which are security-related. More importantly, 34 (87%) crashes found by Domato are also triggered by FreeDom. The evaluation results indicate that the ability of FreeDom to find bugs in the latest DOM engine largely surpasses that of the state-of-the-art DOM fuzzer.

To further understand how context awareness enables FreeDom to outperform Domato, we minimize the inputs of 117 unique crashes found by both fuzzers into PoCs with the HTML minimizer provided by ClusterFuzz. We then determine what types of data dependencies (see §5.2.1) every PoC file involves through manual inspection, which is presented in Table 5.4. Among the PoCs of 39 crashes found by Domato, a majority of them do not contain any context-dependent part. Around 25% of them have context-dependent

	#Crashes	Data Dependence				
		None	CDV_1	CDV_2	CDV_3	CDV_4
Domato	5	4 (80.00%)	1 (20.00%)	N/A	N/A	N/A
FreeDom	78	21 (26.92%)	49 (62.82%)	20 (25.64%)	8 (10.25%)	14 (17.94%)
Both	34	20 (58.82%)	9 (26.47%)	N/A	N/A	1 (2.94%)

Table 5.4: The unique crashes discovered by Domato, FreeDom, and both fuzzers during three 24-hour runs. We also count the number of crashes that have one of the four types of context-dependent values (CDVs) described in §5.2.1. Note that some crashes that involve more than one type of CDVs are counted more than once.

CSS selectors (*i.e.*, CDV_1), which Domato has certain chances to construct correctly through a fixed number of predefined elements, classes, and tags. At most times, a minimized PoC generated by Domato only remains the universal selector (*i.e.*, *), which is context-free. Meanwhile, the context-dependent selectors generated by FreeDom are much more likely to be valid and thus FreeDom manages to find another 49 crashes that require specific live elements to be styled. Furthermore, during nearly 5 million executions in total, Domato fails to trigger any crash but one that involves any of the other three types of data dependencies, which are largely not addressed by its static grammar. Contrary to Domato, FreeDom manages to generate a number of PoCs that cover from CDV_1 to CDV_4.

In a nutshell, our in-depth analysis shows that (1) the RPG-IR-based approach adopted by FreeDom still manages to find the old types of bugs that existing DOM fuzzers target by their context-free grammars, and (2) the additional context information maintained by FreeDom in RPG IR is effective in finding many more bugs in the latest DOM engines that have not been explored. Therefore, we believe FreeDom's IR-based design fundamentally outperforms the state-of-the-art DOM fuzzer, Domato.

5.4.3 Effectiveness of Coverage-driven Fuzzing

We now evaluate FreeDom to understand the effectiveness of coverage guidance in DOM fuzzing. Similarly, we launch 100 FreeDom instances to fuzz the optimized build of WebKitGTK 2.28.0 with the mutation strategies described in §5.3.2 on 100 cores for 24 hours. The DOM engine part of WebKit is instrumented for block coverage measurement.

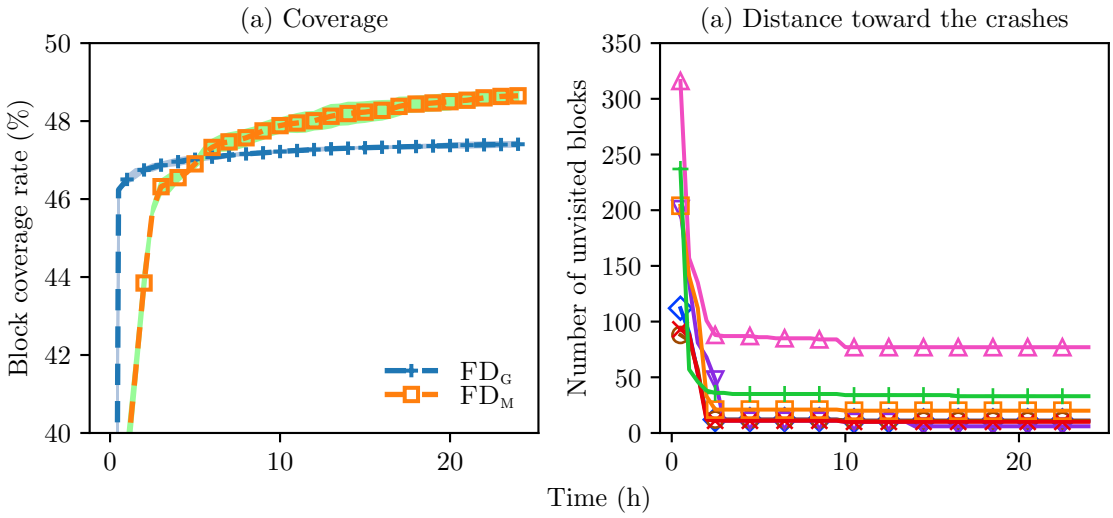


Figure 5.5: (a) Average block coverage rate of running FreeDom with random generation and coverage-guided mutation to fuzz WebKit for 24 hours; (b) Number of basic blocks covered by the PoCs of seven security-related crashes that are not visited by coverage-guided mutation at different times during a 24-hour fuzzing process. FD_G and FD_M represent FreeDom in the generation mode and mutation mode, respectively.

In this experiment, FreeDom bootstraps with two simple seeds: a blank document and a document with a single `<svg>` element under `<body>`. Table 5.5 presents the fuzzing results, and the average coverage growth is illustrated by Figure 5.5(a). We interpret the evaluation results as follows.

Effectiveness of context-aware mutation. Compared to generation-based fuzzing by FreeDom, coverage-guided mutation helps to visit around 1.2% and 2.62% more code blocks, respectively. More importantly, mutation-based fuzzing enables FreeDom to successfully discover three new crashes, including two security-related ones that are never triggered by FreeDom through pure generation. A generation-based fuzzer intends to generate a large-size document that contains a lot of randomly chosen elements, attributes, and CSS rules and hopefully covers various DOM features within one execution. Meanwhile, mutation-based fuzzing focuses on repetitive mutations and gradual growth of its inputs, which is more likely to trigger the crashes that require strict or subtle settings. For example, one crash missed by FreeDom when performing generation-based fuzzing is triggered by an SVG `<text>` element that has a single attribute `x="8192em"`. The element is required to have no

Fuzzer	Execs/s	Coverage rate	#Crash	#Unique	#Security	#New
$\mathbf{FD_G}$	18.17	47.40(±0.05)%	47058.67	79.67	11	-
$\mathbf{FD_M}$	63.94	48.64(±0.13)%	331.67	21	6	3

Table 5.5: Fuzzing results of running FreeDom with random generation (FD_G) and coverage-guided mutation (FD_M) against WebKit for 24 hours. We list the total number of unique security-related crashes found in three fuzzing runs and the average values for other metrics. The New column presents the number of distinct crashes that are only found by the coverage-guided mutation-based fuzzing.

surrounding elements and no additional attributes or CSS styles that affect its text position, which is difficult to find in a document output by random generation that has a deep element tree, 10 attributes for an element, and 50 CSS rules. The PoC of another new crash has three sibling `<set>` elements to animate the same attribute of their parent. FreeDom in the generation mode selects child elements and a parent attribute to be animated uniformly from various available candidates and therefore rarely generates such a document from a statistical perspective. By contrast, FreeDom's fine-grained mutation strategies manages to grow a blank document into the inputs of these new crashes step by step.

Limitation of coverage-guided mutation. Unfortunately, we witness the weak implication of code coverage for finding bugs when comparing the fuzzing results of random generation and coverage-guided mutation enabled by FreeDom. With 3.5× more executions and a 2.62% coverage improvement, mutation-based fuzzing finds nearly 3.8× fewer unique crashes on average compared to random generation. Around 75% unique crashes are missed by FreeDom in this experiment, including seven security-related ones previously found during three 24-hour runs. To further study the failure of coverage-guided fuzzing, we determine the minimal basic blocks required to be covered for triggering the seven crashes and observe how FreeDom approaches the crashes (*i.e.*, covers those code blocks) during this fuzzing run. Figure 5.5(b) presents the result, which shows that FreeDom is extremely close to most of the crashes after four hours. However, due to an increasing number of interesting testcases waiting to be mutated, FreeDom in the mutation mode tries to expand its coverage without any particular direction and thus fails to make a final push to trigger any

of the crashes in the remaining 20 hours. Though FreeDom in the generation mode blindly generates the documents that can never explore the browser code thoroughly in process of time, at least certain deep code paths are consistently tested through every execution due to the fact that a generated document has a large size and rich semantics. By contrast, code coverage makes FreeDom in the mutation mode that starts with a blank document wander around numerous shallow code paths and cannot move downward for a long time because of the extreme complexity of a DOM engine.

In general, blackbox generation is not comprehensive but is still recommended for discovering a vast number of bugs in a DOM engine in a reasonable time. Meanwhile, coverage-driven mutation is also considered an irreplaceable approach, especially for finding the bugs that occur with exacting conditions. One more advantage of mutation-based fuzzing is that minimizing its crashing documents of much smaller sizes is less time consuming. We also believe that the performance can be largely improved with more computing resources and better seeding inputs.

5.5 Conclusion

To prove that RPG IR can address the syntax and semantics of DOM APIs for fuzzing, we propose FreeDom, a context-aware DOM fuzzer that relies on RPG IR to describe both structures and context information of a document so as to avoid semantic errors. We have reported 24 bugs found by FreeDom in mainstream web browsers with 10 CVEs assigned. Due to its context-awareness, FreeDom finds 3$\times$ more unique crashes in WebKit than the state-of-the-art DOM fuzzer, Domato. In addition, RPG IR enables FreeDom to perform context-aware coverage-guided mutation, which is more effective in visiting new code blocks (2.62%) and finds three complex bugs that the generative approach fails to find.

CHAPTER 6

JANUS: CONTEXT-AWARE FILE SYSTEM FUZZING VIA RPG IR

To further show the effectiveness of RPG IR in API fuzzing, we present Janus, a context-aware file system fuzzer that models Linux system calls in the way RPG IR suggests and has discovered numerous unknown bugs in the Linux kernel.

6.1 File System Fuzzing

File systems are one of the most basic system services of an operating system that play an important role in managing the files of users and tolerating system crashes without losing data consistency. Currently, most of the conventional file systems, such as ext4 [36], XFS [37], Btrfs [38], and F2FS [39], run in the OS kernel. Hence, bugs in file systems cause devastating errors, such as system reboots, OS deadlock, and unrecoverable errors of the whole file system image. In addition, they also pose severe security threats. For instance, attackers exploit various file system issues by mounting a crafted disk image [40] or invoking vulnerable file system-specific operations [41] to achieve code execution or privilege escalation on victim machines. However, manually eliminating every bug in a file system that has sheer complexity is a challenge, even for an expert. At the same time, many widely used file systems are still under active development. File system developers consistently optimize performance [42] and add new features [43, 44], meanwhile introducing new bugs [45, 46, 47]. Therefore, we consider fuzzing as a viable approach to automatically discover bugs in a wide range of file systems, which only requires minimal knowledge about the target software. Generally, a disk file system has two-dimensional input space for a fuzzer to explore: (1) the structured *file system image*; and (2) *file operations* that users invoke to access files stored on a mounted image.

