FreeDom: Engineering a State-of-the-Art DOM Fuzzer

Wen Xu
Georgia Institute of Technology
wen.xu@gatech.edu

Soyeon Park
Georgia Institute of Technology
spark720@gatech.edu

Taesoo Kim
Georgia Institute of Technology
taesoo@gatech.edu

ABSTRACT

The DOM engine of a web browser is a popular attack surface and has been thoroughly fuzzed during its development. A common approach adopted by the latest DOM fuzzers is to generate new inputs based on context-free grammars. However, such a generative approach fails to capture the data dependencies in the inputs of a DOM engine, namely, HTML documents. Meanwhile, it is unclear whether or not coverage-guided mutation, which is well-known to be effective in fuzzing numerous software, still remains to be effective against DOM engines. Worse yet, existing DOM fuzzers cannot adopt a coverage-guided approach because they are unable to fully support HTML mutation and suffer from low browser throughput.

To scientifically understand the effectiveness and limitations of the two approaches, we propose FreeDom, a full-fledged cluster-friendly DOM fuzzer that works with *both* generative and coverage-guided modes. FreeDom relies on a context-aware intermediate representation to describe HTML documents with proper data dependencies. FreeDom also exhibits up to 3.74× higher throughput through browser self-termination. FreeDom has found 24 previously unknown bugs in commodity browsers including Safari, Firefox, and Chrome, and 10 CVEs has been assigned so far. With the context-aware generation, FreeDom finds 3× more unique crashes in WebKit than the state-of-the-art DOM fuzzer, Domato. FreeDom guided by coverage is more effective in revealing new code blocks (2.62%) and finds three complex bugs that its generative approach fails to find. However, coverage-guided mutation that bootstraps with an empty corpus triggers 3.8× *fewer* unique crashes than the generative approach. The newly revealed coverage, more often than not, negatively affects the effectiveness of DOM fuzzers in bug finding. Therefore, we consider context-aware generation the best practice to find more DOM engine bugs and expect further improvement on coverage-guided DOM fuzzing facilitated by FreeDom.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Vulnerability scanners**.

KEYWORDS

context-aware DOM fuzzing; coverage-guided DOM fuzzing; browser vulnerability discovery

ACM Reference Format:
Wen Xu, Soyeon Park, and Taesoo Kim. 2020. FreeDom: Engineering a State-of-the-Art DOM Fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3372297.3423340

1 INTRODUCTION

A DOM (Document Object Model) engine is a core component of every modern web browser, which is responsible for displaying HTML documents in an interactive window on an end-user device. Considering its giant code base and extraordinary complexity, a DOM engine has always been one of the largest bug sources in a web browser. Meanwhile, we have witnessed many high-severity DOM engine bugs being exploited consistently in remote attacks over the past decade. Hence, to prevent such a prominent cybersecurity threat, all the browser vendors have been working tirelessly to discover and patch bugs in their DOM engines [14, 15, 26, 36, 37].

Though a DOM engine has complex implementation, its input format (i.e., HTML) has detailed specifications. Therefore, smart fuzzing becomes the dominant approach for finding DOM engine bugs in practice [1, 7, 11, 23, 28, 33, 34, 47, 58]. For instance, Google has heavily fuzzed Chrome on 25,000 cores and successfully found over 16,000 bugs, the majority of which reside in the DOM engine [15]. Nevertheless, after nearly 10 years of development, state-of-the-art DOM fuzzers still adopt an obsolete design with a missing justification for the resistance to the latest fuzzing techniques.

Particularly, all the recent DOM fuzzers [11, 33, 34] use static grammars that describe the DOM specification to generate random HTML documents with correct syntax. Nevertheless, an HTML document has rich semantics, which are mainly reflected by all sorts of explicit and implicit data dependencies that widely exist in the specification. However, random generation driven by context-free grammars suffers from a chicken-and-egg problem: a predefined grammar provides several valid options to construct every possible document unit. Meanwhile, constructing a context-dependent unit relies on the concrete value of another unit in a specific document, namely, the exact option rolled for the unit, which can never be known by the static grammar before generating the document. Unfortunately, existing DOM fuzzers fail to completely solve this problem and suffer from semantic errors in their output documents.

More importantly, in research communities, a conventional wisdom is that coverage-guided mutation, which has recently gained popularity [18, 19, 25, 57], outperforms blackbox generation. However, there is no solid evidence that supports or objects to this claim regarding DOM fuzzing. Unfortunately, we are unable to directly utilize existing DOM fuzzers to address this open problem. First, those fuzzers output textual documents without preserving intermediate information. The generated documents thus can only be further mutated by appending new data rather than by many other operations such as data flipping and splicing. More importantly, without sufficient context awareness, existing fuzzers still rarely avoid semantic errors in mutation. Second, different from general fuzzing targets, a launched browser instance never automatically terminates unless a crash occurs. It is also difficult to know the exact timing point when an input document is completely processed because of dynamic rendering tasks (e.g., repainting, animations, transitions, etc.) that may occur anytime. Therefore, existing DOM fuzzers enforce every launched browser instance to exit within 5 to 10 seconds by a preset timeout. This setting results in severely low throughput, which is not applicable in coverage-driven fuzzing.

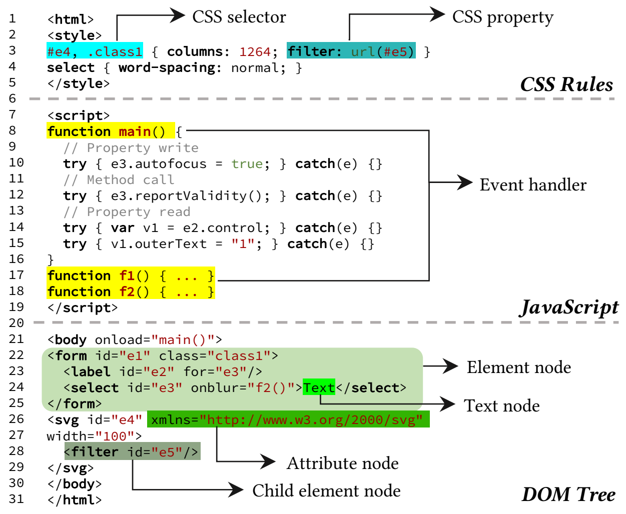
To solve the aforementioned challenges, we present FreeDom, the first end-to-end DOM fuzzing framework that fully supports both document generation and coverage-guided mutation in a distributed environment. We also consider FreeDom as an ideal playground for exploring the possibilities of adopting cutting-edge techniques to fuzz DOM engines. FreeDom uses a custom intermediate representation called FD-IR to describe HTML documents. FD-IR manages to not only follow the DOM specification to record document content in a structural manner but also preserve detailed context information. Instead of emitting plain documents with grammar rules, FreeDom defines various fuzzing operations based on FD-IR, including generating new documents, fully mutating every part of an existing document and merging two documents into a new one. To introduce fewer semantic errors, FreeDom queries the context information to fulfill any data dependence in a document. In addition, FreeDom executes an optimized browser in coverage-guided fuzzing that dynamically kills itself when the processing of an input document mostly completes. The optimization improves the fuzzing throughput of WebKit by 1.48–3.74× compared to using a 5-second time limit and results in very few missed crashes.

We run FreeDom with its generative approach for fuzzing the DOM engines of three mainstream browsers (i.e., Apple Safari, Mozilla Firefox, and Google Chrome) and have successfully found 24 bugs, to which 10 CVEs and 65K USD bug bounty have been awarded. In WebKit, FreeDom discovers nearly 3× more unique crashes than the state-of-the-art DOM fuzzer, Domato, with a similar block coverage, thanks to its context awareness. When fuzzing SVG documents in WebKit, FreeDom triggers around 8 crashes on average, while Dharma, another recent fuzzer, fails to find any crash. We also evaluate FreeDom with its mutation-based approach to determine the advantages and disadvantages of coverage-guided DOM fuzzing. Compared to FreeDom with its generative approach, coverage-guided mutation manages to visit 2.62% more code blocks and discover three new bugs, but meanwhile triggers 3.8× fewer unique crashes in 24 hours. When fuzzing complex software like a DOM engine, the generative approach manages to explore the numerous interfaces in an incomplete but efficient manner through large documents with intended content and thus discovers more bugs in limited time. Nevertheless, the coverage-driven approach is more capable of triggering the bugs that occur with a set of restricted values in a document by incremental mutations.

