Karlsruhe Institute of Technology

FuzzIL: Coverage Guided Fuzzing for JavaScript Engines

Masters Thesis

Samuel Groß

Institute of Theoretical Informatics
Competence Center for Applied Security Technology
Department of Informatics
Karlsruhe Institute of Technology

Supervising Professors: Prof. Dr. Jörn Müller-Quade, KIT
Prof. Dr. Ralf Reussner, KIT
Prof. Dr. Martin Johns, TU Braunschweig

Time for Completion: 01. 05. 2018 – 01. 11. 2018

I assure truthfully, that I wrote this thesis independently, that I have indicated all used aids completely and exactly and that I have marked everything that has been taken from the work of others unchanged or with changes and that I have observed the statutes of the KIT to ensure good scientific practice in the respective valid version.

Karlsruhe, the 29th of October 2018

Abstract

Web Browsers have become a ubiquitous application on desktop operating systems as well as on mobile platforms, making them critical to users' security on the web. In particular JavaScript interpreters, which allow for an interactive experience on the web, have come into focus for attackers as they commonly provide powerful exploitation primitives that enable the compromise of the hosting process with a single vulnerability. As such there is an interest to detect these vulnerabilities in an automatic fashion.

Besides static analysis, fuzzing is commonly used to automatically detect software vulnerabilities. Here, randomly generated input is repeatedly provided to the application as input in the hopes of triggering a faulty condition. Recently, guided fuzzing approaches have shown promising results and often outperform classic mutation-based or generative approaches. Guided fuzzing uses feedback, often in the form of coverage information, collected during execution to score the produced samples. New samples will then be produced by mutating previous samples that were deemed interesting. This enables the fuzzer to eventually generate complex inputs that reach deep into the targets processing logic.

The goal of this work is then to apply the guided fuzzing approach to script interpreters. This requires defining semantically useful mutations on program code. However, in contrast to existing approaches that mutate syntactical elements such as the textual representation of code or a syntax tree, mutations are defined on a newly designed intermediate language (IL) on which, due to its restricted form, mutations of control and data flow can easily be performed. This reflects the fact that the syntactical properties of a program are generally irrelevant. Further, the proposed system is able to generate semantically valid samples with a high probability. This is important as it avoids the need to wrap generated code into try-catch constructs which can negatively affect the runtime behavior.

An initial implementation of the proposed system was able to identify exploitable, previously unknown security vulnerabilities inside the JavaScript engines of the Safari and Firefox web browsers as well as numerous crashes and internal assertion failures across different engines.

Zusammenfassung

Die Sicherheit von Web Browsern ist nach wie vor kritisch für Endnutzer des Internets. Software-Schwachstellen in diesen können ausgenutzt werden, um unerlaubt Zugriff auf das jeweilige Endgerät, und damit auf private Daten der Nutzer zu erlangen. Insbesondere JavaScript Interpreter, welche JavaScript Programme ausführen um interaktive Webseiten zu ermöglichen, sind in den Fokus von Sicherheitsforschern gerückt. Dies ist unter anderem der hohen Komplexität der Software sowie der Tatsache, dass viele der dort gefundenen Schwachstellen zuverlässig ausnutzbar sind, zuzuschreiben. Aus diesem Grund besteht ein Interesse, diese Art von Programmierfehler zu finden und zu beheben.

Generell lassen sich Software-Schwachstellen sowohl durch manuelle Analyse des Quelltextes, als auch mithilfe automatisierter Techniken auffinden. Letztere haben dabei den Vorteil, dass sie besser skalierbar sind. Diese Arbeit widmet sich daher automatisierten Ansätzen, insbesondere dem Fuzzing. Hierbei werden wiederholt zufällige Eingabedaten für das getestete Program erstellt und zur Verarbeitung an das Program gegeben. Die Ausführung des Programmes wird anschließend auf eventuell auftretendes Fehlverhalten, wie z.B. ungültige Speicherzugriffe, überwacht. Grundsätzlich gibt es zwei weit verbreitete Fuzzing Ansätze: Mutationsbasierte Ansätze sowie generative Ansätze. Bei Ersteren werden kleine Änderung, wie z.B. das Kippen einzelner Bits, auf Eingaben aus einem bestehenden Korpus an Daten angewendet, um aus diesen neue Eingaben zu erzeugen. Bei generativen Ansätzen wird dagegen jedes mal eine neue Eingabe erzeugt. Dies kann z.B. mithilfe einer vordefinierten Grammatik passieren.