6.1.1 Disk Image Fuzzing

A disk image is a *large structured* binary blob. The blob has (1) user data and (2) several management structures, called *metadata*, that a file system needs to access, load, recover, and search data or to fulfill other specific requirements of a file system. The size of metadata constitutes merely 1% of the image size [48]. Meanwhile, the minimal size of a valid image can be 100 MBs. Coverage-driven mutation proves to be one of the most effective approaches in fuzzing structured binary data, not excepting disk images. Nevertheless, blindly mutating an entire disk image introduces two issues: (1) Large input size leads to an exponential increase in the input space exploration. Meanwhile, important metadata are mutated infrequently. (2) To detect metadata corruption, several file systems (*e.g.*, XFS v5, GFS, F2FS, etc.) introduce checksums to protect on-disk metadata. Hence, the kernel rejects a corrupt image, with mutated metadata blocks without correct checksums, during initialization. To overcome the issues, an image fuzzer should effectively fuzz a complicated, large disk image by (1) mutating scattered metadata in the image that are probably hardened by checksum, and (2) mitigating frequent disk I/O due to input manipulation.

6.1.2 File Operation Fuzzer

As part of the OS, a kernel file system are considered a API-based system, where users can invoke a set of system calls, namely the file system APIs, to operate the files in the system. Although researchers have proposed a number of system call fuzzers [7, 8, 49] and porting these fuzzers to target file system operations is straightforward, existing fuzzers fail to effectively fuzz file operations due to their context unawareness. First, file operations modify only file objects (*e.g.*, directories, symbolic links, etc.) that exist on the image, and a completed operation affects particular objects. However, existing OS fuzzers do not consider the dynamic dependence between an image and file operations. They blindly generate system calls without context awareness, which explore a file system superficially. For example, the state-of-the-art OS fuzzer, Syzkaller, generates system calls based upon

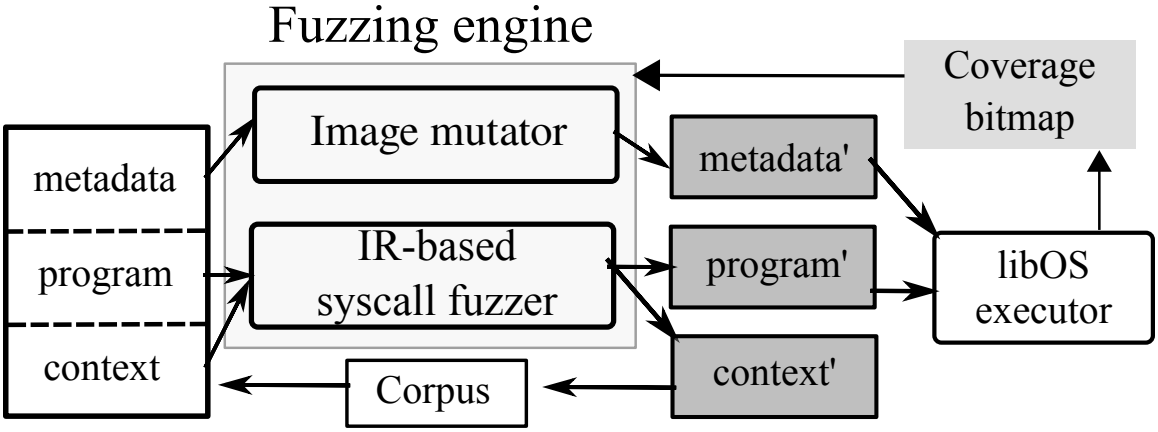


Figure 6.1: An overview of Janus.

static grammar rules describing the data types of every argument and return value for every target system call. Therefore, Syzkaller is able to generate a single semantically correct system call but fails to explore the collective behavior of a set of system calls and the modified file system image. For instance, Syzkaller may use a file descriptor that has been closed by `open()`. In addition, Syzkaller may emit multiples of `open()` calls on a file with its old path that has been either renamed (`rename()`) or removed (`unlink()`).

6.2 Design

6.2.1 Overview

Janus is a feedback-driven fuzzer that mutates the metadata of a seed image, while generating context-aware file operations (*i.e.*, system calls) modeled by RPG IR to comprehensively explore a file system. More specifically, Janus merely stores the metadata extracted from the seed image as its mutation target, which is critical for a file system to manage user data. In addition, Janus re-calculates every metadata checksum value after mutation. Since the metadata occupy a small space (1%), the size of an input test case is much smaller than that of an entire disk image, which enables high fuzzing throughput. More importantly, unlike Syzkaller, Janus does not rely on manually specified information about the files stored on a seed image, as it becomes stale over time and results in generating ineffective file

operations. Instead, Janus considers system call effects and generates new file operations based upon the deduced status of the file objects on an image after completing old operations in a workload. Moreover, Janus manages to explore the two-dimensional input space of a file system by wisely scheduling image fuzzing and file operation fuzzing. To improve the fuzzing throughput, Janus relies on a library OS to test kernel functions in user space. A library OS instance with file system support runs as a user application, which can be re-launched with negligible overhead, and also helps to increase the chance of reproducing a found bug.

Figure 6.1 presents the detailed design of Janus. An input for Janus consists of three parts: (1) a binary blob comprising the metadata blocks of a seed image, (2) a serialized program (*i.e.*, file system workload) that describes a sequence of system calls in a context-aware manner, and (3) a program context, namely the speculated status of the file objects on the image after the program operates the image. In the beginning, Janus relies on a file system-specific parser to extract metadata from a seed image. Janus also inspects the seed image to retrieve initial image status as the starting context of an empty system call program. Janus initiates fuzzing with both the image mutator and the system call fuzzer by selecting a test case from the corpus for exploring the two-dimensional input space in an infinite loop. First, the fuzzing engine invokes the image mutator to flip the bytes of the metadata blob in several ways and outputs mutated blobs. At the same time, the program in the test case remains unchanged. Later on, the system call fuzzer based on RPG IR enables Janus to either mutate the argument values of existing system calls in the program or append new ones to the program. The system call fuzzer also produces new image status according to the newly generated program. Meanwhile, the metadata part remains intact. The output metadata is combined with other unchanged parts (*i.e.*, user data) to produce a full-size image with all the checksum values re-calculated by Janus. And the output program is also serialized and saved onto the disk. A user-space system call executor, which relies on a library OS, launches a new instance to mount the full-size image and perform the

file operations involved in the program loaded from the disk. The runtime path coverage of the executor is profiled into a bitmap shared with Janus's fuzzing engine. The fuzzing engine inspects the bitmap; on discovering a new path, Janus saves the shrunken image, the serialized program, and the speculated file object states into one binary input for further mutation in successive runs. Note that for each test case, Janus always launches the image mutator first for certain rounds and invokes the system call fuzzer if no interesting test case is discovered.

6.2.2 Building Corpus

Janus relies on its image parser and system call fuzzer to build its initial corpus upon a seed image. The first part of the test cases in the corpus is the essential metadata blocks of the seed image, which constitutes around 1% of the total size, thereby overcoming the challenges of fuzzing a disk image, as described in §6.1. Specifically, Janus first maps the entire image into a memory buffer. Then a file system-specific image parser scans the image and locates all the on-disk metadata according to the specification of the applied file system. Janus reassembles these metadata into a shrunken blob for mutation afterward and records their sizes and in-image offsets. For any metadata structure protected by checksum, Janus records the in-metadata offset of the checksum field recognized by the image parser. Second, the starting test cases also include the information of every file and directory on the image that allows Janus to use that knowledge for generating context-aware workloads afterward. In particular, Janus probes the seed image and retrieves the path, type and extended attributes of every file object on it, which forms the starting context of an empty system call program.

6.2.3 Fuzzing Images

Janus relies on the image mutator to fuzz images. In particular, the image mutator loads the metadata blocks of a test case, and applies several common fuzzing strategies [30] (*e.g.*, bit flipping, arithmetic operation on random bytes, etc.) to randomly mutate the bytes of

the metadata. Similar to existing fuzzers [50], Janus prefers a group of specific integers such as `-1`, `0`, `INT_MAX`, etc., instead of purely random values to mutate the metadata. In our evaluation, these special values enable the image mutator to produce more corner cases, which are not correctly handled by the file system (*e.g.*, bug #1, #6, #14, #28, #33, etc. in Table 6.2 found by Janus) and also more extreme cases that increase the probability of crashing the kernel by triggering a specific bug at runtime (*e.g.*, most of the out-of-bound access bugs discovered by Janus).

After mutating the entire metadata blob, Janus copies each metadata block in the blob back to its corresponding position inside the memory buffer, which stores the original full-size image. To maintain the sanctity of the image, the image parser recalculates the checksum value of every metadata block by following the specific algorithm adopted by the target file system, and fills the value at the recorded offset of the checksum field.

6.2.4 File Operations in RPG IR

Generally, a testcase of Janus in RPG IR consists of a sequence of system calls and a context that stores the file objects being operated by the system calls.

Program context. A system call program in RPG IR only involves one global scope, which stores all the file objects in the file system. For every file object, Janus records its path, type (*e.g.*, normal file, directory, symbolic link, FIFO file, etc.), and extended attributes as its object state. Those file information is used and meanwhile, updated by different system calls. The global scope also contains a list of active file descriptors that are opened by `open()` calls and have not been closed by the program.

System calls. Each system call is a tuple of a syscall number, a list of arguments and a return value, which perfectly fits the API model depicted by RPG IR. A system call can have context-free arguments such as flags and operation modes and meanwhile, context-dependent arguments including file descriptors, file paths and extended attributes, which are either object references or property values of the file objects stored in the context.

6.2.5 File Operation Generation and Mutation

The system call fuzzer generates new programs from an input program in RPG IR in two ways: (1) Syscall mutation. The system call fuzzer randomly selects one system call in the program, and mutates the value of a randomly selected argument; (2) Syscall generation. The system call fuzzer appends a new system call to the program, whose arguments have randomly generated values. In particular, Janus adopts the same strategies that Syzkaller uses to generate values for the trivial arguments of a system call. The candidate values of these arguments are independent of the program context. For any argument that has a clearly defined set of its available values, Janus randomly selects values from the set for it. (*e.g.*, `int whence` for `lseek()`). Moreover, Janus generates random numbers in a certain range for the arguments of an integer type (*e.g.*, `size_t count` for `write()`). Furthermore, a number of file operations requires an argument of a pointer type. Such a pointer normally points to a buffer that is used to store either user data (*e.g.*, `void *buf` for `write()`) or kernel output (*e.g.*, `void *buf` for `read()`). For the former case, the system call fuzzer declares an array object filled with random values for the argument. A fixed array object is always used in the latter case, since Janus is not driven by what the kernel outputs at runtime except for its code coverage.

Nevertheless, for those non-trivial arguments whose proper values depend on the file objects in a file system, Janus generates their values based not only on their expected types, but more importantly, on the file information maintained in the global context by following mainly three rules: (1) If a file descriptor is required, the system call fuzzer randomly picks an opened file descriptor of proper type. For instance, `write()` requires a normal file descriptor, while `getdents()` asks for the file descriptor of a directory; (2) If a path is required, the system call fuzzer randomly selects the path of an existing file or directory, or a stale file or directory that is removed by recent operations. For instance, Janus provides the path of a normal file or a directory to `rename()`, but delivers only that of a valid directory required by `rmdir()`. If the path is used to create a new file or directory, Janus may also

randomly generate a brand new path that is located under an existing directory; (3) If a system call operates the existing extended attribute of a particular file (*e.g.*, `getxattr()` and `setxattr()`), the system call fuzzer randomly picks a recorded extended attribute name of the file. The generation strategies enable Janus to emit context-aware workloads on fresh file objects that are free of runtime errors and achieve high code coverage.

For a newly generated system call, Janus appends it to the program, summarizes the potential changes to the file system caused by the system call (*i.e.*, the API effects) and updates the states of the involved objects correspondingly. Note that the three types of API effects defined by RPG IR (see §3.3.3) fully address the semantics of file operations. For instance, `open()`, `mkdir()`, `link()`, or `symlink()` may create a new file or directory, while `open()` also introduces an active file descriptor; `rmdir()` or `unlink()` removes a file or a directory from the image; `rename()` updates the path of a file and `setxattr()` or `removexattr()` updates a particular extended attribute.