In summary, this paper makes the following contributions:

- We present an open-sourced[1] DOM fuzzer, FreeDom, with a redefined design, that can run with either a generative approach or a coverage-guided mutational approach based on a context-aware IR for describing HTML documents.
- We perform the first systematic study on the application of coverage-guided mutation in DOM fuzzing and have a detailed discussion of its opportunities and obstacles.
- We have reported 24 bugs found by FreeDom in three mainstream browsers and gained 10 CVEs. Further, FreeDom outperforms the most recent DOM fuzzer by discovering 3× more unique crashes in WebKit.

[1]https://github.com/sslab-gatech/freedom


```
<html>
<style>
#e4, .class1 { columns: 1264; filter: url(#e5) }
select { word-spacing: normal; }
</style>

<script>
function main() {
  // Property write
  try { e3.autofocus = true; } catch(e) {}
  // Method call
  try { e3.reportValidity(); } catch(e) {}
  // Property read
  try { var v1 = e2.control; } catch(e) {}
  try { v1.outerText = "1"; } catch(e) {}
}
function f1() { ... }
function f2() { ... }
</script>

<body onload="main()">
<form id="e1" class="class1">
  <label id="e2" for="e3"/>
  <select id="e3" onblur="f2()">Text</select>
</form>
<svg id="e4" xmlns="http://www.w3.org/2000/svg"
width="100">
  <filter id="e5"/>
</svg>
</body>
</html>
```

Figure 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.

2 BACKGROUND

2.1 DOM Explained

2.1.1 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 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 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, Line 3 in Figure 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 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.

2.1.2 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. In this work, we aim 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 [14, 15, 26, 36, 37], there still have been quite a few full browser exploit chains that target DOM bugs in recent years [5, 24, 49], including one developed by us based on a bug found by FreeDom in Safari.

Caveat. In this work, we are not interested in the logical issues of a DOM engine, such as Universal Cross-Site Scripting (UXSS). In addition, finding the bugs that reside in the JavaScript engine used by a DOM engine is also beyond the scope of this paper.

2.2 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 1. The earliest DOM fuzzers, such as domfuzz [28] and cross_fuzz [58], 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 [55]. By contrast, most recent fuzzers are *static* [1, 11, 34, 47] 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. Among the recent static DOM fuzzers, Domato is considered the most successful one, which has publicly revealed more than 60 bugs in popular browsers. It is actively maintained and widely used by browser vendors for internal testing [12, 16, 17, 39]. Thus, we select Domato as the main state-of-the-art fuzzer for evaluation and comparison in this paper.

DOM fuzzer	Year	Type	Method	Str.	Ctx.	Cov.	Active
domfuzz [28]	2008	D	G	-	-		
Bf3 [1]	2010	S	G				
cross_fuzz [58]	2011	D	G	-	-		
Dharma [34]	2015	S	G	✓			✓
Avalanche [33]	2016	S	G	✓			✓
Wadi [47]	2017	S	G	✓			
Domato [11]	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 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. As a state-of-the-art DOM fuzzer, FreeDom is also static, which supports both blackbox generation and coverage-guided mutation in a context-aware manner.

3 MOTIVATION

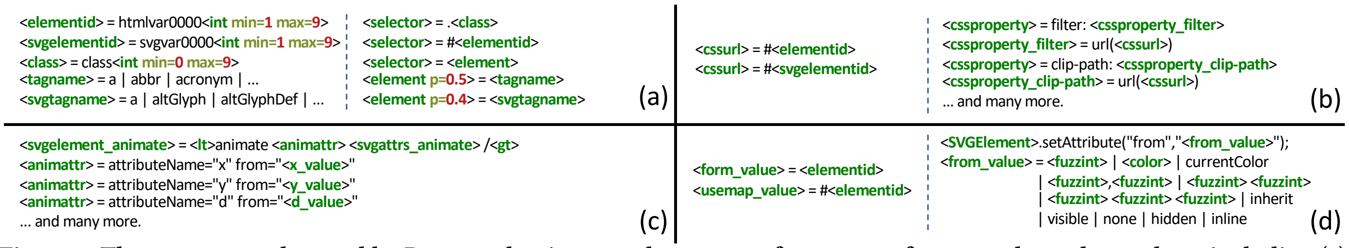
In this section, we systematically analyze the defects of conventional generation-based DOM fuzzers. Their common approach fails to construct inputs with complex semantics and, more importantly, restricts the exploration of applying up-to-date techniques to a DOM fuzzer, which motivates us to propose FreeDom.

3.1 On the Ineffectiveness of Static Grammars

Previous research has pointed out that one crux of fuzzing complex software effectively is to avoid semantic errors [20, 55], 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.

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 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 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



```
<elementid> = htmlvar0000<int min=1 max=9>
<svgelementid> = svgvar0000<int min=1 max=9>
<class> = class<int min=0 max=9>
<tagname> = a | abbr | acronym | ...
<svgtagname> = a | altGlyph | altGlyphDef | ...

<selector> = .<class>
<selector> = #<elementid>
<selector> = <element>
<element p=0.5> = <tagname>
<element p=0.4> = <svgtagname>
```
(a)

```
<cssurl> = #<elementid>
<cssurl> = #<svgelementid>

<cssproperty> = filter: <cssproperty_filter>
<cssproperty_filter> = url(<cssurl>)
<cssproperty> = clip-path: <cssproperty_clip-path>
<cssproperty_clip-path> = url(<cssurl>)
... and many more.
```
(b)

```
<svgelement_animate> = <lt>animate <animattr> <svgattrs_animate> /<gt>
<animattr> = attributeName="x" from="<x_value>"
<animattr> = attributeName="y" from="<y_value>"
<animattr> = attributeName="d" from="<d_value>"
... and many more.
```
(c)

```
<form_value> = <elementid>
<usemap_value> = #<elementid>

<SVGElement>.setAttribute("from","<from_value>");
<from_value> = <fuzzint> | <color> | currentColor
             | <fuzzint>,<fuzzint> | <fuzzint> <fuzzint>
             | <fuzzint> <fuzzint> <fuzzint> | inherit
             | visible | none | hidden | inline
```
(d)

Figure 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.

to the DOM standard. Nevertheless, Domato incorrectly describes this data dependence as shown in Figure 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. First, a number of previous DOM bugs involve dynamic updates of a particular attribute of an element [7], which cannot be achieved by Domato due to its lack of knowledge of what attributes every existing element in a document has. For example, Domato changes the value of a `from` attribute of any SVG element through `setAttribute()` in Figure 2(d). However, `from` is only valid for an SVG animation element. In addition, an SVG `<animate>` element uses its `attributeName` attribute to indicate a particular attribute of its parent element to be animated [52]. Being unaware of what element the `<animate>` element exactly serves, Domato randomly sets `attributeName` to any animatable SVG attribute, as described in Figure 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. First, 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 2(d). More importantly, an attribute value can also have implicit dependence on other attribute values. Regarding an `<animate>` element, the value of the `from` attribute is determined by the value of the sibling attribute, `attributeName`. Without knowing the exact value ever generated for the `attributeName`, the only option for Domato is to statelessly specify some common attribute values for the `from`, as listed in Figure 2(d), which does not take effect at most times.

Summary. 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, FreeDom always memorizes the values that have been generated (i.e., context information) in the current document for generating new values afterwards.

3.2 Exploring Coverage-guided DOM Fuzzing

Most emerging fuzzers adopt a coverage-driven mutation-based approach, which is proven to be effective in practice [18, 25, 57]. 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. To determine the answer, we cannot directly utilize existing DOM fuzzers but design and implement a new one (i.e., FreeDom) for two primary reasons.

Stateless testcases. First, 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 [12], 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 [25, 57] and merging two or more existing inputs [20, 21], are not feasible over plaintext. The extension is thus incapable of achieving the full potential of mutation. By contrast, FreeDom uses an intermediate representation called FD-IR to present a document with stateful structures. FD-IR carries detailed context information to ensure the semantic correctness of a document after all sorts of mutations.