In dieser Arbeit wird ein neuer Ansatz für das Fuzzing von JavaScript Interpretern vorgestellt. Er beruht darauf, eine neue Zwischensprache, genannt FuzzIL, für JavaScript zu entwickeln, auf welcher sich diverse Mutationen definieren lassen. Diese arbeiten nicht wie bisherige Ansätze auf einem syntaktischen Level sondern auf einem Daten- und Kontrollfluss Level. Diese Mutationen werden dann genutzt, um mithilfe des sogenannten "Guided Fuzzing" Ansatzes, im Grunde einem genetischen Algorithmus, neue Eingaben zu entdecken, welche vorher unerreichte Teile des getesteten Programmes abdecken. Neben dem Übersetzen von FuzzIL in die Zielsprache sind weitere Schritte notwendig. Diese beinhalten das Messen von Programmabdeckung um eine Metrik für die Güte eines erzeugten Programmes zu haben, sowie das Minimieren von interessanten Programmen. Dies ist nötig da die definierten Mutationen ein Program nur vergrößern können, nie verkleinern.

Der beschriebene Ansatz wurde in der Swift Programmiersprache umgesetzt und ausgewertet. Es gelang dabei vier bisher unbekannte Schwachstellen in den weit verbreiteten Web Browsern Safari und Firefox zu entdecken. Weiterhin wurden mehrere fehlerhafte Konditionen in diesen und weiteren JavaScript Interpretern gefunden.

Contents

1. Introduction

Web Browsers continue to be critical for end user's security on the web, as vulnerabilities therein can be exploited by attackers to gain access to end user devices and thus personal data stored on them. Specifically JavaScript engines have moved into the focus of security researchers for various reasons: first, they are often implemented in memory unsafe languages for performance reasons, making them susceptible to memory corruption vulnerabilities. Second, they have become highly complex software, increasing the chances for programming mistakes. This is in part due to the continuing race for performance among browser vendors. As an example, they commonly contain multiple different just-in-time (JIT) compilers which compile JavaScript code to machine code at runtime to boost performance. Last but not least, vulnerabilities found in these engines often provide powerful exploitation primitives, allowing the compromise of the hosting process with a single vulnerability despite modern exploit mitigation technologies.

A public example of an attack featuring security vulnerabilities in a web browser is the Pegasus spyware[1]. Here, a vulnerability in the JavaScript engine powering the Safari browser, as well as two vulnerabilities in the iOS system, were exploited to remotely gain control of the device. Furthermore, the yearly pwn2own[2] and similar contests continue to demonstrate the feasibility of such attacks. As such, there is a clear incentive to discover these types of vulnerabilities.

Besides manual code auditing and static analysis, fuzz-testing, or fuzzing, is applicable to detect programming errors and thus software vulnerabilities. During fuzzing, random input is produced and provided to the tested program for processing. The program's execution is then monitored for misbehavior such as memory access violations. As fuzzing works in an automatic fashion and is easy to scale it remains a relevant research topic. Recently, guided fuzzing, as for instance implemented by afl [Zal15], has been explored as well. Here, feedback in the form of coverage information is collected and used to score the produced samples. Samples that increase coverage are then kept for future mutations while other samples are discarded. This enables the fuzzer to uncover vulnerabilities deep in

[1] https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/
[2] https://www.thezdi.com/blog/2018/3/15/pwn2own-2018-day-two-results-and-master-of-pwn

the programs processing logic. However, it is non-trivial to apply guided fuzzing to script interpreters as it requires the definition of sensible mutations on program code.