Note that in the current design, Janus maintains only the speculated status of the file objects after completing the execution of a program. Therefore, Janus avoids any mutation on the existing arguments that result in potential changes to the context. For instance, Janus may mutate `fd` of a `write()` in the program while never touching `path` of `unlink()`, since such a mutation may invalidate the system calls after the mutated ones (*e.g.*, changing `unlink("A")` to `unlink("B")` affect all the existing file operations afterward on file `B` in a test case).

6.2.6 Exploring Two-Dimensional Input Space

To fuzz both metadata and system calls together, Janus schedules its two core fuzzers in order. Specifically, for an input test case, which contains a shrunken image and a program, Janus first launches the image mutator to mutate random bytes on the shrunken image. If no new code path is discovered with the unchanged program, Janus invokes the system call fuzzer to mutate the argument values of an existing system call in the program for certain

rounds. If still no new code path is explored, Janus eventually tries to append new system calls to the program. Note that rounds in every fuzzing stage are user defined.

Scheduling image fuzzing and file operation fuzzing in such an order is effective as follows: (1) The extracted metadata indicate the initial state of an image, whose impacts on the executions of file operations gradually decreases when the image has been operated by several system calls. Hence, Janus always tries to mutate metadata first. (2) Introducing new file operations exponentially increases the mutation space of a program and may also erase the changes from past operations of the image. Therefore, Janus prefers mutating existing system calls rather than generating new ones.

6.2.7 Library OS based Kernel Execution Environment

To avoid using an aging OS or file system that results in unstable executions and irreproducible bugs, Janus relies on a library OS based application (*i.e.*, executor) to fuzz OS functionalities. Specifically, Janus forks a new instance of the executor to test every newly generated image and workload from the fuzzing engine. Note that forking a user application incurs negligible time compared with resetting a VM instance. Hence, Janus guarantees a clean-slate kernel for every test case with low overhead. Moreover, as both fuzzing engine and executor run in user space on one machine, sharing input files and coverage bitmap between them is straightforward, which is challenging for VM-based fuzzers that run the fuzzing engine outside VM instances. In addition, a library OS instance requires far less computing resources compared with any type of VMs. Therefore, we can deploy Janus instances on a large scale without severe contention.

6.3 Evaluation

In this section, we evaluate the effectiveness of Janus in terms of its ability to find bugs in the latest file systems and achieve higher code coverage than existing file system fuzzers. In particular, we answer the following questions:

- **Q1:** How effective is Janus in discovering previously unknown bugs in file systems? (§6.3.1)
- **Q2:** How effective is Janus in exploring (1) file operations and (2) the two-dimensional input space including images and file operations by adopting a context-aware approach based on RPG IR? (§6.3.2 and §6.3.3)

Experimental Setup. We evaluate Janus on a 2-socket, 24-core machine running Ubuntu 16.04 with Intel Xeon E5-2670 processors and 256GB memory. We use Janus to fuzz file systems in Linux v4.18-rc1, unless otherwise stated. In particular, we test eight file systems including ext4, XFS, Btrfs, F2FS, GFS2, HFS+, ReiserFS, and VFAT. We create a seed image for each file system with most features enabled except ext4 and XFS. For ext4, we create two seed images: one compatible with `ext2/3` and the other with ext4 features. Similarly, we do the same for XFS representing XFS v4 and XFS v5, which introduces on-disk checksums to enforce metadata integrity. In total, we evaluate 10 seed images. In addition, we compare our results with Syzkaller, which is the state-of-the-art OS fuzzer. We run Syzkaller with KVM instances, each of which has two cores and 2GB of memory.

Note that Syzkaller relies on KCOV to profile code coverage, while Janus relies on the method of AFL. For an apples-to-apples comparison between Syzkaller and Janus, after fuzzing 12 hours, we mount every image mutated by Janus, and execute the corresponding program generated by Janus on a KCOV-enabled kernel to get the KCOV-style coverage.

6.3.1 Bug Discovery in the Upstream File Systems

We intermittently ran Janus for four months (*i.e.*, from April 2018 to July 2018) to fuzz the aforementioned file systems in upstream kernels from v4.16 to v4.18. Over the span of few days to a week, we ran three instances of Janus to test each file system. Janus found 90 unique bugs that resulted in kernel panics or deadlocks, which we reported to the Linux kernel community. We differentiated bugs on the basis of KASAN reports and call stack traces. Among them, developers confirmed 62 as previously unknown bugs, including 36 in

File Systems	#Reported	#Confirmed	#Fixed	#Patches	#CVEs
ext4	18	16	16	20	13
XFS	17	11	7	9	5
Btrfs	9	9	8	10	5
F2FS	11	11	11	12	8
GFS2	14	0	0	0	0
HFS+	8	7	1	1	1
ReiserFS	13	8	0	0	0
VFAT	0	0	0	0	0
Total	90	62	43	52	32

Table 6.1: An overview of bugs found by Janus in eight widely-used file systems in upstream Linux kernels. The column **#Reported** shows the number of bugs reported to the Linux kernel community; **#Confirmed** presents the number of reported bugs that are previously unknown and confirmed by kernel developers; **#Fixed** indicates the number of bugs that have already been fixed, at least in the development branch, and **#Patches** reports the number of git commits for fixing found bugs; **#CVEs** lists the number of CVEs assigned for confirmed bugs.

ext4, XFS, and Btrfs—the three most widely used file systems on Linux. 32 bugs have been assigned with CVEs (see Table 6.1). Another important finding is that some bugs, (*e.g.*, four bugs related to log recovery in XFS and six bugs about extended attributes in HFS+) are not going to be fixed by developers in the near future, as these bugs require large-scale code refactoring. In addition, ReiserFS developers will not fix five bugs that lead to the `BUG()` condition, as ReiserFS is in maintenance mode.

Note that there are other notable efforts on finding file system bugs through fuzzing or manual auditing.

- Syzkaller, the state-of-the-art system call fuzzer that started to support mutating file system images in March, 2018. Note that Google deployed many more instances of Syzkaller (*i.e.*, syzbot) than those of Janus for continuously fuzzing the upstream kernel. Although syzbot fuzzes the whole kernel, we found more file system bugs with Janus in four months. According to our investigation, Syzkaller reported only two ext4 bugs, one XFS bug, four F2FS bugs, and one HFS+ bug during our evaluation period, among which one of the ext4 bugs, the XFS bug, and the HFS+ bug were also found by Janus. Janus missed one ext4 bug requiring a 4K block size, which is larger

than that of our seed images. And we started using Janus to fuzz F2FS after these four F2FS bugs were fixed.

- Google Project Zero, a team of security researchers seeking zero-day vulnerabilities who found one ext4 bug through source review. The bug was also discovered by Janus.
- Internal efforts from the file system development community. XFS developers noticed four XFS bugs found by Janus before we reported them. Unfortunately, we were unable to provide the total number of memory safety bugs found by developers whose patches cannot easily be differentiated from the ones for fixing functionalities.

6.3.2 Exploring File Operations

We now evaluate the effectiveness of Janus in only fuzzing file operations without mutating the file system image, *i.e.*, we discard the image fuzzing stage. We compare Janus with Syzkaller by fuzzing 27 file system-specific system calls[1] and executing generated programs on a seed image after being mounted. We hardcode the paths of all available files and directories on a seed image in the description file for Syzkaller to fill the values of certain arguments when fuzzing particular system calls. We run both of these fuzzers against eight file systems for 12 hours. In particular, we launch one Janus instance and one KVM instance for Syzkaller in this experiment. Moreover, we re-execute all the programs generated by Janus to obtain comparable path coverage in KCOV style.

Figure 6.2 presents the evaluation result, which shows that with a context-aware approach based on RPG IR, Janus keeps exploring more code paths than Syzkaller in the span of 12 hours. In particular, Janus eventually visits 2.24×, 1.27×, and 1.25× more unique code paths than Syzkaller when fuzzing the three most popular file systems, XFS v5, Btrfs, and ext4, respectively. Moreover, Janus also outperforms Syzkaller 1.72× and 1.49× on HFS+

[1] Syzkaller and Janus fuzz the following system calls: `read()`, `write()`, `open()`, `lseek()`, `getdents64()`, `pread64()`, `pwrite64()`, `stat()`, `lstat()`, `rename()`, `fsync()`, `fdatasync()`, `access()`, `ftruncate()`, `truncate()`, `utimes()`, `mkdir()`, `rmdir()`, `link()`, `unlink()`, `symlink()`, `readlink()`, `chmod()`, `setxattr()`, `fallocate()`, `listxattr()` and `removexattr()`

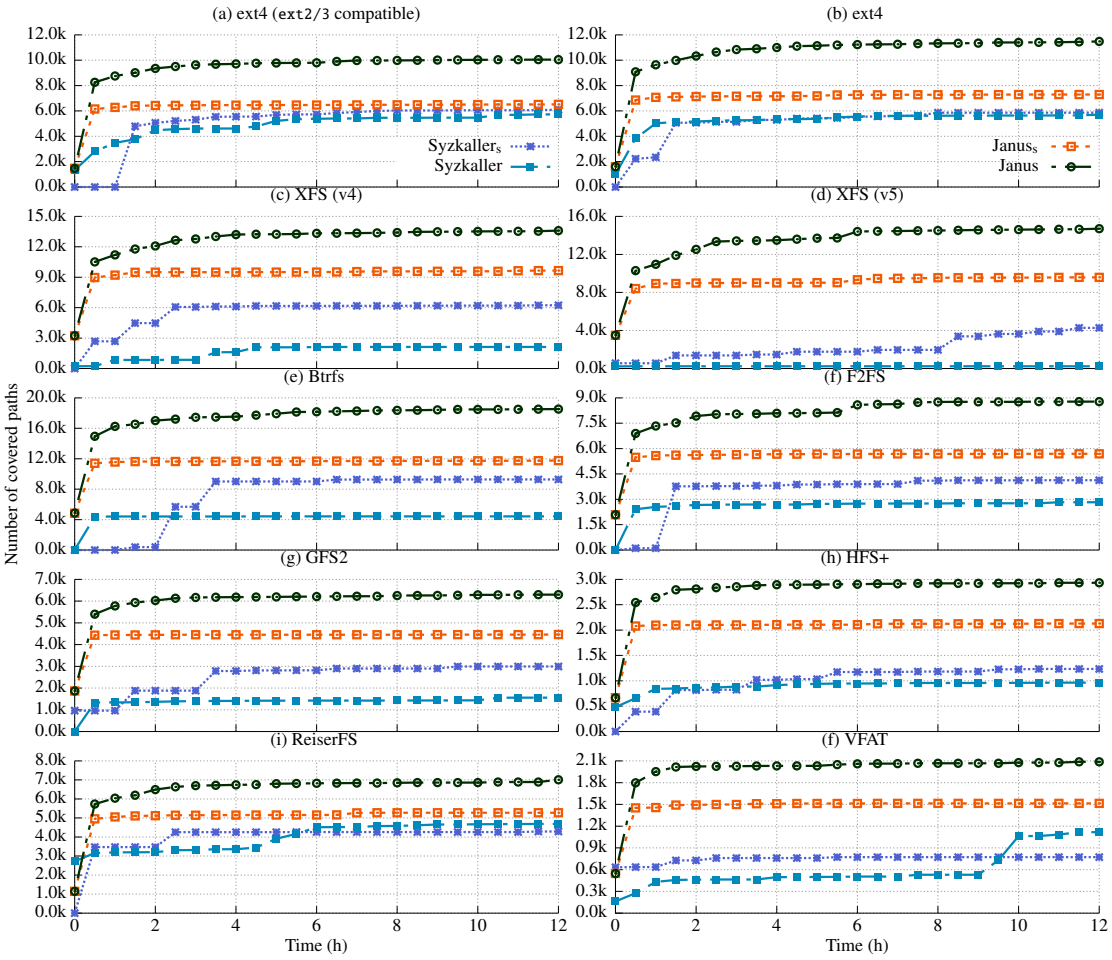


Figure 6.2: The overall path coverage of using Syzkaller and Janus to fuzz eight file system images for 12 hours. The y-axis represents the number of unique code paths of each file system visited during the fuzzing process. In particular, Janus$_s$ and Syzkaller$_s$ generate random system calls to be executed on a fixed seed image, in which Janus$_s$ achieves up to 2.24× higher coverage than Syzkaller$_s$. Janus and Syzkaller fuzz both image bytes and file operations, and Janus visits at most 4.19× unique paths.

and GFS2, respectively.