Low-throughput executions. Unlike many other general applications, a browser never automatically terminates without any user interaction. More importantly, particular DOM-rendering tasks such as animations and timing events can be scheduled on-the-fly. Therefore, managing the lifetime of a browser instance becomes challenging. The common solution adopted by existing DOM fuzzers is to run every browser instance with a conservative time limit. By default, ClusterFuzz tests an input for at most 10 seconds. This setting is reasonable for generation-based fuzzing, where a generated input has certain complexity and is more likely to consume much time to be rendered. However, always using several seconds to execute an input largely degrades the performance of mutation-based fuzzing. From our observation, a WebKit window becomes completely idle after 1 second when loading more than 60% of the documents generated by Domato, which generally have sizes of more than 200K bytes. Most inputs processed by a mutation-based fuzzer have a much smaller size, which do not require such a long execution time. Therefore, FreeDom applies a workaround that dynamically terminates a browser instance when the processing of an input document mostly completes.

Summary. Building a mutation-based DOM fuzzer has two unsolved challenges: (1) describing documents in mutable structures rather than in text and (2) improving browser throughput. As the first known fuzzer that fully supports coverage-guided mutation-based DOM fuzzing in practice, FreeDom adopts a stateful IR for context-aware document mutation and optimizes browser throughput with dynamic termination.

4 DESIGN

4.1 Overview

FreeDom is a distributed DOM fuzzer that can perform either blackbox generation-based fuzzing or coverage-guided mutation-based fuzzing. Figure 3 illustrates the overall design of FreeDom. Basically, a number of FreeDom instances run in a cluster and communicate with a central server that manages all the fuzzing data. The testcases processed by FreeDom, namely, HTML documents, are described in FD-IR, which presents document structures along with context information. When doing generation-based fuzzing, a FreeDom instance repeatedly generates new documents in FD-IR, each of which is lowered into an HTML file and opened by a browser instance. If the browser crashes, a crash report that includes the crashing input is uploaded to the server. In coverage-guided fuzzing, the central server not only collects crashes but also maintains (1) a queue that stores the testcases that discover new code paths and (2) a global coverage map. A fuzzer instance iteratively fetches a document in FD-IR from the queue and generates new documents for testing by mutating different parts of it or merging it with another document. The DOM engine of the browser is instrumented for coverage measurement. Every newly generated document that increases overall coverage is saved by the central server for further mutation. In addition, FreeDom targets a high-throughput browser in coverage-guided fuzzing. To avoid waiting for a long time to be terminated by a preset timeout, the optimized browser instance exits with an on-demand timeout installed by itself at the time it completes loading the entire document. In the following paragraphs, we use $\mathbf{FD_G}$ and $\mathbf{FD_M}$ to notate FreeDom working with generation and coverage-guided mutation, respectively.

4.2 Context-aware Document Representation

FreeDom uses a custom intermediate representation called FD-IR to describe both the syntax and semantics of an HTML document. FD-IR strictly follows the DOM specification. Hence, a document in FD-IR can be directly lowered into a real document in plaintext for testing. More importantly, FD-IR carries the context information to generate documents with fewer semantic errors.

4.2.1 Document Context. FreeDom maintains two types of contexts in FD-IR to enable context-aware fuzzing.

Global context. FD-IR maintains a tree structure that records all the elements and their tags and attributes contained in the initial DOM tree for reference. FD-IR also keeps track of all the available tokens (e.g., `class` names, CSS `counter` names, CSS `keyframes` names, etc.). Note that the in-tree elements and tokens are separately organized by FD-IR into two maps, which are indexed by element and token type for performant queries. The global context serves as a basis for constructing four types of context-dependent values, summarized in §3.1, in a document.

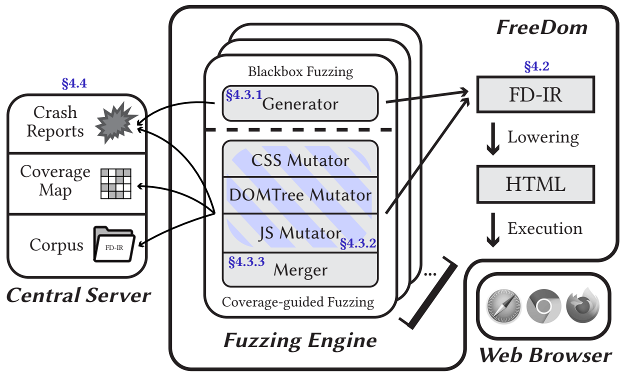


Figure 3: Overview of FreeDom's architecture and workflow. FD-IR is a context-aware IR for describing HTML documents, which supports random generation and mutation.

Local contexts. FD-IR describes a local context for every JavaScript function, which is used to generate semantically correct DOM API calls. The local context not only contains a reference to the global context but also preserves every DOM object locally defined by a particular API call in a distinct map. Different from global elements that can be used by any API call in a function, the local objects are only valid after being defined. Therefore, FD-IR is also aware of the exact location where (i.e., at which line) the object is defined in order to support various API mutations (see §4.3.2) and function merging (see §4.3.3).

4.2.2 Document Representation. We first introduce an important interface of FD-IR called `Value`. As the first-class citizen in FD-IR, `Value` is implemented to represent all sorts of data values that may appear in a document, including but not limited to, *CSS selectors, CSS properties, attributes and arguments and return values of API calls*. In particular, a concrete `Value` involves the definition of three indispensable methods:

(1) `generate()`, which specifies how to randomly generate the data that forms the `Value` given the global or local context.
(2) `mutate()`, which defines how to randomly mutate the `Value` with the global or local context.
(3) `lower()`, which describes how to turn the FD-IR `Value` into the corresponding text string in a document.

A `Value` in FD-IR can be built upon one or more sub-`Values`. For example, the top code in Figure 4 presents the description of a CSS `filter` property. The property itself is an FD-IR `Value`. And its value is defined as `CSSFilterValue`, which is another `Value` that can be either literal or context-dependent. Most basic `Values` like `CSSFilterValue` that do not contain any other `Values` simply mutate through regeneration. A compound `Value` such as `CSSProperty` selectively mutates its contained `Values` (Line 6). However, the FD-IR `Values` that serve as a part of the context to be referred to by other `Values` are immutable once generated. For instance, the bottom code in Figure 4 describes `ReturnValue`, which represents a

```
class CSSProperty(Value):
    def __init__(name, value):
        self.name = name
        self.value = value
    def generate(ctx): self.value.generate(ctx)
    def mutate(ctx): self.value.mutate(ctx)
    def lower(): return "{}: {}".format(name, lower(value))

class CSSFilter(CSSProperty):
    def __init__(name, value):
        super().__init__("filter", CSSFilterValue())

# filter: blur(5px) / url(#id)
class CSSFilterValue(Value):
    def generate(ctx):
        self.ref = None
        if Random.bool():
            self.ref = ctx.getElement("SVGFilterElement")
            if self.ref is not None: return
        self.value = "blur({})".format(Random.length())
    def mutate(ctx): generate(ctx)
    def lower():
        if self.ref is None: return value
        else: return "url(#{})".format(self.ref.id)
```

```
class ReturnValue(Value):
    def __init__(t):
        self.type = t
        self.ref = None
    def generate(ctx): # Here ctx is the local context
        if self.ref is None:
            self.ref = ctx.createObject(self.type)
    def mutate(ctx): pass
    def lower(): return self.ref.id
```

Figure 4: Two non-trivial `Value` instances in FD-IR, including a CSS `filter` property, whose generation and mutation depend on the context, and a return value of a DOM API call, which updates the context when being generated and thus becomes immutable. Note that a `filter` property has many more value choices, which are omitted here.

return value of an API call. Its `mutate()` method is empty because its generation introduces a new object into the local context. If the return value were regenerated, the uses of the object in the subsequent API calls would become invalid. Another example is an `attributeName` attribute that decides the value of its sibling attributes such as `from` (see §3.1). Based on `Value`, FD-IR manages to describe the three parts of a document as follows.

The DOM tree. As mentioned in §4.2.1, FD-IR intuitively uses a multi-branch tree to describe the DOM tree, whose nodes are DOM elements and whose root is the only <body> element. In each element node, FD-IR records its type, a unique `id` for reference, a list of child nodes, and a list of attributes. Each attribute is a particular `Value` instance.