This work aims to research how guided fuzzing can be implemented for scripting engines. One key contribution lies in the proposal of new mutations for program code which work on a different level, closer to the data and control flow of a program, than existing mutational approaches which usually target syntactical elements such as the abstract syntax tree (AST) of a program.

This work is then structured as follows. First, chapter 2 provides a technical context for this work. In particular, an overview of web browsers and JavaScript engines, as well as commonly found vulnerabilities, is provided. Furthermore, fuzz-testing, a technique to discover software defects automatically, is introduced. Afterwards, chapter 3 defines the goal of this work and presents a list of requirements which the proposed system should meet to achieve said goal. Chapter 4 then introduces the newly proposed approach for guided fuzzing of JavaScript engines and is followed by chapter 5, which discusses some aspects of the implementation of the proposed system. The presented approach is evaluated in chapter 6, where a number of previously unknown security vulnerabilities will be discussed. Finally, chapter 7 discusses relevant related work and is followed by chapter 8, which summarizes this work and provides suggestions for future improvements.

2. Background

This chapter aims to give an introduction to web browsers and JavaScript engines from a security point of view as well as fuzz-testing, a technique used for automatic vulnerability discovery.

The chapter begins with a brief overview of a web browser, then continues with a discussion of the major components that make up a JavaScript engine. Next, common vulnerabilities in JavaScript engines will be discussed at the example of CVE-2016-4622 and CVE-2018-4233, two vulnerabilities discovered through manual code auditing. Afterwards, fuzz-testing will be introduced. The two common approaches to fuzz-testing, namely mutation-based and generative fuzzing will be described and finally guided fuzzing, a scheme that involves feedback of some sort to produce complex input samples, will be discussed.

2.1 Web Browser Overview

A web browser's task is to retrieve and display resources such as HTML web pages identified by a URL while also processing user input. To allow for an interactive web experience, web browsers commonly support scripting of web pages through the JavaScript language.

A typical web browser today is made up of different processes: a browser process and one or more renderer processes, as well as possibly additional processes for managing resources such as the network or the graphics processing unit (GPU). The browser process displays the user interface and manages the renderer processes, while the renderer processes parse HTML, CSS, a multitude of media formats such as images and videos, and various other data formats, and combine them into a document which is then presented to the user. JavaScript needs to be parsed and executed as well. In addition to the core language features, a wide range of browser APIs are available to JavaScript code to allow web developers to interact with the current document, to use web databases, to retrieve data over the network, and more. As some features, for instance web databases, require persistent storage, they are managed by the browser process and made available to renderer processes through IPC.

From a security standpoint, the renderer process is the most exposed component of a browser as it handles a multitude of untrusted and potentially complex inputs. As such it

is commonly *sandboxed*, meaning it has limited access to sensitive data such as a user's personal data. For that reason a second vulnerability in a component that is not sandboxed, but which is reachable from the sandboxed process, is commonly needed to gain full access to the system. This vulnerability could be inside the browser process or the operating system. However, initial code execution inside the renderer process is in most cases a necessity to exploit these vulnerabilities and escalate privileges. Moreover, a compromised renderer is in some cases able to perform cross-origin attacks, also known as universal cross-site scripting (UXSS). With these, an attacker is able to gain access to web authentication data, mainly cookies, and subsequently gain access to a victims online activities such as emails, cloud storage, social networks, banking, and more. As such, the integrity of the renderer process remains critical to the overall system security.

While clearly not the only relevant attack surface, JavaScript engines are becoming the preferred way to gain initial code execution inside a renderer process. This is partially due to the powerful exploitation primitives gained from vulnerabilities herein, as further discussed in 2.3. As such the focus of this work lies in detecting vulnerabilities in the JavaScript engine itself, although the approach can be generalized to target web APIs reachable through JavaScript code.