In brief, Janus generating semantically correct workloads explores more code paths than Syzkaller in all eight popular file systems when only targeting the system calls related to file operations. In particular, the programs generated by Janus manage to visit at most 2.24× more paths. The evaluation result fully demonstrates the effectiveness of RPG IR in terms of file operation fuzzing.

6.3.3 Exploring Two-Dimensional Input Space

To demonstrate the effectiveness of Janus in mutating both image bytes and file operations, we run Janus and Syzkaller on the eight aforementioned file systems with the same seed images for 12 hours. We provide `syz_mount_image()` in the description file to make Syzkaller not only generate system calls but also mutate the bytes in a seed image while invoking 27 file system-specific system calls (see §6.3.2). In this experiment, we simultaneously launch three instances for both Janus and Syzkaller for parallel fuzzing. Moreover, both fuzzers share generated test cases for each corresponding file systems. Figure 6.2 presents the results of this experiment.

We observe that mutating both images and file operations achieves more code coverage than mutating file operations only, which reveals the importance of fuzzing both images and file operations to comprehensively explore a file system. More important, Janus further outperforms Syzkaller on all tested file systems. In particular, Janus achieves at most 4.19×, 4.04×, and 3.11× higher code coverage than Syzkaller when fuzzing Btrfs, GFS2, and F2FS, respectively. For ext4, Janus also hits 2.01× more unique code paths. One reason for such difference is that Janus fuzzes every testcase with a clean LKL instance while Syzkaller keeps using one VM instance to run numerous system calls for a long time and an aging OS may show non-deterministic behaviors when getting the same input and thus miss crashes. More importantly, Syzkaller completely forgoes the context awareness of the file system being fuzzed and blindly emits many file operations that have semantic issues. Meanwhile, Janus being aware of the status of the file objects generates many more valid file operations.

In summary, the end-to-end evaluation of Janus proves the importance of mutating both images and operations in file system fuzzing. Moreover, Janus outperforms Syzkaller on all eight file systems, which originates from the context awareness of Janus enabled via RPG IR. In particular, Janus outperforms Syzkaller at most 4.19× on Btrfs, one of the popular file systems that has an extremely complex design.

6.4 Conclusion

Janus is an evolutionary file system fuzzer that explores an in-kernel file system by exploring its two-dimensional input space, namely images and file operations. By modeling file operations in a context-aware manner as suggested by RPG IR, Janus manages to generate more semantically correct file operations compared to the state-of-the-art system call fuzzer, Syzkaller. We have reported 90 bugs found by Janus in the upstream kernel with 32 CVEs assigned. Moreover, Janus outperforms Syzkaller by exploring at most 4.19$\times$ more code paths when fuzzing popular file systems for 12 hours.

#	File system	CVE	File	Function	Type
1	ext4	CVE-2018-1092	fs/ext4/inode.c	ext4_iget	Use-after-free
2		CVE-2018-1093	fs/ext4/balloc.c	ext4_valid_block_bitmap	Out-of-bounds access
3		CVE-2018-1094	fs/ext4/super.c	ext4_fill_super	Null pointer dereference
4		CVE-2018-1095	fs/ext4/xattr.c	ext4_xattr_check_entries	Out-of-bounds access
5		CVE-2018-10840	fs/ext4/xattr.c	ext4_xattr_set_entry	Heap overflow
6		CVE-2018-10876	fs/ext4/extents.c	ext4_ext_remove_space	Use-after-free
7		CVE-2018-10877	fs/ext4/extents.c	ext4_ext_drop_refs	Out-of-bounds access
8		CVE-2018-10878	fs/ext4/balloc.c	ext4_init_block_bitmap	Out-of-bounds access
9		CVE-2018-10879	fs/ext4/xattr.c	ext4_xattr_set_entry	Use-after-free
10		CVE-2018-10880	fs/ext4/inline.c	ext4_update_inline_data	Out-of-bounds access
11		CVE-2018-10881	fs/ext4/ext4.h	ext4_get_group_info	Uninitialized memory
12		CVE-2018-10882	fs/jbd2/transaction.c	start_this_handle	BUG()
13		CVE-2018-10883	fs/jbd2/transaction.c	jbd2_journal_dirty_metadata	BUG()
14		-	fs/ext4/xattr.c	ext4_xattr_set_entry	Heap overflow
15		-	fs/ext4/namei.c	ext4_rename	Use-after-free
16		-	fs/ext4/inline.c	empty_inline_dir	Divide by zero
17	XFS	CVE-2018-13093	fs/xfs/xfs_icache.c	xfs_iget_cache_hit	Use-after-free
18		CVE-2018-10322	fs/xfs/xfs_inode.c	xfs_ilock_attr_map_shared	Null pointer dereference
19		CVE-2018-10323	fs/xfs/libxfs/xfs_bmap.c	xfs_bmapi_write	Null pointer dereference
20		CVE-2018-13094	fs/xfs/xfs_trans_buf.c	xfs_trans_binval	Null pointer dereference
21		CVE-2018-13095	fs/xfs/libxfs/xfs_bmap.c	xfs_bmap_extents_to_btree	Out-of-bounds access
22		-	fs/xfs/libxfs/xfs_alloc.c	xfs_alloc_get_freelist	Null pointer dereference
23		-	fs/xfs/libxfs/xfs_dir2.c	xfs_dir_isempty	Null pointer dereference
24	Btrfs	CVE-2018-14609	fs/btrfs/relocation.c	__del_reloc_root	Null pointer dereference
25		CVE-2018-14610	fs/btrfs/extent_io.c	write_extent_buffer	Out-of-bounds access
26		CVE-2018-14611	fs/btrfs/free-space-cache.c	try_merge_free_space	Use-after-free
27		CVE-2018-14612	fs/btrfs/ctree.c	btrfs_root_node	Null pointer dereference
28		CVE-2018-14613	fs/btrfs/free-space-cache.c	io_ctl_map_page	Null pointer dereference
29		-	fs/btrfs/volumes.c	btrfs_free_dev_extent	BUG()
30		-	fs/btrfs/locking.c	btrfs_tree_lock	Deadlock
31		-	fs/btrfs/volumes.c	read_one_chunk	BUG()
32	F2FS	CVE-2018-13096	fs/f2fs/segment.c	build_sit_info	Heap overflow
33		CVE-2018-13097	fs/f2fs/segment.h	utilization	Divide by zero
34		CVE-2018-13098	fs/f2fs/inode.c	f2fs_iget	Out-of-bounds access
35		-	fs/f2fs/segment.h	verify_block_addr	BUG()
36		CVE-2018-13099	fs/f2fs/segment.c	update_sit_entry	Use-after-free
37		CVE-2018-13100	fs/f2fs/segment.c	reset_curseg	Divide by zero
38		-	fs/inode.c	clear_inode	BUG()
39		-	fs/f2fs/node.c	f2fs_truncate_inode_blocks	BUG()
40		CVE-2018-14614	fs/f2fs/segment.c	__remove_dirty_segment	Out-of-bounds access
41		CVE-2018-14615	fs/f2fs/inline.c	f2fs_truncate_inline_inode	Heap overflow
42		CVE-2018-14616	fs/crypto/crypto.c	fscrypt_do_page_crypto	Null pointer dereference
43	HFS+	CVE-2018-14617	fs/hfsplus/dir.c	hfsplus_lookup	Null pointer dereference

Table 6.2: The list of previously unknown bugs in widely used file systems found by Janus that have already been fixed in Linux kernel v4.16, v4.17, and v4.18.

CHAPTER 7
RPG: RANDOM PROGRAM GENERATOR

To figure out a more generic approach to utilize RPG IR for context-aware API fuzzing in different domains, we propose RPG (Random Program Generator) in this chapter.

7.1 Overview

RPG proposes a formal approach to randomly generate the API programs that can be described by RPG IR. Figure 7.1 presents an overview of RPG by showing the workflow of generating a random call of the DOM API `Document.createElement()` as an example.

To make RPG generate random API programs for a targeted API-based system, one needs to provide a number of ASL (**A**PI **S**pecification **L**anguage) files as the input. ASL is a specification language elaborated by RPG for users to formally describe the concerned APIs, including the objects they access (*i.e.*, object type and state declarations), the arguments that compose them (*i.e.*, argument definitions), and their syntax and semantics (*i.e.*, API definitions). After validating the input ASL files, RPG compiles them into a JSON object (see ①).

Next, RPG relies on a transpiler to automatically translate the JSON object into the fuzzer code that manages to generate random calls of described APIs in RPG IR according to the fuzzing algorithms mentioned in §4 (see ②). More specifically, a class implementation is emitted by RPG for each API that realizes the following:

- The representation of an API call in RPG IR, including arguments and returned objects, etc.
- The context-aware generation of such an API call based on the context via `generate()`.
- The translation from the API call in RPG IR into plain text via `__str__()` for testing.

For instance, the `createElement_input` class in Figure 7.1 implements not only how an