CSS rules. FD-IR also records a list of CSS rules in a document. For each rule, FD-IR maintains a list of CSS selectors and a list of CSS properties, all of which are `Value` instances.

Event handlers. FD-IR maintains a list of event handlers for an HTML document. Among them, one event handler, which is the `onload` event handler of the <body> element, is treated specially as the `main` event handler. The total number of other event handlers in a document is predefined by FreeDom, which never increases during mutation. Each event handler is composed of a list of DOM API calls and the local context described in §4.2.1. FD-IR supports all three types of DOM API calls in JavaScript (see §2.1.1). For an object property read or write, besides the property name, FD-IR records the accessed object and the return or new property value as two `Value` instances. Similarly, for an object method call, in addition to the method name, FD-IR presents the object, the return value, and a list of arguments with more `Value` instances. Note that all these `Value`s rely on the local context for generation and mutation and are lowered into corresponding JavaScript code units.

In general, a document in FD-IR is guaranteed to be:

(1) **Stateful.** FD-IR presents a document in a structural and programmable format rather than in plaintext.
(2) **Context-aware.** FD-IR carries not only document content but also context information, including tree hierarchy and available objects in the global and local scopes.
(3) **Extensible.** One can introduce more `Value` instances to support more DOM features effortlessly.

4.3 Context-aware DOM Fuzzing

A document in FD-IR is composed of many `Value` instances. Fundamentally, FreeDom performs document generation and mutation in a context-aware manner by systematically calling `generate()` and `mutate()` of specific `Value`s with context information.

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

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, FD_G builds a DOM tree by repeatedly invoking the following three methods.

(1) $\mathbf{G_{t1}}$**:** *Insert an element.* FD_G 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, FD_G randomly decides the corresponding element type of the new child by specification. FD_G finally adds the element into the global context.
(2) $\mathbf{G_{t2}}$**:** *Append an attribute.* FD_G creates a new attribute that is owned by an existing element yet is not set. FD_G relies on the global context to generate the initial value of the attribute, which is defined as an FD-IR `Value`. If the attribute value introduces a new token (e.g., `class`, CSS counters, etc.) during generation, the global context will be updated correspondingly.
(3) $\mathbf{G_{t3}}$**:** *Insert a text node.* FD_G 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, FD_G 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.* FD_G generates a CSS selector and adds it into the rule.
(2) $\mathbf{G_{c3}}$**:** *Append a CSS property.* FD_G constructs a CSS property and appends it into the rule.

Both CSS selectors and properties are `Value` instances that are generated based on the global context.

Event Handler Generation. FD_G 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 $\mathbf{G_f}$), FD_G first queries both global and local contexts for available

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

DOM objects that can be used as the arguments of an API call in the current event handler. Then, FD_G 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 line number of its definition is recorded by the current local context.

4.3.2 Single Document Mutation. FD_M aims to mutate three different parts of an existing document with various granularities, while maintaining context information during mutation. We present the detailed mutation algorithms adopted by FD_M as follows.

DOM tree mutation. FD_M may call G_{t1}, G_{t2}, and G_{t3}, as described in §4.3.1, to grow the DOM tree in a document. In addition, FD_M mutates existing nodes in the tree in three ways.

(1) **M_{t1}:** *Mutate an attribute value.* FD_M selects an existing attribute and mutates it as a `Value` instance based on the global context.
(2) **M_{t2}:** *Replace an attribute.* FD_M first selects an element and randomly removes one of its attributes. Then, FD_M applies G_{t2} to append a new attribute to the element. Here, FD_M 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. FD_M may directly invoke 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.* FD_M removes an existing CSS rule from the document and inserts a new one generated by G_{c1}.
(2) **M_{c2}:** *Mutate a CSS selector.* FD_M selects and mutates a selector in an existing rule.
(3) **M_{c3}:** *Mutate a CSS property.* FD_M selects a CSS property, and similarly mutates its value.

Event handler mutation. FD_M is also able to mutate event handlers in JavaScript. In particular, FD_M 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, FD_M runs the following three mutation methods.