2.2 JavaScript Engine Overview

The task of a JavaScript engine is to parse and execute JavaScript code. In contrast to most other scripting environments, a JavaScript engine embedded in a web browser is expected to safely process untrusted scripts. Further, it is developed with an extensive focus on execution performance to enable interactive, client-side web applications. As is often the case, increased performance comes with increased complexity of the code base, which in turn leads to programming mistakes that are sometimes security critical.

The following JavaScript engines are readily available as open source projects and power the most prominent web browsers to date:

- Spidermonkey, the engine inside the Firefox browser, is available as part of the gecko browser engine. It is developed using mercurial[1], but a git mirror is also available[2]
- JavaScriptCore (JSC), the engine inside WebKit and thus Safari, is available as part of the WebKit project, which is developed using subversion[3]. A git mirror is available as well[4]
- V8, the engine inside the Chrome browser, is available as part of the Chromium open source project[5] as well as standalone[6]
- Chakra, the engine inside the Edge browser, is available as a standalone project on github[7]

[1] https://hg.mozilla.org/mozilla-central/
[2] https://github.com/mozilla/gecko-dev
[3] https://trac.webkit.org
[4] https://github.com/WebKit/webkit
[5] https://chromium.googlesource.com/
[6] https://github.com/v8/v8
[7] https://github.com/Microsoft/ChakraCore

A modern JavaScript engine consists at least of the following components: a parser and bytecode compiler, an interpreter and possibly one or more Just-in-Time (JIT) compilers, a runtime environment, and a garbage collector. Each of them will be discussed next.

2.2.1 Parser and Bytecode Compiler

The parser and bytecode compiler are responsible for converting JavaScript code from its source code representation to an engine-specific *bytecode* which is then used by the interpreter and JIT compiler. Parsing happens by first tokenizing the input stream, then constructing an abstract syntax tree (AST). The AST is subsequently compiled to bytecode by a bytecode compiler [AU77].

The bytecode format differs between the engines. While some engines such as Spidermonkey use a stack-based virtual machine, other engines, for example v8, use a register-based virtual machine. As such, their bytecode format is fundamentally different. However, one property common to the different bytecode formats is that they are typically untyped: the bytecode operates on dynamically-typed values which contain both type- and value information. During execution of the bytecode in an interpreter, different actions will then be performed depending on the runtime type of the operands.

Another common property of the resulting bytecode is that it is usually unoptimized. This is due to several reasons: first, it is desirable to keep the overall startup time as low as possible, forbidding the use of costly optimizations. Second, many optimizations require type information, which is not available at this point due to the dynamically-typed nature of the JavaScript language.

2.2.2 Interpreter

The task of an interpreter is to consume the bytecode and execute it. As the bytecode is custom to each engine, so is the interpreter. Interpretation happens by fetching the next bytecode instruction from the currently executing code and dispatching it to a handler for the bytecode operation. An example of a bytecode handler for the the `ADD` operation in the Spidermonkey engine is shown in listing 2.1. The virtual machine is stack based. As such the operation essentially pops the first two values from the stack, adds them together, and places the result on the stack again.

```
1  CASE(JSOP_ADD)
2  {
3      MutableHandleValue lval = REGS.stackHandleAt(-2);
4      MutableHandleValue rval = REGS.stackHandleAt(-1);
5      MutableHandleValue res = REGS.stackHandleAt(-2);
6      if (!AddOperation(cx, lval, rval, res))
7          goto error;
8      REGS.sp--;
9  }
10 END_CASE(JSOP_ADD)
```

Listing 2.1: The handler code for the `JSOP_ADD` bytecode operation inside the Spidermonkey interpreter.

While interpretation of bytecode is fairly slow, in part due to the unoptimized nature of the bytecode as previously discussed as well as the dispatching overhead for every operation, the initial startup time is very low. As such, for code that is only executed once or a few times, it is overall better to execute it in the interpreter, as the overhead of compilation would far outweigh the faster execution speed of the resulting machine code. However, if a unit of JavaScript code is repeatedly executed by the application it is worth optimizing that code. This is the job of a JIT compiler. As current JIT compilers usually require type hints to be able to produce optimized machine code, it is also the job of the interpreter to collect type profiles during execution of the bytecode. These commonly augment the bytecode with previously seen input types for each bytecode operation. As an example, a type profile for the Spidermonkey bytecode would, for an operation such as `JSOP_ADD`, shown in listing 2.1, associate one set of observed types with the first input to the operation and a second set of types with the second input.