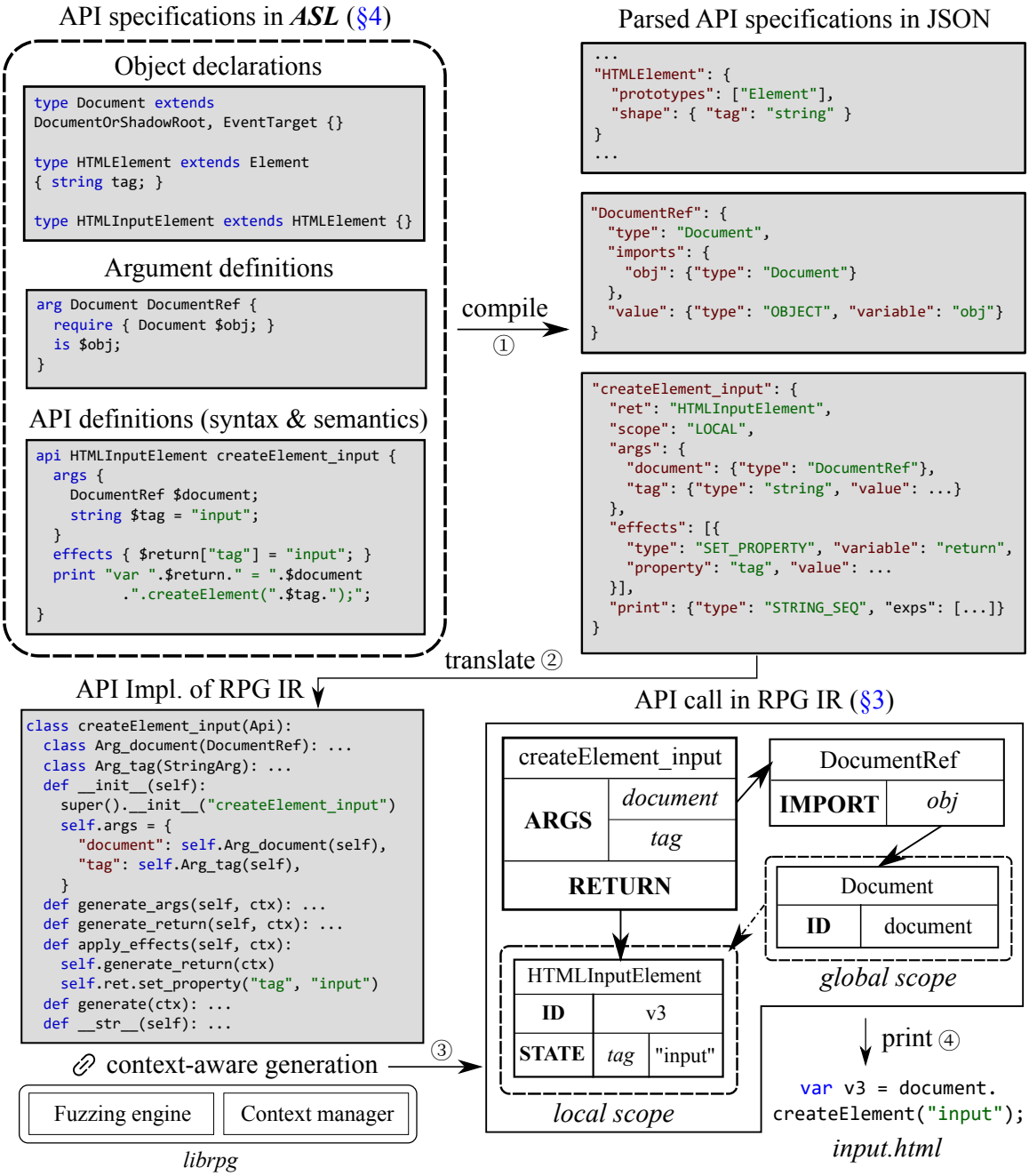


Figure 7.1: Overview of RPG, including major components and their interactions. Users describe the APIs to be fuzzed in ASL (API Specification Language). RPG compiles the ASL files into a JSON object (①), which is further translated into the fuzzer code (②) that randomly generates the described APIs in RPG IR (③) with the help of a shared library, `librpg`. A generated API program in RPG IR can be converted into a text file for testing at the end.

API call of `Document.createElement("input")` is described in RPG IR but also how to randomly generate such an API call.

As part of the toolchain of RPG, `librpg` is a shared library that utilizes the emitted

classes to conduct program generation. First, `librpg` involves a context implementation (*i.e.*, context manager) to maintain scopes and objects in an RPG IR program. More importantly, the fuzzing engine in `librpg` drives the actual API generation by cooperating with the emitted code and the context manager. As shown in Figure 7.1, to generate a call of `Document.createElement("input")` at a given program location, the fuzzing engine instantiates the `createElement_input` class and invokes its `generate()` method with the program context maintained by the context manager (see ③).

An API program output by RPG is composed of a number of generated API calls in RPG IR and the present context. The API program is eventually converted into plain text for testing (see ④). One can also implement a custom executor of the program that directly accesses the generated API calls through the interfaces of RPG IR and tests them in a preferred way.

Caveat. RPG, as its name suggests, focuses only on producing random API programs with reduced semantic errors, which is *the most important* task of an end-to-end API fuzzer. RPG is not in charge of scheduling or evaluating the generated programs. As an input generator with a self-contained design, RPG can be seamlessly integrated into existing fuzzing platforms, such as Google ClusterFuzz [18] and Mozilla Grizzly [19], which manage the jobs that are beyond the scope of RPG.

7.2 ASL

ASL (**A**PI **S**pecification **L**anguage) is designed for users to formally describe API specifications based on the API model proposed by RPG IR. RPG automatically compiles ASL files into the fuzzer code that generates random API programs in RPG IR. In other words, ASL is used to implement an API fuzzer driven by RPG IR at a high level of abstraction. Therefore, ASL has two design goals:

- Describing an API by strictly following the model proposed by RPG IR.
- Defining the randomness of an API. Unlike an API call in RPG IR whose argument

values are deterministic for testing, ASL aims to specify every possible values for an API argument, which is randomly decided during generation.

In this chapter, we present the detailed design of ASL.

7.2.1 Object Declaration

Before defining APIs, one needs to first declare all types of objects being created and used by the APIs in ASL. ASL provides the `type` statement to declare a custom object type, which describes its type name, the optional parent types it extends from, and a list of properties including their names and types. Figure 7.2 specifies the syntax of a `type` statement.

```
type name extends [basetype1 [,basetype2] ... [,basetypeN]] {
    [type1 key1; [type2 key2;] ... [typeN keyN;]]
}
```

Figure 7.2: Syntax of object type specification in ASL.

The properties form the state of an object defined in RPG IR. Note that an object of a child type also owns the properties from its inherited type. In addition, when an object of a specific type T is wanted by an API call, the objects of the types that extend from T should also be considered. In Figure 7.1, we present the declarations of three object types: `Document`, `HTMLElement`, and `HTMLInputElement`. All of them are extended from specific base types. Moreover, both `HTMLElement` and `HTMLInputElement` have a `tag` property in their states due to type inheritance.

Object declaration in ASL strictly follows the object model of RPG IR (see §3.2). Meanwhile, type inheritance makes object declaration concise in ASL when describing API-based systems like DOM engines that involve numerous object types extending from each other, such as a DOM engine.

7.2.2 Macro

A *macro* is a context-free expression that represents a random value not depending on any object. ASL's macros are similar to *symbols* in the context-free grammars adopted by

existing fuzzers. A macro is defined by a `macro` statement, which specifies its type, name, and an expression calculating a random value. For reference, we list several macros defined in the ASL file for DOM fuzzing in Figure 7.3. ASL mainly supports the following types of macro expressions besides simple string and number literals.

Number intervals. The expression samples a random number, which can be either integers or floats, from the given interval (see Equation 7.1).

$$exp := number \sim number \tag{7.1}$$

String concatenations. As described by Equation 7.2, dots are used to concatenate a list of string values into one string.

$$exp := exp_1.exp_2.\ \ldots\ .exp_n \tag{7.2}$$

Macro calls. As described by Equation 7.3, a macro can be recursively constructed by other defined macros. This is the main purpose of introducing macros in ASL, which eases the description of a random value that has a complicated structure. When a macro is used in an expression, it is surrounded by a pair of angle brackets.

$$exp := \langle macro \rangle \tag{7.3}$$

Choices. A choice expression defined in Equation 7.4 enumerates a list of possible values by commas for random selection during generation.

$$exp := exp_1, exp_2,\ \ldots\ , exp_n \tag{7.4}$$

Repeats. As shown in Equation 7.5, the curly braces, enclosing one or two integers by a comma indicate that its decorated value is repeated for A times or any times between A and

```
// A random integer from [0, 100].
macro int Int100 = 0~100;

// A random string that is either "on" or "off".
macro string OnOrOff = "on", "off";

// A random percentage value.
macro string Percent = "0%", "100%", <Int100>."%";

// A random value for the CSS property: border-width.
macro string BorderWidth = <LineWidth>{1,4};

// A random value for the CSS property: border.
macro string Border = ||[<LineWidth>, <LineStyle>, <Color>];

// A random value for the CSS transform function: matrix().
macro string MatrixFunction = "matrix(".<Number>#{6}.")";
```

Figure 7.3: Examples of ASL macros, which are used to describe DOM specification. See [51] for the specification of the CSS properties and CSS function.

B, respectively. By default, the repeated values are joined with spaces. One can also specify the optional hash symbol to set the separator to be a comma, which is also commonly used for connecting a list of values in common APIs.

$$exp := exp\text{'}\ (\#)?\{A(,B)?\} \tag{7.5}$$

Ordered samples. Equation 7.6 presents an expression consisting of a list of unit values led by a `||` operator. Those unit values are randomly selected to be joined into a string, where each unit value is optionally present.

$$exp := ||[exp_1, exp_2, \ \ldots\ , exp_n] \tag{7.6}$$

The last three types of expressions are additionally supported by ASL to get rid of repeated efforts when describing common random values.

In general, the macros guarantee that ASL is able to describe what existing context-free grammars can describe. ASL relies on macros to specify random values for the context-free arguments of an API call in RPG IR (see §7.2.3).

7.2.3 Argument Definition

ASL uses `arg` statements to describe API arguments in RPG IR (see §3.3.2). Figure 7.5 lists examples of various argument specifications. Basically, `arg` statements have two forms, whose syntax is demonstrated in Figure 7.4.

```
1 // Simple form
2 arg type name = expression;
3
4 // Complete form
5 arg type name {
6     [require { type1 $obj1; [type2 $obj2;...[typeN $objN;]] }]
7     [assert { cond1; [cond2; ... [condN;]] }]
8     [args { argname1 $arg1; [argname2 $arg2;...[argnameN $argN;]] }]
9     is expression;
10 }
```

Figure 7.4: Syntax of argument specification in ASL. For specifying an argument, only its type, name, and the `is` statement that calculates the argument value are mandatory.

The `arg` statement in a simple form (line 2 in Figure 7.4) describes an argument that does not explicitly depend on any imported object or other arguments with its name, type, and value expression. The value expression is normally a literal or a macro. For instance, the argument `MutationEventType` (line 2) is such a simple one that has a random string value returned by the macro <`MutationEvent`>. Not only that, one can use an `arg` statement associated with a block (line 5-10 in Figure 7.4) to describe the complete form of an argument in RPG IR, which consists of its name, type, dependent objects and their state checks, sub-arguments, and value expression (see Equation 3.14).

The `require` block in an `arg` block specifies the imported objects, including their wanted types. Similarly, the internal `args` block describes a list of sub-arguments. Every imported object and sub-argument are assigned a unique identifier starting with `$` and becomes *a named variable* being used in object state assertions and argument value expressions.

Object state assertions are described in the `assert` block. Table 7.1 summarizes how ASL describes the three types of asserting conditions defined in RPG IR. In ASL, `$obj["k"]` represents the value of property `k` in the state of an object `obj`. For example, the argument

```
1  // Name of a random mutation event in DOM.
2  arg string MutationEventType = <MutationEvent>;
3
4  // Reference to a write file descriptor for syscalls.
5  arg FD WriteFDRef {
6      require { FD $obj; }
7      assert { $obj["W"] == 1; }
8      is $obj;
9  }
10
11 // A random CSS ID selector.
12 arg unquoted HTMLElementSelector {
13     require { HTMLElement $element; }
14     is "#".$element;
15 }
16
17 // A random :focus CSS pseudo-class.
18 arg unquoted FocusPseudoClass {
19     args {
20         BaseSelector $base;
21     }
22     is $base.":focus";
23 }
```

Figure 7.5: Specifications of four different types of arguments in ASL. Note that `unquoted` on lines 12 and 18 is the primitive type of a string value whose text representation is not surrounded by two quotes.

`WriteFDRef` defined on line 5 in Figure 7.5 represents a reference to a write file descriptor object. The argument requires its imported object called `$obj` to have the `W` property equal to 1.

Assertion in ASL	Assertion in RPG IR
`"k" in $obj`	`HAS_PROPERTY(OBJ, K)`
`$obj["k"] == v`	`PROPERTY_EQ(OBJ, K, V)`
`$obj["k"] != v`	`PROPERTY_NEQ(OBJ, K, V)`

Table 7.1: Syntax of object state assertions in ASL and their corresponding definitions in RPG IR.