Algorithm 1: 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.
Procedure mergeElement($n_a$, $n_b$, ObjectMap)
    TargetSet ← ∅;
    for each $n \in$ getOffsprings($n_a$) do
        if getType($n$) = getType($n_b$) then
            TargetSet ← TargetSet ∪ {$n$};
        end
    end
    if TargetSet = ∅ then
        // Move the sub-tree rooted at $n_b$ to be a child of $n_a$.
        insertChild($n_a$, $n_b$);
    else
        $n_t \sim$ TargetSet; // Randomly sample a node from the set.
        mergeAttributesAndText($n_t$, $n_b$);
        ObjectMap[$n_b$] ← $n_t$;
        for each $n \in$ getChildren($n_b$) do
            mergeElement($n_t$, $n$, ObjectMap);
        end
    end
Procedure mergeTree($T_a$, $T_b$, ObjectMap)
    $r_a$ ← getRoot($T_a$); $r_b$ ← getRoot($T_b$); // The tree root is the <body> element.
    for each $n_b \in$ getChildren($r_b$) do
        mergeElement($r_a$, $n_b$, ObjectMap);
    end
    for each $n_b \in T_b - \{r_b\}$ do
        if ¬∃ ObjectMap[$n_b$] then
            addElementIntoGlobalContext($n_b$);
        end
    end
```

(1) **M_{f1}:** *Insert an API call.* FD_M first chooses a particular line of the JavaScript function as the insertion point. After that, FD_M generates a new API call similar to how FD_M does in G_f. The only difference is that when FD_M queries the context for available DOM objects, all the elements in the global context are still usable. Meanwhile, only the local objects defined above 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 line number of the definition of every DOM object created below the line is incremented by one.
(2) **M_{f2}:** *Replace an API call.* FD_M first selects a random line within the event handler. The original API call at this line is removed. Then, FD_M 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 FD_M 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.* FD_M 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 contexts.

Table 2 summarizes the document mutation algorithms supported by FD_M with examples. FD_M 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, FD_M 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.

4.3.3 Document Merging. Besides mutating a single document, FD_M also supports merging two or more documents into a new document due to the effectiveness of combining existing seed inputs for testing proven by [19, 20, 54]. 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 1 presents how FD_M 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, FD_M copies every missing attribute and all the text content from n_b into n_t. In addition, FD_M uses an object map (i.e., `ObjectMap` in algorithm 1) 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, FD_M 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, FD_M records every element that originates from D_b and is directly inserted into D_a without merging in D_a's global context.
Merging CSS rules. Second, FD_M directly copies the CSS rules from D_b into D_a, which does not involve any merging conflicts.
Merging event handlers. Next, FD_M merges the event handlers in D_a and D_b. As every document in FD_M is initialized with a fixed number of event handlers, FD_M 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} in §4.3.2). Note that the relative order of any two API calls from F_b is not changed.
Fixing references. FD_M 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.

FD_M'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.

4.3.4 Mutation-based DOM Fuzzing. Based on the aforementioned algorithms, FD_M has a straightforward workflow. First, FD_M supports bootstrapping with the aid of FD_G or restarting from an old document corpus in FD-IR. FD_M is currently incapable of working with existing documents in plaintext because a transpiler that lifts them into FD-IR is missing (see §7.1 for further discussion). Given an input document in FD-IR every time, FD_M first starts its mutation phase, which is iterated N_1 times. During each fuzzing iteration, FD_M mutates the selected document M times to generate a new one for testing. Each time, one of the mutation algorithms listed in Table 2 is randomly chosen by weight. After N_1 rounds of single document mutation, if there is no progress in code coverage, FD_M merges the current document with another one randomly selected from the corpus into a new one for testing. The merging phase lasts N_2 times. All the newly generated documents during both phases that increase code coverage are saved into the corpus. Note that FD_M prefers mutation over merging to avoid a rapid growth in document size. Also, N_1, N_2, and M are all configurable and the heuristic values we use are 50, 5, and 5.

4.4 Distributed Fuzzing Environment

Due to the enormous input space and high execution cost of a browser, it is common practice to run multiple DOM fuzzer instances in a cluster. FreeDom is also designed to be cluster-friendly with a centralized network. Basically, each FreeDom instance runs on a dedicated core of a client machine and exchanges fuzzing data with a central server. During generation-based fuzzing, an FD_G instance simply transfers crash information to the server. When performing mutation-based fuzzing, the server additionally maintains the overall code coverage and a queue storing the documents in FD-IR that discover new code blocks. An FD_M instance always fetches a testcase from the head of the queue for mutation. Meanwhile, the testcase itself is moved to the tail of the queue. The FD_M instance also maintains a local coverage map that is periodically updated with the global coverage from the server. A newly generated document that increases the local coverage is uploaded to the server for verification. If the document indeed increases the overall coverage, the server pushes it to the head of the queue for mutation. In general, the fuzzing infrastructure adopted by FreeDom is lightweight and fully utilizes large-scale resources for DOM fuzzing.

5 IMPLEMENTATION

FreeDom is implemented in around 30K lines of Python3 code, which consists of three main components.
Fuzzing engine. The majority of the code is used to implement FreeDom's fuzzing engine. First, all the FD-IR structures of a document from the DOM tree and its nodes, CSS rules, event handlers and their APIs and context information down to thousands of `Values` are implemented as Python classes, which have their specific implementation of generation and mutation based on the context. In addition, FreeDom utilizes Python `Pickle` to serialize a document in FD-IR into a string for storage and deserialize the string back to FD-IR structures for mutation. FreeDom currently covers the most common DOM features supported by a modern browser, including HTML, CSS, SVG, and WebGL.
Browser instance management. FreeDom now supports fuzzing Apple Safari (WebKit), Mozilla Firefox, and Google Chrome. FreeDom compiles target browsers with AddressSanitizer (ASan) [48] for memory-error detection. To automate browser process creation and termination, FreeDom either uses existing libraries [13, 35] for Firefox and WebKit on Linux or implements custom scripts for Safari on macOS and Chrome on Windows.
Distributed fuzzing protocol. The current implementation of FreeDom involves a central server that runs Redis [46], a fast in-memory database, for storing fuzzing data. We use a Redis list to store crashes. For FD_M instances, we use an additional list to save interesting documents and a set to record the distinct code blocks that are covered in total.

Moreover, we introduce fewer than 10 lines of C++ code into the WebKit project to increase browser throughput for FD_M. Particularly, we determine the code point where the handling of an `onload` event finishes. Then, we make the browser check whether or not this happens in the main frame as the `onload` events in the embedded frames (i.e., `<iframe>`) may take place in advance. If so, we consider that the document is mostly processed but may still have ongoing or planned transitions, animations, or painting tasks. To leave a certain time for rendering those dynamic graphical effects, the browser sets an alarm clock on-the-fly, which will kill itself within a fixed amount of time. This ad-hoc timeout value used by FreeDom is 500ms considering the scale of its generated inputs. Though the optimization requires source code modification, it is technically straightforward to apply it in the same way to other mainstream browsers like Chrome and Firefox with a tiny patch.

6 EVALUATION

We evaluate FreeDom by answering four questions:

- **Q1.** How effective is FreeDom in discovering new bugs in the mainstream browsers (§6.1)?
- **Q2.** Does FreeDom become state of the art due to its context-aware approach (§6.2)?
- **Q3.** How effective is coverage-driven mutation in fuzzing DOM engines (§6.3)?
- **Q4.** How does our browser optimization impact fuzzing throughput and results (§6.4)?

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.

6.1 Discovering New DOM Engine Bugs

We have intermittently run FD_G for finding zero-day vulnerabilities in the DOM engines of all the mainstream browsers for two months. By default, FD_G fuzzes the HTML, CSS, and SVG standards together since their corresponding implementations in a browser are closely related. FD_G separately fuzzes WebGL on different OS platforms, whose implementation is independent of other browser components but involves platform-dependent code. Table 5 lists a total of 24 bugs found by FD_G 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.

6.2 Effectiveness of Context-aware Fuzzing

To prove that FreeDom is state of the art with its context-aware fuzzing approach, we compare the fuzzing performance of FD_G with that of Dharma [34] and Domato [11]. 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 (see Appendix C for the detailed build options). According to public record [11], 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/`).

6.2.1 Comparison with Dharma. We first evaluate FD_G 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 FD_G to only generate the initial DOM tree with SVG tags

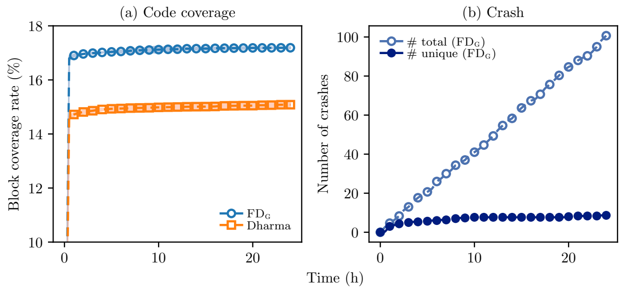


Figure 5: Achieved code coverage and triggered crashes of FD_G and Dharma when fuzzing SVG documents in WebKit with 100 cores for 24 hours. Dharma fails to find any crash during three fuzzing runs. In (b), we differentiate unique crashes found by FD_G based on their crashing `PC` values.

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 FD_G 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 presents the experimental result. FD_G visits 13.96% more code blocks than Dharma on average. More importantly, FD_G 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 FD_G's crashes by adding more random options for generating certain constant values like integers and images into its grammar. Nevertheless, around 60% of the crashes triggered by FD_G involve SVG animations. Similar to Domato discussed in §3.1, 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, FD_G 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.

6.2.2 Comparison with Domato. We then evaluate FD_G 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 FD_G to generate the documents that have complexity similar to those generated by Domato. We present the detailed composition of a random document generated by both fuzzers in Appendix D for reference. In addition, both fuzzers run with a 5-second timeout and are evaluated three times.

Figure 6 presents the evaluation results. With nearly 3% more executions, the overall code coverage of Domato is slightly higher than that of FD_G by 2.69% on average. Nevertheless, FD_G triggers around 9.7× more crashes and 3× more unique ones than Domato. In particular, FD_G discovers 112 unique crashes in three runs, 11 of which have explicit security implications reported by ASan (i.e., heap