2.2.3 JIT Compiler

A JIT compiler acts as an alternative to an interpreter for script execution. It, too, usually consumes bytecode by the parser, as well as type profiles gathered during execution by the interpreter. Having access to both, it converts the bytecode to optimized machine code which can be executed directly on the host CPU.

As JIT compilers are crucial to execution performance, they are the subject of intensive research. Many implementations and key mechanisms have been discussed, e.g. by Gal et al. [GES+09] and Hackett et al. [HG12] who presented mechanisms used by the Spidermonkey JIT compiler to generate optimized machine code and deal with missing type information. Moreover, Hölzle et al. [HCU91] introduced polymorphic inline caches, a broadly used mechanism to speed up polymorphic operations in dynamic language interpreters and at the same time gather type information for the JIT compiler by caching results of previous executions. Further, numerous online resources on the topic were published, such as can for example be found on the github page for the turbofan compiler[8], the JIT compiler inside v8. What follows is a brief, simplified, and high-level summary of the inner workings of current JIT compilers for the JavaScript language.

A JIT compiler will commonly first convert the bytecode to another custom intermediate representation. This IR is often graph based to facilitate various optimizations that are later performed on it. In addition, most engines currently use a static single assignment (SSA) form [CFR+91] to further simplify code analysis and optimization.

One essential mechanism that allows generation of performant machine code for JavaScript and dynamically typed languages in general are *speculations*: equipped with type profiles from the interpreter, the compiler speculates that the same types will be used in the future. It will then guard these assumptions with runtime guards: small fragments of code that perform an inexpensive type check and *bailout* if the check fails, in which case execution will continue in a more generic execution tier such as the interpreter. These type guards essentially convert the previously untyped code into strictly typed code, which can subsequently be optimized in a similar fashion as other strictly-typed languages such as C++ or Java. This mechanism is shown in listings 2.2, 2.3, and 2.4 with an imaginary bytecode and compiler IR format. Here, the function to be compiled, which is shown in

[8] https://github.com/v8/v8/wiki/TurboFan

listing 2.2, is invoked with integers as arguments. This is captured by the interpreter in type profiles associated with the bytecode as shown in listing 2.3. Based on those, the JIT compiler speculates that the same types will be used in the future and guards that assumption with two type checks. Afterwards, the compiler can use fast, specific operations, such as the addition of two integers, instead of slow, generic ones that cover all possible scenarios of different types. This can be seen in listing 2.4. The integer addition used in this example could well be implemented with a single machine instruction instead of the generic addition operation as defined by the language specification[9].

```
1 function add(a, b) {
2     return a + b;
3 }
4
5 for (let i = 0; i < 10000; i++) {
6     add(i, 42);
7 }
```

Listing 2.2: Example for the compilation of a JavaScript function by a JIT compiler.

```
1 function add:
2     PushArg arg0       ; observed types: {Integer}
3     PushArg arg1       ; observed types: {Integer}
4     Add                ; observed types: {Integer}, {Integer}
5     Return
```

Listing 2.3: Exemplary interpreter bytecode with type profile for the JavaScript code given in listing 2.2.

```
1 function add:
2     v0 = LoadArgument 0
3     CheckIsInteger v0
4     v1 = LoadArgument 1
5     CheckIsInteger v1
6     v2 = IntegerAdd v0, v1
7     Return v2
```

Listing 2.4: Exemplary compiler IR for the bytecode given in listing 2.3. The compiler used the type profiles to emit type checks and a specialized instruction to perform an integer addition.