Lastly, `is` in an `arg` block is followed by the argument value. Table 7.2 lists various expressions used by ASL to describe the argument value, which strictly follow the definition in RPG IR (see Equation 3.24). Similar to macro expressions, argument value expressions are also concatenated by dots. Note that when `$obj` is directly used as the argument value, the argument is a reference to the corresponding imported object. A typical example

is the argument `WriteFDRef` presented in Figure 7.5. Meanwhile, `$obj` being used in a string concatenation represents the identifier of the object. For instance, the argument `HTMLElementSelector` described at line 12 in Figure 7.5 has a CSS selector value starting with the hash symbol followed by the ID of a random HTML Element.

Expression in ASL	**Expression in RPG IR**	**Concatenable**
`$obj`	`OBJ`	
1 or `<macro>`	*number literal*	
`"AAAA"` or `<macro>`	*string literal*	✓
`$obj`	`OBJ[ID]`	✓
`$obj["k"]`	`OBJ[STATE][K]`	-
`$arg`	`ARG[VALUE]`	✓
$\text{exp}_1.\text{exp}_2.\ \ldots\ .\text{exp}_n$	$\texttt{EXP}_1 + \texttt{EXP}_2 + \ldots + \texttt{EXP}_n$	

Table 7.2: Syntax of argument value expressions in ASL and their corresponding definitions in RPG IR (see §3.3.2). The **-** mark in the table indicates that only object property values of a string type can be used in string concatenations.

In general, the `arg` statement clearly specifies the following for an API argument: (1) its contextual structure in RPG IR and (2) its random values for fuzzing. The two forms of the statement also increase the flexibility of ASL in argument specification.

7.2.4 API Definition

The `api` statements in ASL describe API specifications based on RPG IR. Figure 7.6 briefly illustrates the syntax of an `api` statement. We also present two examples of describing API specifications in ASL in Figure 7.7.

```
1 api returntype[/g] name {
2     [args { argname1 $arg1; [argname2 $arg2;...[argnameN $argN;]] }]
3     [effects { effect1; [effect2;...[effectN;]] }]
4     [child { api1; [api2;...[apiN;]] }]
5     print expression;
6     [eprint expression;]
7 }
```

Figure 7.6: Syntax of API specification in ASL. For specifying an API, only its return type, name, and the `print` statement are mandatory. The suffix of the return type, `/g`, is optional, which implies that the returned object has a global scope.

```
1 // DOM API: Document.createElement("input")
2 api HTMLInputElement createElement_input {
3     args {
4         DocumentRef $document;
5         string $tag = "input";
6     }
7     effects {
8         $return["tag"] = "input";
9     }
10    print "var ".$return." = ".$document.".createElement(".$tag.");";
11 }
12
13 // File system syscall: close(fd)
14 api void close {
15    args { FDRef $fd; }
16    effects { delete $fd::obj; }
17    print "close(".$fd.");";
18 }
```

Figure 7.7: Specifications of a DOM API and a file system call.

By default, ASL assumes that an API mostly has *one* returned object. Therefore, an `api` statement first declares the type of the returned object. For example, the DOM API `Document.createElement("input")` returns an `HTMLInputElement` object as specified on line 2 in Figure 7.7. Meanwhile, `void` is a special return type for the APIs that do not return any new object, such as the system call `close` described on lines 14-18 in Figure 7.7. ASL predefines a variable called `$return` to represent the returned object referenced in API effects and print functions.

Next, the optional `args` statement in the block of an `api` statement declares the arguments of an API. Every declared argument is also associated with a variable name tagged by `$` for references in object operations (*i.e.*, API effects) and print functions. Most arguments like `$document` of the API `createElement_input` and `$fd` of the API `close` are defined by specific `arg` statements externally. Meanwhile, an inline argument such as `$tag` on line 5 in Figure 7.7 is directly defined by an expression in the `args` statement.

Furthermore, the optional `effects` argument describes the object operations performed by an API. Table 7.3 lists the syntax of the operations in ASL, which are formally defined by RPG IR in §3.3.3. In particular, object creation is implied by the returned type of an API

Effect in ASL	Effect in RPG IR
Indicated by the return type	`NEW_OBJECT(API, OBJ)` `NEW_GOBJECT(OBJ)`
`delete $obj`	`DEL_OBJECT(API, OBJ)`
`$obj["k"] = v`	`SET_STATE(OBJ, K, V)`

Table 7.3: Syntax of API effects in ASL, including their corresponding definitions in RPG IR (see §3.3.3).

described in ASL and thus omitted in the `effects` statement. Regarding the examples in Figure 7.7, the effect of `createElement_input` is to update the state of the returned object (line 8). Meanwhile, `close` invalidates the file descriptor imported by the only argument `$fd` (line 16).

The `child` statement is designed for the API creating a new scope. More specifically, the statement enumerates the APIs that can be called in the created scope. Lastly, the `print` and `eprint` statements describe the string representation of an API, which is a string concatenation of string literals, argument values, and the identifier of returned objects as defined by RPG IR.

In brief, RPG IR models an API call as a tuple of six elements (see Equation 3.11), each of which is described by a corresponding statement mentioned above in ASL. Therefore, it becomes straightforward to translate the specification of an API in ASL into the fuzzer code that can generate random API calls in RPG IR.

7.2.5 Further Notes

In this section, we further explain certain details of ASL that are not covered so far.

Predefined global objects. An API-based systems may predefine certain global objects with specific identifiers. For example, `document` and `window` can be globally used without explicit creation in an HTML document. ASL supports defining such an object with the `global` statement, as shown in Figure 7.8.

Object notations. Due to the recursive definition of an argument, referencing an imported

```
// A predefined global object
global type identifier;
```

Figure 7.8: Syntax of declaring a predefined global object in ASL.

object in an `api` statement requires locating the underlying argument that exactly imports the object. This is achievable because every argument declared in the `args` statement is tagged with an identifier. More specifically, ASL introduces the `::` operator to notate an object outside where it is imported. Considering a general api `F` as described in Figure 7.9, the object `$obj` is notated as `$arg1::arg2:: ... ::argN::obj` when being used in the description block of `F`. For instance, in Figure 7.7, the object imported by the argument `$fd` is notated as `$fd::obj` for describing the API effect.

```
api string F { args { ARG1 $arg1; } }
arg string ARG1 { args { ARG2 $arg2; } }
arg string ARG2 { args { ARG3 $arg3; } }
...
arg string ARGN { require { T $obj; } }
```

Figure 7.9: Specification of an example API `F` for explaining the external notation of an imported object declared in an `args` statement.

User extensions. Similar to existing grammar-based approaches, ASL supports calling external functions for generating the values that cannot be described by ASL itself, such as a current timestamp. An external function call starts with the `%` symbol (*e.g.*, `%timestamp()`). One needs to implement such a function manually, which is imported by the fuzzing engine of RPG.

Comment. For the convenience of users, ASL files support line comments and block comments via `//` and `/*...*/`, respectively.

7.2.6 Summary

ASL is designed for users to formally program a context-aware API fuzzer based on RPG IR. Particularly, ASL strictly follows the model of RPG IR to describe APIs.

It is thus straightforward for RPG to automatically compile ASL files into the class

implementations for describing and generating API calls, which does not require any manual effort. Meanwhile, ASL has a clean design, which is easy to master by developers who are not even familiar with fuzzing. Compared to the previous context-aware API fuzzers such as Janus and FreeDom that do not rely on any description language, ASL makes RPG IR much more usable in practice.

7.3 Implementation

We implement a prototype of RPG that internally supports end-to-end DOM fuzzing and SVG fuzzing by default. Table 7.4 presents the lines of code (LoC) of each part of RPG. Except for the ASL grammar file and the internal ASL files that describe DOM specifications,

	LoC	Language
RPG		
ASL grammar	340	ANTLR v4
ASL compiler	540	
Transpiler	930	Python
`librpg`	250	
Default fuzzing support		
DOM specifications	51,150	ASL
SVG specifications	21,282	ASL
Specification extractor	3,960	Python
Executor	180	Python

Table 7.4: Implementation complexity of prototyping RPG, including the ASL files for describing DOM and SVG APIs.

all the other parts of RPG are implemented in Python 3.8. In this chapter, we describe the implementation details of several main parts.

7.3.1 ASL Compiler and Transpiler

We write ASL grammar for ANTLR v4, a popular *LL* parser generator [52]. ANTLR turns the grammar into corresponding lexer and parser code in Python. The ASL compiler accepts a group of ASL files as input and compiles them by leveraging the lexer and parser generated

by ANTLR. During the compilation, the compiler performs essential semantic checks on the input files and parses them into a JSON object. The transpiler of RPG further translates the JSON object into the class implementations for the described APIs and their arguments. The macros described in the ASL files are also translated into Python functions to be invoked for specific random values.

7.3.2 LIBRPG

`librpg` is a Python library that contains two parts: (1) the fuzzing engine, which guides the generation of an API program and (2) the context manager, which implements scopes in RPG IR, whose design is presented in §4.

7.3.3 DOM Fuzzing via RPG

Our prototype of RPG has a default support for generating random HTML and SVG documents to fuzz web browsers. More specifically, we provide the ASL files that describe the specification of DOM and SVG APIs. Note that modern web browsers support thousands of APIs in HTML and SVG documents. To reduce manual efforts on writing those ASL files, we create Python scripts to crawl the definitions of the related APIs from two sources, including Mozilla MDN Web Docs [53] and the IDL files in the source of WebKit that contains DOM API declarations, and translate them into ASL. We still need to manually specify the API effects which are not formally described in the sources. In addition, we also implement an executor to test the generated HTML and SVG documents in WebKit, a mainstream web browser, on multiple cores and collect crashes at runtime.

7.4 Evaluation

In this section, we evaluate RPG and ASL by answering the following questions.

- **Q1.** Compared to existing grammars for API fuzzing, is ASL more capable of describing distinct API syntax and semantics? (§7.4.1)

- **Q2.** Does RPG have the same ability as existing context-aware API fuzzers to generate random API programs with reduced semantic errors? (§7.4.2)

In our experiments, we run RPG on a 24-core machine running Ubuntu 18.04 with Intel Xeon Gold 6126 processors and 384 GB memory.

7.4.1 Comparison between ASL and Existing Grammars

To objectively evaluate the capability of ASL in terms of API description, we compare ASL with the grammars adopted by existing fuzzers in Table 7.5 from two aspects.

Syntactic Features

First, we compare the abilities of ASL and existing grammars to address API syntax.

Context-free grammars. Dharma [2], Avalanche [4] and Domato [1] rely on similar context-free grammars to describe APIs. In such grammars, one can define a *symbol* [1] that describes the generation rule for a specific type of random value. The symbols and literals can be freely combined with various operators to define new symbols that represent more complicated values such as an API argument and ultimately a complete API call. Theoretically, such a symbol-based approach manages to output API calls in any format. In addition, those context-free grammars support extensions through external functions implemented by users to calculate the values not covered in a grammar file.

ASL is as expressive as the context-free grammars adopted by Dharma, Avalanche, and Domato in terms of API syntax description. In particular, the macros in ASL and the symbols in those grammars are functionally the same. Moreover, one can program the `print` argument (see §7.2.4) to output an API call in any preferred string representation.

Syzlang. Similarly, syzlang, the grammar of Syzkaller [7], allows for recursively defining random values in a complicated structure via `arrays` and `structs`. Syzlang also supports custom system calls externally implemented by users. Nevertheless, syzlang hardcodes its

[1] We refer to Domato and Avalanche for this naming.

Fuzzer	Syntactic Features				Semantic Features			
	Format	Symbol	Operation	External	Ref	Type	State	Effect
Dharma	✓	✓	✓	✓	✓			
Avalanche	✓	✓	✓	✓	✓			
Domato	✓	✓	✓	✓	△	△		
syzlang		✓	✓	✓	✓	✓		
ASL	✓	✓	✓	✓	✓	✓	✓	✓

Format: Customizable API formats.
Symbol: An identifier representing a specific random value (*i.e.*, macro in ASL).
Operation: Symbol operations for describing random values.
External: Extension via user-defined functions.