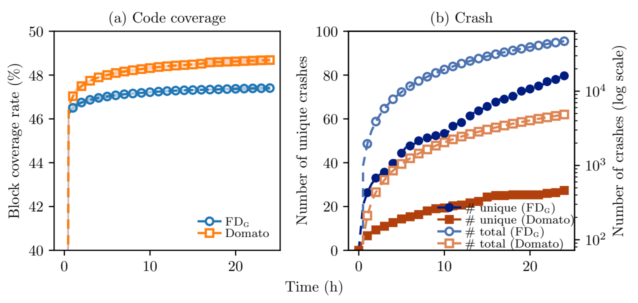


Figure 6: Achieved code coverage and triggered crashes of FD_G and Domato for a 24-hour run with 100 cores. Note that we use a log scale on the right side to present the total number of crashes. We differentiate unique crashes based on their crashing PC values.

	#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
FD_G	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 3: The unique crashes discovered by Domato, FD_G, 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 §3.1. Note that some crashes that involve more than one type of CDVs are counted more than once.

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 FD_G. 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 FD_G 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 §3.1) every PoC file involves through manual inspection, which is presented in Table 3. 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 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 FD_G are much more likely to be valid and thus FD_G 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, FD_G manages to generate a number of PoCs that cover from CDV_1 to CDV_4. To have a grasp on the context complexity of such a PoC that Domato fails to produce, we list an example in Appendix E for additional reference.

In a nutshell, our in-depth analysis shows that (1) the context-aware approach adopted by FD_G 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 FD_G for generation is effective in finding many more bugs in the latest DOM engines that have not been explored. Therefore, we believe FreeDom's design fundamentally outperforms the state-of-the-art DOM fuzzer, Domato.

6.3 Effectiveness of Coverage-driven Fuzzing

We now evaluate FD_M to understand the effectiveness of coverage guidance in DOM fuzzing. Similarly, we launch 100 FD_M instances to fuzz the optimized build of WebKitGTK 2.28.0 on 100 cores for 24 hours. The DOM engine part of WebKit is instrumented for block coverage measurement. Meanwhile, we introduce a tailored version of FD_M, called **$FD_{M\text{-}}$**, whose merging phase is removed. We run $FD_{M\text{-}}$ together with FD_M to verify the effectiveness of testcase merging (see §4.3.3) in DOM fuzzing. Both FD_M and $FD_{M\text{-}}$ bootstrap with two simple seeds: a blank document and a document with a single `<svg>` element under `<body>`. Table 4 presents the fuzzing results, and the average coverage growth is illustrated by Figure 7(a). We interpret the evaluation results as follows.

Effectiveness of merging. FD_M completely outperforms $FD_{M\text{-}}$ with 1.8% more code coverage and 1.8× more unique crashes, two of which are heap-related memory issues. By contrast, $FD_{M\text{-}}$ fails to find any security-related crashes. All of the above facts indicate that adopting document merging largely improves the performance of coverage-guided DOM fuzzing.

Effectiveness of coverage-guided mutation. Compared to FD_G, $FD_{M\text{-}}$ and FD_M visit around 1.2% and 2.62% more code blocks, respectively. More importantly, FD_M successfully discover three new crashes, including two security-related ones that are never triggered by running FD_G for 24 hours. As a generation-based fuzzer, FD_G 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, FD_M focuses on repetitive mutations and gradual growth of its inputs. Compared to FD_G, FD_M is more likely to trigger the crashes that require strict or subtle settings. For example, one crash missed by FD_G is triggered by an SVG `<text>` element that has a single attribute `x="8192em"`. The element is required to have no surrounding elements and no additional attributes or CSS styles that affect its text position, which is difficult to find in a document output by FD_G 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. FD_G 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, FD_M equipped with 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 FD_M to FD_G. With 3.5× more executions and a

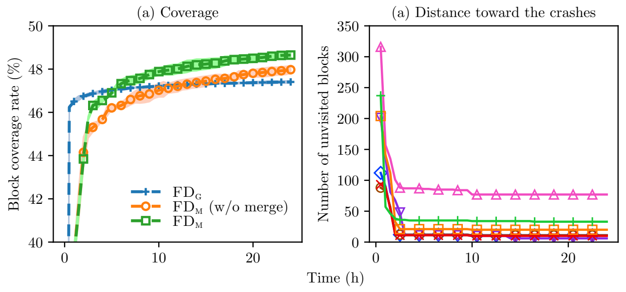


Figure 7: (a) Average block coverage rate of running FD_G and FD_M with or without merging to fuzz WebKit for 24 hours; (b) Number of basic blocks covered by the PoCs of seven security-related crashes that are not visited by FD_M at different times during a 24-hour fuzzing process.

2.62% coverage improvement, FD_M finds nearly 3.8× fewer unique crashes on average compared to FD_G. FD_M misses around 75% unique crashes, including seven security-related ones found by FD_G 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 FD_M approaches the crashes (i.e., covers those code blocks) during a fuzzing run. Figure 7(b) presents the result, which shows that FD_M is extremely close to most of the crashes after four hours. However, due to an increasing number of interesting testcases waiting to be mutated, FD_M 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 FD_G blindly generates the documents that can never explore the browser code thoroughly in process of time, at least FD_G consistently tests certain deep code paths through every execution due to the fact that a generated document has a large size and rich semantics. By contrast, code coverage makes FD_M 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 its performance can be largely improved with more computing resources and better seeding inputs (see §7.2).

6.4 High-Throughput Executions

We finally present several micro-benchmarks to evaluate the efficiency and effectiveness of our self-terminating browser.

Performance. Table 4 shows that the throughput of an optimized browser arrives at most 74.7 executions per second when used in the coverage-guided fuzzing. Considering using a 5-second hard time limit, the maximum throughput in theory is only 20 executions per second with 100 cores, which is 3.74× fewer. We also perform a stress test by running the optimized browser with large documents generated by FD_G to understand the least throughput improvement

ID	Execs/s	Coverage rate	#Crash	#Unique	#Security	#New
FD_G	18.17	47.40(±0.05)%	47058.67	79.67	11	-
FD_{M^-}	74.73	47.97(±0.18)%	68.33	11.67	0	1
FD_M	63.94	48.64(±0.13)%	331.67	21	6	3

Table 4: Fuzzing results of running FreeDom with several approaches 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.

we can achieve. In particular, FD_G fuzzes our custom WebKit build under the same setting used in §6.2. The evaluation result shows that FD_G can test around 26.92 documents per second with a dynamic alarm, which is 1.48× more than the throughput of fuzzing the original WebKit with a 5-second limit.

False negatives. We then justify that the dynamic alarm rarely makes a browser instance terminate too early to trigger an expected crash in practice. Regarding a total of 112 unique crashes found by FD_G in the original WebKit, our optimized WebKit can trigger 97% of them, including all of the 11 security-related crashes. Particularly, we observe two types of crashes that are not reproduced by the optimized WebKit. First, one missed crash involves an SVG animation with specific timing control. The PoC of the crash sets the attribute `max="1s"` for an SVG `<set>` element [51], which specifies the maximum animation duration. If we set the attribute value below `300ms`, the crash can be triggered again under the optimization. Generating relatively small timing values is considered practical to avoid such a case. The other type of missed crashes involve a `window.requestAnimationFrame(f)` call, which registers a function `f` that is invoked by the browser before repainting the window [50]. Due to the unique feature of the API, it is infeasible to control when `f` starts to execute. Therefore, we cannot find a general way to prevent such a missed crash with optimization. Based on our finding, one workaround is to use `setTimeout(f, delay)` instead, which allows us to schedule the execution of `f` to a certain extent.

7 DISCUSSION

We now discuss the limitations and future directions of FreeDom.

7.1 More Discussions on FD-IR

Limitations of FD-IR. First, FreeDom currently does not support parsing an existing HTML document into FD-IR. Developing a transpiler to achieve this is technically feasible, as FD-IR strictly follows the specification. Next, FD-IR does not cover various web APIs such as Canvas, Audio and IndexedDB APIs [27, 29, 32]. As they are also used with JavaScript, we can support them with minimal efforts. In addition, different browsers do not provide exactly the same support of certain character encodings, HTML tags, CSS functions, and DOM APIs [10, 30, 31], which are not handled by FD-IR. Thanks to its strong programmability, it is possible to make FD-IR to selectively enable or disable DOM features depending on the target browser. Lastly, FreeDom involves a great deal of engineering effort to implement numerous FD-IR `Values` (see §4.2). To minimize the manual labor, the code of the `Values` that are context-independent can be automatically emitted from existing grammars.

Generalizability of FD-IR. Besides HTML documents, FD-IR can be generalized to fuzz other complicated file formats by mainly adding new `Value` instances with custom generation, mutation, and lowering methods. FD-IR is mostly suitable for targeting the tree-like formats that consist of interdependent fields, such as PDF [6], Microsoft Office Open XML [40], Apple property list files [3], and hierarchical filesystems [2, 45]. Meanwhile, extending FD-IR to describe the API calls of other systems like RPC services and OS kernels is also practicable by adding new API formats and lowering methods for other languages or encodings besides JavaScript.

7.2 Potentials of Coverage-driven DOM fuzzing

Our performance comparison between FD_M and FD_G does not necessarily justify one approach is superior than the other. In particular, we propose future research directions for coverage-driven DOM fuzzing, which can be implemented by using FD_M.

Corpus-based fuzzing. Recent mutation-based fuzzers [20, 22, 41] achieve great success by leveraging a high-quality corpus. With respect to DOM engines, all the mainstream browsers maintain their regression and unit test suites that contain thousands of HTML documents. Once we support lifting documents in plaintext into FD-IR, FD_M is likely to have better performance by starting with these existing testcases.

Intelligent seed scheduling. FD_M now simply prefers to further mutate the testcases that are generated more recently. Nevertheless, it is worthwhile to extend FD_M with various recent testcase scheduling algorithms [8, 9, 43, 44], which are proven to be effective in discovering more deep program paths in many general applications.

Non-compliant document fuzzing. FD-IR now strictly observes the DOM standard. Nevertheless, all the mainstream browsers try to load even a non-standard document as much as possible with automatic repair in their own ways. For instance, some browsers automatically complete non-closing HTML tags in a document or still display an SVG document that involves incorrect transform functions to a certain extent. FreeDom based on FD-IR is thus unable to find the bugs in the related code. On the one hand, the design principle of FD-IR is still essential to FD_G, which will blindly produce numerous invalid documents with common errors if FD-IR's non-complicance with the standard cannot be fully handled by the browser. On the other hand, FD_M can work effectively with modified FD-IR that describes and mutates documents in a non-standard manner, because code coverage helps to self-correct its mutation directions for avoiding repetitive loading errors and effectively explore autocorrection in the browser.