Table 7.5: Comparing ASL with other grammars to describe API specifications. Syzlang is the name of the grammar used by Syzkaller.

API format based on system call interfaces. In other words, syzlang organizes the return value and arguments of an API call in a fixed way. Therefore, syzlang cannot describe any other types of APIs beyond system calls. By contrast, ASL and those context-free grammars have no assumption imposed on the API format and are considered more expressive.

Semantic Features

We now compare the semantic awareness of ASL with that of existing grammars for fuzzing.

Dharma and Avalanche. Dharma and Avalanche support marking a created object with an identifier in their grammars, which can be used for references afterward. Nevertheless, unlike ASL, their grammars are typeless and do not describe object states or API effects in any form.

Domato. The context-free grammar used by Domato marks created objects with their types for DOM APIs in JavaScript. Nevertheless, we still observe reference and type errors in CSS rules and HTML element nodes (see §5.2.1), which indicates its incomplete awareness of object references and types. Similar to Dharma and Avalanche, the grammar used by Domato does not tackle object states and API effects.

Syzlang. Syzlang uses a *resource* to represent the object of a specific type being created or used by an API. Therefore, Syzlang outperforms the context-free grammars in terms of

avoiding reference and type errors. Nevertheless, Syzlang does not consider object states and API effects as well.

Summary

Compared to existing grammars for API fuzzing, ASL is effective in describing distinct API formats and addresses object-related API semantics to the greatest extent. Therefore, we consider ASL as a more generic and advanced language to describe APIs for the fuzzing purpose.

7.4.2 Effectiveness of RPG

DOM Fuzzing

To prove that RPG manages to generate random API calls in a context-aware manner, we compare the fuzzing performance of RPG with that of Domato, a typical DOM fuzzer based on a context-free grammar, and FreeDom, the state-of-the-art DOM fuzzer performing context-aware generation. More specifically, we run 40 instances of every fuzzer to fuzz an ASan (AddressSanitizer [54]) build of WebKitGTK 2.28.0 with random HTML documents for 24 hours. For a fair comparison, we ensure that the ASL files used by RPG in this experiment exactly describe the DOM APIs covered by the other two fuzzers. We expect that RPG and FreeDom have similar evaluation results, largely outperforming Domato.

Figure 7.10 presents the experimental results. Compared to Domato, RPG triggers around 3× more unique crashes with similar code coverage. More importantly, RPG covers 17 out of 18 (94%) unique crashes found by Domato. §5.2.1 summarizes four types of context dependencies in an HTML document. To further verify that context-awareness enables RPG to outperform Domato, we minimize the inputs of 53 unique crashes found by RPG and observe that 53% of the inputs involve at least one type of context dependency. Especially among the inputs of 37 crashes missed by Domato, around 60% contain object-dependent values. Different from RPG, Domato's being unaware of the context suffers from

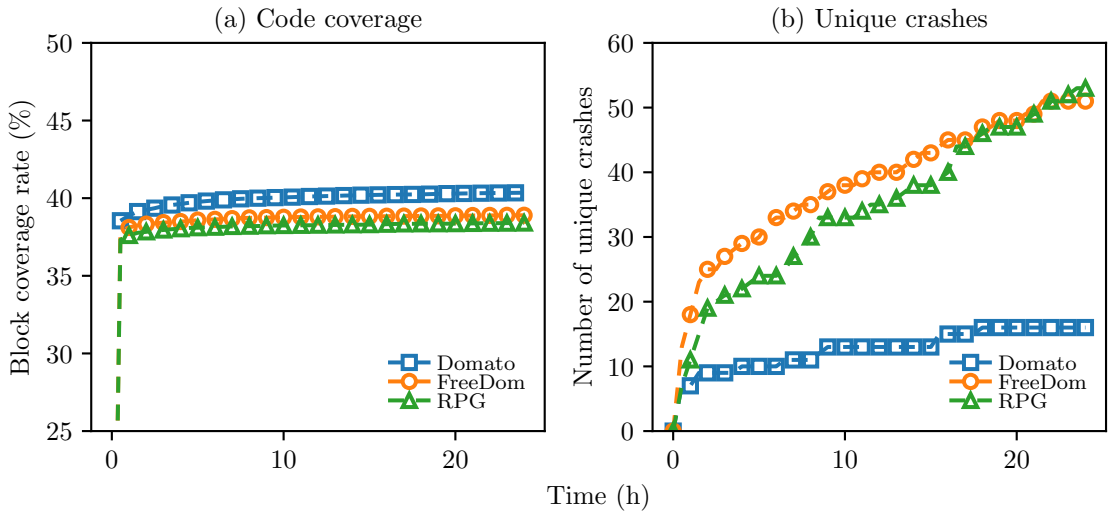


Figure 7.10: Code coverage and discovered unique crashes of Domato, FreeDom, and RPG when fuzzing HTML documents against WebKit with 40 cores for 24 hours.

semantic errors when it tries to generate those values. Meanwhile, RPG has almost the same experimental results as FreeDom. More specifically, the difference of the code coverage achieved by two fuzzers is less than 2%. Meanwhile, FreeDom discovers 51 unique crashes, 55% out of which are contextual ones.

SVG Fuzzing

We also evaluate the effectiveness of RPG by fuzzing SVG documents described by the corresponding specifications in ASL. We compare the fuzzing result with that of FreeDom and also Dharma, another known context-free grammar-based fuzzer developed by Mozilla. Again, the targeted API-based system is WebKit 2.28.0 and we run 40 instances of every fuzzer for 24 hours.

As shown in Figure 7.11, RPG visits around 12% more basic blocks than Dharma and discovers 12 unique crashes in total. Meanwhile, Dharma fails to find any crash during the experiment. §5.4.2 mentions that Dharma is not aware of the exact SVG element whose attributes are to be animated and rarely constructs a valid SVG animation, which is a requisite for a majority of the crashes found by RPG. Moreover, RPG manages to generate

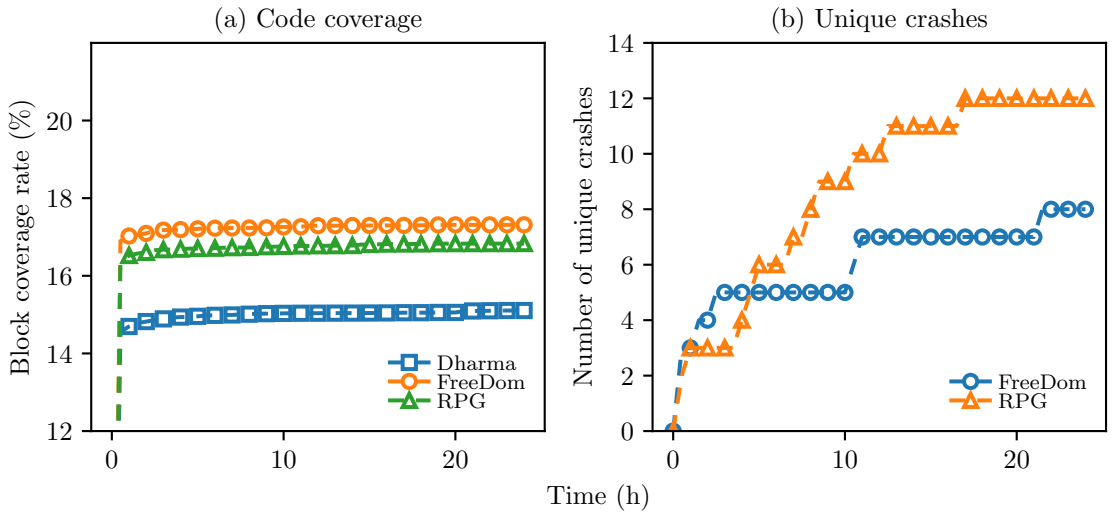


Figure 7.11: Code coverage and discovered unique crashes of Dharma, FreeDom, and RPG when fuzzing HTML documents against WebKit with 40 cores for 24 hours. During the experiment, Dharma failed to trigger any crash and RPG triggered four more crashes than FreeDom.

random SVG attributes that involve the reference to an SVG element of a specific type, such as `xlink:href`, `marker` and `filter`, which are not correctly described by the typeless grammar of Dharma. On the other hand, RPG achieves similar code coverage as FreeDom. In addition, RPG triggers all the crashes discovered by FreeDom in the fuzzing run.

Summary

Similar as FreeDom, RPG adopts a context-aware approach to generate random HTML or SVG documents and thus fundamentally outperforms the leading fuzzers based on context-free grammars. More importantly, RPG proves to be as effective as FreeDom, which indicates that RPG is able to fully resolve the complex semantics of DOM and SVG APIs as the state-of-the-art context-aware fuzzer does with the help of ASL.

CHAPTER 8

DISCUSSION

In this chapter, we discuss the limitations and future directions of the IR-based context-aware fuzzing approach presented by this thesis.

8.1 Comparison between RPG IR and Existing Input Representations

To judge the effectiveness of RPG IR in practice, we compare it with several input representations adopted by existing fuzzers. Table 8.1 presents the comparison results.

Fuzzer	Supposed target	Reference	Type	State	Effect	Scope
Syzkaller	OS kernels	✓	✓			
Fuzzilli	Language processors	✓	✓			✓
PolyGlot		✓	✓			✓
Nyx	Hypervisors	✓	✓		△	
RPG IR	-	✓	✓	✓	✓	✓

Table 8.1: Comparing RPG IR with the representations adopted by existing fuzzers for describing random inputs.

Syzkaller. Syzkaller guarantees that only defined objects (*i.e.*, resources) of correct types serve as the input of a system call in a generated program. Nevertheless, compared to RPG IR, the program model of Syzkaller does not record and validate object states and evaluate API effects, which results in semantic errors, as pointed out in §6.1.2.

Nyx. Nyx [15] is a greybox hypervisor fuzzer, which encodes hypercalls and MMIOs in a similar way as Syzkaller encodes system calls. In addition, Nyx evaluates whether an API deletes an existing object, which is one type of API effect considered by RPG IR. Nevertheless, Nyx does not maintain any additional state information for an object except for its lifetime. Moreover, such an approach adopted by Syzkaller or Nyx to program encoding

is mostly suitable for low-level APIs only.

Fuzzilli and PolyGlot. We also compare RPG IR with the IRs used by two language fuzzers: Fuzzilli [34] and PolyGlot [55]. Their IR programs are expected to have no reference and type errors. Nevertheless, object state evaluation is beyond their scopes. It is noteworthy that their IRs support branches and union types for an object. Those language features are not the focus of API fuzzing and currently not supported by RPG IR.

Generally, existing IRs fail to record, check, and evaluate object states for API calls, which results in the three types of semantic errors as mentioned in §2. Only RPG IR allows for a comprehensive interpretation of object state based semantics of an API.

8.2 Limited API Semantics by RPG IR

We claim that RPG IR cannot deliver the semantics of any APIs in practice. Instead, RPG IR focuses on modeling the API semantics that are related to object states in order to help a context-aware fuzzer generate random API calls that operates the objects in a valid way. For example, certain APIs that are normally exposed by computation platforms such as TensorFlow [56] requires the values of several arguments meet a computational condition. Meanwhile, some API systems require developers to call API A always before API B, where API A and B do not depend on any common object. Such semantics cannot be addressed by RPG IR. Nevertheless, API fuzzing is a random process where it is meaningful to reduce semantic errors but impossible to completely avoid them. RPG IR enables a fuzzer to be context-aware and helps to prevent a large class of semantic errors in common APIs indeed, which still demonstrates its practical value.

8.3 Automated ASL Programming

Currently, we only have the ASL files that describe the DOM specifications, which are extracted from the public sources that list the DOM APIs systematically. To further evaluate the effectiveness of RPG in practice, we need the specifications in ASL for more API-based

systems. However, the API specifications of certain systems are not well-documented or even infeasible, especially for the API semantics. Meanwhile, various research works have been proposed to automatically infer the syntax and semantics of targeted inputs through static or dynamic analysis, such as RESTler [57] for REST APIs, WINNIE [58] for Windows library APIs, and [59] for programming languages. We note that such retrieved information can largely be described by RPG IR, including syntax definition in context-free grammars, argument types of an API, object dependence of an API, etc. Considering the fact that ASL strictly follows the API model of RPG IR, it is not difficult to further translate those analysis results into corresponding ASL files automatically.

8.4 Potential Improvements in RPG IR

We observe several features of existing fuzzing approaches that are not supported by RPG IR. For example, Syzlang supports describing random values at the bit level and using `len`, `bytesize` and `bitsize` to represent the length of a value. Meanwhile, the IRs used by language fuzzers supports branches and union typing. First, those missing features do not affect the context-awareness of RPG in terms of API fuzzing. More importantly, RPG can be easily extended to accomplish those features by referring to the existing approaches.

8.5 Potential Improvements in ASL

In an ASL file, users now can specify a list of simple object checks and API effects in the `assert` statement and `effects` statement, respectively (see §7.2.3 and §7.2.4). To allows users to deliver more complicated semantics of an API, we plan to introduce conditional and loop statements for systematically programming the object checks and API effects based on context information.

CHAPTER 9
RELATED WORK

In this chapter, we introduce existing API fuzzers, description-based fuzzers, and semantic-aware fuzzers and compare them with the context-aware fuzzing approach based on RPG IR.

9.1 API Fuzzing

Numerous API fuzzers have been proposed in recent years. Syzkaller [7], Trinity [8], Razzer [16], IMF [10], kAFL [11], TriforceAFL [49], CAB-FUZZ [12], and Moonshine [60] fuzz OS kernels with random system calls. Meanwhile, known DOM fuzzers include Dharma [2], Avalanche [4], Wadi [3], and Domato [1]. In addition, HYPER-CUBE [14], VDF [17], and Nyx [15] fuzz hypervisors with random hypercalls and I/O operations. Those fuzzers use different approaches to improve the effectiveness and efficiency of API fuzzing in their domains. For example, Syzkaller, kAFL, and Nyx use code coverage to drive input generation. Meanwhile, CAB-FUZZ and Razzer takes a hybrid approach to generate syscall arguments. Moreover, IMF and Moonshine focus on inferring the dependence model between APIs. On the other side, kAFL, HYPER-CUBE, and Nyx propose novel methods to increase the fuzzing throughput of their targets. This thesis presents context-aware API fuzzing, which focuses on reducing semantic errors by maintaining the program context when generating random API calls.

9.2 Description-based Fuzzing

Description-based fuzzers use formal methods to describe a class of input formats based on expert knowledge (*i.e.*, specification) and generate random inputs that match the description.

For example, Dharma [2], Avalanche [4], and Domato [1] generate random documents described in context-free grammars. Peach [61] is known for fuzzing structured inputs expressed by its XML-based data model. Syzkaller [7] relies on a grammar called `syzlang` to specify the system calls for testing. In addition, ProtoFuzz [62] and libprotobuf-mutator [63] are generic fuzzers for structured data serialized in Google Protocol Buffers [64]. In this thesis, we propose ASL in RPG, which is a formal language for API description. Compared to most existing descriptions that focus on presenting the syntax format of an input, ASL is designed to address not only the syntax but also the semantics of APIs.

9.3 Semantics-aware Fuzzing

Semantics-aware fuzzers aim to clarify input semantics and avoid semantic errors in their generated inputs. For example, DIE [65] performs random mutation over the typed AST of a JavaScript program to get rid of type errors. CodeAlchemist [28] interprets the semantics of JavaScript code blocks and assembles them into a semantically correct program for testing. PolyGlot [55], a generic language fuzzer, utilizes a semantic validator to verify the reference and type correctness of a generated program. Compared to programming languages, APIs have more complicated high-level semantics related to the states of the objects being operated by the APIs, which is modeled by RPG IR proposed in this thesis.

CHAPTER 10
CONCLUSION

This thesis summarizes common semantic errors introduced by existing API fuzzers and proposes a context-aware fuzzing approach based on RPG IR. RPG IR is a formal and contextual representation of an API program for fuzzing. To reduce such semantic errors, RPG IR maintains a program context for an API program, which reserves the states of the objects being operated by the API calls in the program, and models the effects of every generated API call to the object states. To evaluate the effectiveness of API fuzzing based on RPG IR, we present two context-aware fuzzers, FreeDom and Janus, for finding bugs in web browsers and file systems, respectively. Both fuzzers rely on RPG IR to describe their targeted APIs and prove the importance of context awareness in API fuzzing by discovering numerous bugs in mainstream browsers and Linux file systems with totally 42 CVEs assigned. In addition, both fuzzers largely outperform the existing API fuzzers relying on context-free grammars to generate semantically incorrect API calls.

To generally adopt context-aware API fuzzing via RPG IR for finding bugs in different API-based systems, we further present RPG in this thesis. RPG accepts API description in ASL, which is automatically compiled into a context-aware API fuzzer targeting the described APIs. In particular, ASL is a formal language for developers to describe API syntax and semantics in a context-aware manner. When being used to fuzz WebKit with the ASL files that describe DOM and SVG specifications, RPG is as effective as FreeDom and finds $3\times$ more unique crashes than Domato, a context-free grammar-based fuzzer.

REFERENCES

[1] I. Fratric, *Domato*, https://github.com/googleprojectzero/domato, 2021.

[2] Mozilla Security, *dharma*, https://github.com/MozillaSecurity/dharma, 2020.

[3] SensePost, *Wadi fuzzing harness*, https://github.com/sensepost/wadi, 2017.

[4] Mozilla Security, *Avalanche*, https://github.com/MozillaSecurity/avalanche, 2020.

[5] Aldeid, *Bf3*, https://www.aldeid.com/wiki/Bf3, 2013.

[6] W. Xu, S. Park, and T. Kim, "Freedom: Engineering a state-of-the-art dom fuzzer," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, FL, Nov. 2020.

[7] Google, *syzkaller is an unsupervised, coverage-guided kernel fuzzer*, https://github.com/google/syzkaller, 2021.

[8] D. Jones, *Linux system call fuzzer*, https://github.com/kernelslacker/trinity, 2018.

[9] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[10] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2017.

[11] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[12] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[13] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[14] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Hyper-cube: High-dimensional hypervisor fuzzing," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[15] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Aug. 2021.

[16] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[17] A. Henderson, H. Yin, G. Jin, H. Han, and H. Deng, "Vdf: Targeted evolutionary fuzz testing of virtual devices," in *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Atlanta, Georgia, Sep. 2017.

[18] Google, *Clusterfuzz*, https://google.github.io/clusterfuzz, 2021.

[19] M. Security, *Grizzly browser fuzzing framework*, https://blog.mozilla.org/security/2019/07/10/grizzly, 2021.

[20] Microsoft Security Research and Defense, *Vulnscan - automated triage and root cause analysis of memory corruption issues*, https://msrc-blog.microsoft.com/2017/10/03/vulnscan-automated-triage-and-root-cause-analysis-of-memory-corruption-issues/, 2017.

[21] Mozilla Security, *Introducing the asan nightly project*, https://blog.mozilla.org/security/2018/07/19/introducing-the-asan-nightly-project/, 2018.

[22] Google, *Chrome vulnerability reward program rules*, https://www.google.com/about/appsecurity/chrome-rewards/index.html, 2020.

[23] F. Beterke, G. Geshev, and A. Plaskett, *Apple safari - pwn2own desktop exploit*, https://labs.f-secure.com/assets/BlogFiles/apple-safari-pwn2own-vuln-write-up-2018-10-29-final.pdf, 2018.

[24] J. Liu and C. Xu, *Pwning microsoft edge browser: From memory safety vulnerability to remote code execution*, POC, 2018.

[25] D. Veditz, *Fixing an svg animation vulnerability*, https://blog.mozilla.org/security/2016/11/30/fixing-an-svg-animation-vulnerability/, 2016.

[26] Mozilla, *Dom fuzzers*, https://github.com/MozillaSecurity/domfuzz, 2019.

[27] M. Zalewski, *crossfuzz*, https://lcamtuf.coredump.cx/cross_fuzz/, 2011.

[28] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[29] W3C, *Svg animations level 2: Attributes to identify the target attribute or property for an animation*, https://svgwg.org/specs/animations/, 2021.

[30] M. Zalewski, *American fuzzy lop*, http://lcamtuf.coredump.cx/afl/, 2015.

[31] LLVM Project, *Libfuzzer - a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html, 2021.

[32] I. Fratric, *The great dom fuzz-off of 2017*, https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html, 2017.

[33] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[34] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Master's thesis, Karlsruhe Institute of Technology, 2018.

[35] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, May 2019.

[36] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The next generation of ext2/3 filesystem.," in *USENIX Linux Storage and Filesystem Workshop*, 2007.

[37] Silicon Graphics Inc. (SGI) and Red Hat Inc., *XFS*, http://xfs.org, 2018.

[38] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," in *Proceedings of the ACM Transactions on Storage (TOS)*, 2013.

[39] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage.," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2015.

[40] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2011.

[41] MITRE Corporation, *CVE-2009-1235*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1235, 2009.

[42] M. Xie and L. Zefan, "Performance improvement of btrfs," *LinuxCon Japan*, 2011.

[43] J. Corbet, *Improving ext4: bigalloc, inline data, and metadata checksums*, https://lwn.net/Articles/469805, 2011.

[44] M. Larabel, *F2FS File-System Moves Forward With Encryption Support*, https://www.phoronix.com/scan.php?page=news_item&px=F2FS-Encryption-Support, 2015.

[45] Kernel.org Bugzilla, *ext4 bug entries*, https://bugzilla.kernel.org/buglist.cgi?component=ext4, 2018.

[46] ——, *Btrfs bug entries*, https://bugzilla.kernel.org/buglist.cgi?component=btrfs, 2018.

[47] MITRE Corporation, *F2FS CVE entries*, http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=f2fs, 2018.

[48] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A Five-year Study of File-system Metadata," in *Proceedings of the ACM Transactions on Storage (TOS)*, 2007.

[49] NCC Group, *AFL/QEMU fuzzing with full-system emulation.* https://github.com/nccgroup/TriforceAFL, 2017.

[50] M. Zalewski, *American fuzzy lop (2.52b) - config.h*, https://github.com/mirrorer/afl/blob/master/config.h, 2017.

[51] W3C, *Cascading style sheets level 2 revision 2 (css 2.2) specification*, https://www.w3.org/TR/CSS22/, 2016.

[52] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[53] Mozilla, *Mdn web docs*, https://developer.mozilla.org/en-US/, 2021.

[54] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[55] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz 'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, May 2021.

[56] M. Abadi, "Tensorflow: Learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.

[57] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Rest-ler: Automatic intelligent rest api fuzzing," *arXiv preprint arXiv:1806.09739*, 2018.

[58] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[59] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 172–183.

[60] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[61] PeachTech, *Peach fuzzer*, https://www.peach.tech/products/peach-fuzzer/, 2007.

[62] Trail of Bits, *Protofuzz*, https://github.com/trailofbits/protofuzz, 2020.

[63] Google, *Libprotobuf-mutator*, https://github.com/google/libprotobuf-mutator, 2020.

[64] ——, *Protocol buffers*, https://developers.google.com/protocol-buffers, 2021.

[65] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*, May 2020.