8 RELATED WORK

DOM fuzzing. The pioneering DOM fuzzers [23, 28, 47, 58] embedded themselves in a page loaded by the target browser and called random DOM APIs on-the-fly. For example, domfuzz [28] randomly acts on the DOM tree of a web page with 35 fuzzing modules. Cross_fuzz [58] focuses on visiting DOM tree nodes and creating circular references among them to stress memory management. SDF [23] proposes an integration of several DOM fuzzers for better performance. All of those dynamic fuzzers are no longer maintained and the generate-and-test fuzzers have become mainstream. Qu et al. [7] find more than 100 use-after-free bugs by generating paired DOM operations based on static rules. Domato [11], Dharma [34], Avalanche [33], and Wadi [47] are recent generation-based fuzzers based on context-free grammars. FreeDom is also a static DOM fuzzer but works in a context-aware manner.

Generation-based versus mutation-based fuzzing. Generation-based fuzzing has been long established, especially for a structural input format not limited to HTML documents. For example, Peach [42] is known for generating the inputs of any format that can be expressed as a grammar definition. Jsfunfuzz [38] constructs random code chunks for testing JavaScript engines. Csmith [56] generates random C source text for stress-testing compilers. On the other side, more and more fuzzers nowadays also mutate existing testcases. A series of JavaScript engine fuzzers [4, 19, 21, 53, 54] transform existing JavaScript programs into new ones for testing. Syzkaller [18] mutates system calls for fuzzing OS kernels. As a font fuzzer, BrokenType can generate valid TTF files and also mutate them in a structure-aware manner. In the DOM fuzzing field, FreeDom is considered the first static fuzzer to work both generatively and mutationally guided by coverage.

Structure-aware versus context-aware fuzzing. To generate valid payload for testing, smart fuzzers are aware of input format or grammar. For example, the aforementioned DOM and JavaScript engine fuzzers always produce syntactically correct HTML files and JavaScript programs. Nevertheless, recent research reveals the importance of leveraging context information for reducing semantic errors in generated inputs. For example, CodeAlchemist [20] assembles JavaScript code with guaranteed type correctness. Janus [55], a filesystem fuzzer, maintains the status of every file and directory on an image for generating valid file operations.

9 CONCLUSION

In this work, we propose FreeDom, an evolutionary static DOM fuzzer, that supports fuzzing HTML documents generatively and mutationally. FreeDom relies on a newly designed IR to describe both structures and context information of a document so as to avoid semantic errors. In addition, FreeDom also utilizes a dynamic timeout mechanism to largely improve the browser throughput by 1.48–3.74×. We have reported 24 bugs found by FreeDom in mainstream browsers. In particular, FreeDom that adopts context-aware generation can find 3× more unique crashes than the state-of-the-art fuzzer, Domato. FreeDom also finds several bugs in WebKit's SVG renderer, where another recent fuzzer, Dharma, fails to trigger any crash. Compared to blackbox generation, FreeDom guided by coverage covers 2.62% more code blocks and discovers several new bugs, but meanwhile triggers a total of 3.8× fewer crashes. Nevertheless, we expect further research based on FreeDom for an improved utilization of code coverage in DOM fuzzing.

10 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware, and Google.

REFERENCES

[1] Aldeid. Bf3. https://www.aldeid.com/wiki/Bf3 (visited on September 12, 2020).

[2] Apple Inc. HFS Plus Volume Format. https://developer.apple.com/library/archive/technotes/tn/tn1150.html (visited on September 12, 2020).

[3] Apple Inc. Property List Programming Topics for Core Foundation. https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html (visited on September 12, 2020).

[4] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2019).

[5] Beterke, F., Geshev, G., and Plaskett, A. Apple Safari - PWN2OWN Desktop Exploit. https://labs.f-secure.com/assets/BlogFiles/apple-safari-pwn2own-vuln-write-up-2018-10-29-final.pdf (visited on September 12, 2020).

[6] Bienz, T., Cohn, R., and Systems, A. *Portable document format reference manual.* Citeseer, 1993.

[7] Bo, Q., and Lu, R. POWER IN PAIRS: How one fuzzing template revealed over 100 IE UAF vulnerabilities. In *Black Hat USA Briefings (Black Hat USA)* (Amsterdam, The Netherlands, Oct. 2014).

[8] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)* (Dallas, TX, Oct.–Nov. 2017).

[9] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the IEEE Transactions on Software Engineering* (2017).

[10] Deveria, A. WebGL 2.0. https://caniuse.com/#feat=webgl2 (visited on September 12, 2020).

[11] Fratric, I. DOM fuzzer. https://github.com/googleprojectzero/domato (visited on September 12, 2020).

[12] Fratric, I. The Great DOM Fuzz-off of 2017. https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html (visited on September 12, 2020).

[13] Fratric, I. WebKit Fuzzing. https://github.com/googleprojectzero/p0tools (visited on September 12, 2020).

[14] Google. Chrome Vulnerability Reward Program Rules. https://www.google.com/about/appsecurity/chrome-rewards/index.html (visited on September 12, 2020).

[15] Google. ClusterFuzz. https://google.github.io/clusterfuzz (visited on September 12, 2020).

[16] Google. Issue 666246. https://bugs.chromium.org/p/chromium/issues/detail?id=666246 (visited on September 12, 2020).

[17] Google. Issue 671328. https://bugs.chromium.org/p/chromium/issues/detail?id=671328 (visited on September 12, 2020).

[18] Google. syzkaller is an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller (visited on September 12, 2020).

[19] Gross, S. Fuzzil: Coverage guided fuzzing for javascript engines. Master's thesis, TU Braunschweig, 2018.

[20] Han, H., Oh, D., and Cha, S. K. 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).

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

[22] Lee, S., Han, H., Cha, S. K., and Son, S. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *Proceedings of the 29th USENIX Security Symposium (Security)* (Boston, MA, Aug. 2020).

[23] Lin, Y.-D., Liao, F.-Z., Huang, S.-K., and Lai, Y.-C. Browser fuzzing by scheduled mutation and generation of document object models. In *Proceedings of the 49th IEEE International Carnahan Conference on Security Technology* (Taipei, Taiwan, Sept. 2015).

[24] Liu, J., and Xu, C. Pwning Microsoft Edge Browser: From Memory Safety Vulnerability to Remote Code Execution. POC.

[25] LLVM Project. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html (visited on September 12, 2020).

[26] 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/ (visited on September 12, 2020).

[27] Mozilla. Canvas API. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (visited on September 12, 2020).

[28] Mozilla. DOM fuzzers. https://github.com/MozillaSecurity/domfuzz (visited on September 12, 2020).

[29] Mozilla. IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on September 12, 2020).

[30] Mozilla. Localizations and character encodings. https://developer.mozilla.org/en-US/docs/Web/Guide/Localizations_and_character_encodings (visited on September 12, 2020).

[31] Mozilla. Vendor Prefix. https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix (visited on September 12, 2020).

[32] Mozilla. Web Audio API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API (visited on September 12, 2020).

[33] Mozilla Security. Avalanche. https://github.com/MozillaSecurity/avalanche (visited on September 12, 2020).

[34] Mozilla Security. dharma. https://github.com/MozillaSecurity/dharma (visited on September 12, 2020).

[35] Mozilla Security. FFPuppet. https://github.com/MozillaSecurity/ffpuppet (visited on September 12, 2020).

[36] Mozilla Security. Grizzly Browser Fuzzing Framework. https://blog.mozilla.org/security/2019/07/10/grizzly (visited on September 12, 2020).

[37] Mozilla Security. Introducing the ASan Nightly Project. https://blog.mozilla.org/security/2018/07/19/introducing-the-asan-nightly-project/ (visited on September 12, 2020).

[38] Mozilla Security. JavaScript engine fuzzers. https://github.com/MozillaSecurity/funfuzz (visited on September 12, 2020).

[39] Mozilla Security. Writing an Adapter. https://github.com/MozillaSecurity/grizzly/wiki/Writing-an-Adapter (visited on September 12, 2020).

[40] Paoli, J., Valet-Harper, I., Farquhar, A., and Sebestyen, I. Ecma-376 office open xml file formats. http://www.ecmainternational.org/publications/standards/Ecma-376.htm (visited on September 12, 2020).

[41] Park, S., Xu, W., Yun, I., Jang, D., and Kim, T. Fuzzing JavaScript Engines with Aspect-preserving Mutation (to appear). In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2020).

[42] Peach Tech. Peach Fuzzer. https://sourceforge.net/projects/peachfuzz (visited on September 12, 2020).

[43] Pham, V.-T., Böhme, M., Santosa, A. E., Caciulescu, A. R., and Roychoudhury, A. Smart greybox fuzzing. In *Proceedings of the IEEE Transactions on Software Engineering* (2019).

[44] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (Security)* (San Diego, CA, Aug. 2014).

[45] Rodeh, O., Bacik, J., and Mason, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 1–32.

[46] Sanfilippo, S. Redis, Open source in-memory database, cache and message broker. https://redis.io/ (visited on September 12, 2020).

[47] SensePost. Wadi Fuzzing Harness. https://github.com/sensepost/wadi (visited on September 12, 2020).

[48] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2012).

[49] Veditz, D. Fixing an SVG Animation Vulnerability. https://blog.mozilla.org/security/2016/11/30/fixing-an-svg-animation-vulnerability/ (visited on September 12, 2020).

[50] W3C. HTML: 8.11 Animation frames. https://html.spec.whatwg.org/multipage/imagebitmap-and-animations.html#animation-frames (visited on September 12, 2020).

[51] W3C. SVG Animations Level 2: 2.14. The 'set' element. https://svgwg.org/specs/animations/#SetElement (visited on September 12, 2020).

[52] W3C. SVG Animations Level 2: Attributes to identify the target attribute or property for an animation. https://svgwg.org/specs/animations/#AttributeNameAttribute (visited on September 12, 2020).

[53] Wang, J., Chen, B., Wei, L., and Liu, Y. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*

(Oakland) (San Jose, CA, May 2017).

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

[55] Xu, W., Moon, H., Kashyap, S., Tseng, P.-N., and Kim, T. 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).

[56] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, CA, June 2011).

[57] Zalewski, M. american fuzzy lop (2.52b). http://lcamtuf.coredump.cx/afl (visited on September 12, 2020).

[58] Zalewski, M. cross_fuzz. https://lcamtuf.coredump.cx/cross_fuzz/ (visited on September 12, 2020).

#	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: 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.

A DOM ENGINE BUGS FOUND BY FREEDOM

Table 5 presents the description of the bugs discovered by FreeDom in the DOM engines of three known web browsers (i.e., Apple Safari, Google Chrome and Mozilla Firefox).

B THE GENERATION ENGINE OF DOMATO

Figure 8 serves as an examples of how Domato generates HTML documents based on its grammar. Basically, the listed grammar rules are essential for generating the CSS rules, JavaScript code, and initial DOM tree below. Note that the demonstrated document is incomplete. The CSS rule originally owns more CSS properties. And both the `<form>` and `<select>` are supposed to have more attributes and children.

```
<!--
<rule> = <selector> { <declaration> }
<selector> = .<class>
<class> = class<int min=0 max=9>
<declaration> = <cssproperty>; <cssproperty>; ...
<cssproperty> = columns: <cssproperty_columns>
<cssproperty_columns> = <fuzzint>
-->
<style>
.class1 { columns: 1246; }
</style>

<!--
<HTMLSelectElement>.autofocus = <boolean>;
<boolean> = true
<boolean> = false

<new boolean> = <HTMLSelectElement>.reportValidity();
-->
<script>
function jsfuzzer() {
  htmlvar00002.autofocus = true;
  var var00001 = htmlvar00002.reportValidity();
}
</script>

<!--
<HTMLFormElement> = <lt>form <form_attributes>
        <attributes><gt><formchildren><lt>/form<gt>
<attributes> = <attribute> <attribute> <attribute>
        <attribute> <attribute>
<attribute> = <attribute_class>
<attribute_class> = class="<class_value>"
<class_value> = <class>
<class> = class<int min=0 max=9>
<formchildren> = <newline><formchildelement><newline>
<formchildelement> = <HTMLSelectElement>

<HTMLSelectElement> = <lt>select <select_attributes>
        <attributes><gt><selectchildren><lt>/select<gt>
<attribute> = <attribute_eventhandler>
<attribute_eventhandler> = <attribute_onblur>
<attribute_onblur> = onblur="<eventhandler>"
<eventhandler> = eventhandler1()
<eventhandler> = eventhandler2()
<eventhandler> = eventhandler3()
<eventhandler> = eventhandler4()
<eventhandler> = eventhandler5()
<selectchildren> = <optionelements>
<optionelements> = <htmlsafestring min=32 max=126>
-->
<body onload="jsfuzzer()">
<form id="htmlvar00001" class="class1">
  <select id="htmlvar00002" onblur="eventhandler2()">
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  </select>
</form>
</body>
```

Figure 8: An example of how Domato randomly generates an HTML document.

C WEBKIT BUILD OPTIONS

We refer to the build options Domato uses to compile WebKit on Linux for our evaluation. The detailed command line is listed in Figure 9.

```
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=.
-DCMAKE_SKIP_RPATH=ON -DPORT=GTK -DUSE_LIBHYPHEN=OFF
-DENABLE_MINIBROWSER=ON -DUSE_SYSTEM_MALLOC=ON -DUSE_WPE_RENDERER=OFF
-DENABLE_INTROSPECTION=OFF -DENABLE_SPELLCHECK=OFF
-DENABLE_BUBBLEWRAP_SANDBOX=OFF -DENABLE_GTKDOC=OFF
-DUSE_GSTREAMER_GL=OFF -DENABLE_MEDIA_STREAM=OFF
-DENABLE_VIDEO=OFF -DENABLE_WEB_AUDIO=OFF -DENABLE_GEOLOCATION=OFF
-DUSE_LIBSECRET=OFF -DENABLE_WEB_RTC=OFF -DUSE_LIBNOTIFY=OFF
-DENABLE_WEB_CRYPTO=OFF -DUSE_WOFF2=OFF -Wno-dev
```

Figure 9: The WebKit build options used in evaluation.

D TESTCASE COMPLEXITY OF FREEDOM AND DOMATO

To have a fair comparison between FreeDom and Domato (see §6.2.2), we use the original Domato and configure FreeDom to make a random document generated by both fuzzers generally have:

- a tree of 60 HTML and SVG elements on average;
- 50 CSS rules, each of which has two selectors on average and 20 CSS properties;
- five event handlers besides `main()`. `main()` has 1,000 lines of DOM API calls, while every other event handler only has 500. Every event handler installs a counter check at the beginning to ensure that it is never executed for more than two times.

E A POC GENERATED BY FREEDOM

```
1  <style>
2  *::first-letter { -webkit-justify-content: space-between; }
3  #v22 { content: url(#foo); }
4  * { -webkit-filter: url(#v21); -webkit-border-radius: inherit }
5  </style>
6
7  <script>
8  function main() { document.bgColor = "rgba(40,125,181,209)"; }
9  function f3() { var v62 = v22.getBoundingClientRect(); }
10 </script>
11
12 <body onload="main()">
13 <svg id="v9" xmlns="http://www.w3.org/2000/svg">
14 <defs id="v10"></defs>
15 <set id="v20" attributeName="preserveAspectRatio" onbegin="f3()"></set>
16 <filter id="v21"></filter>
17 <rect id="v22" y="32" rx="32" height="82%" x="32"></rect>
18 </svg>
19 </body>
```

Figure 10: The PoC snippet of a bug in WebKit found by FreeDom.

In Figure 10, we list the PoC of a crash found by FreeDom that has various context-dependent values (CDVs) and is thus missed by Domato in our evaluation. The PoC includes (1) `#v22` at Line 3, which is CDV$_1$, (2) `-webkit-filter: url(#v21)` at Line 4, which is CDV$_2$, and (3) `attributeName="preserveAspectRatio"` at Line 15, which is CDV$_3$.