It is not uncommon for an engine to have multiple levels of JIT compilers. These correspond to different optimization levels: the early JIT compilers perform less optimizations, thereby produce machine code faster. The late JIT compiler stages perform more optimizations, thereby generating faster machine code. A unit of code may be recompiled by a higher

[9] https://www.ecma-international.org/ecma-262/8.0/#sec-addition-operator-plus

level JIT compiler when its execution count reaches another threshold. JavaScriptCore, the engine inside WebKit currently features three different JIT compilers in addition to an interpreter. V8, the engine inside the Chrome browser, on the other hand, only uses one JIT compiler and one interpreter.

2.2.4 The Runtime

The runtime environment provides algorithms and data structures needed during the execution of JavaScript code as well as a number of *builtin* objects and functions as dictated by the language specification[10]. Examples of the former include the implementation of JavaScript values, objects, strings, and arrays which are defined by the language specification, as well as internal helper objects such as hash maps. Examples for builtin objects and functions include the mathematical functions that are defined on the global `Math` object or functions related to arrays installed on the array prototype, which are thus accessible to every regular array object and through the `prototype` property on the global `Array` object.

2.2.5 Garbage Collector

As JavaScript frees the programmer of the responsibility to return allocated memory back to the system, it requires a garbage collector (GC) to detect and reclaim unused memory.

Most high-performance garbage collector systems are based on the mark-and-sweep algorithm [JL96], which operates in two phases. First, during the periodic marking phases, the whole graph of reachable objects is scanned. Then, during the sweeping phase, every memory allocation which was not visited, implying that it is a dead object, is freed and returned to the allocator.

Various extensions of this core algorithm exist and are in use today. One key improvement lies in concurrent marking to avoid noticeable pauses in the application caused by ”stop-the-world” garbage collection. An example of such a collector was suggested by Doligez et al. [DL93, DG94]. Modern collectors such as Riptide, the collector used in JavaScriptCore[11], further strive for a minimal amount of application interrupts due to garbage collection through parallelism if multiple CPU cores are available.

Another variant, described by Baker [BJ78] and later used by Halstead to implement Multilisp [HJ84], uses two memory regions to improve sweeping: an active one in which new allocations are placed and an inactive one. After marking has finished, live objects are ”evacuated” to the inactive region, after which the first region is declared free and the active and inactive region are swapped. This algorithm performs well if most allocated objects are dead, as it has zero cost for cleaning up dead objects. Orinoco the collector used by v8[12], uses a scheme similar to Halstead’s collector.

2.3 JavaScript Engine Vulnerabilities

JavaScript engines have come into focus of security researchers in the last years. There are several reasons for that. For one, the need for high performance script execution on the web

[10] https://www.ecma-international.org/ecma-262/8.0/#sec-ecmascript-standard-built-in-objects
[11] https://webkit.org/blog/7122/introducing-riptide-webkits-retreating-wavefront-concurrent-garbage-collector/
[12] https://v8.dev/blog/orinoco-parallel-scavenger

has resulted in complex engine implementations, an ideal environment for vulnerabilities. Second, the great amount of control over the engine's internal state e.g. in the form of memory allocations, together with the powerful exploitation primitives gained from typical vulnerabilities, often make it possible to bypass all modern exploit mitigations with a single vulnerability.

While "classic" vulnerabilities such as buffer overflows or use-after-frees are now rarely found in scripting engines, complex, domain specific vulnerabilities have mostly replaced them. In an attempt to categorize the commonly found issues in scripting engines, the following list has been created. This list makes no claims of completeness, but rather aims to provide a quick overview of common vulnerabilities discovered in recent years.

- Integer-overflow related issues, usually leading to out-of-bounds access into a memory buffer. Examples include CVE-2017-7092, CVE-2017-2536, and CVE-2018-6065.
- Issues due to unexpected callbacks in the implementation of builtin functions, such as CVE-2017-5030, CVE-2016-7189, and CVE-2016-4622, which will be discussed further in section 2.3.1.
- Use-after-free issues due to bugs that lead to the garbage collector not finding an object in the marking phase. Example of this type of bug include CVE-2017-2491 and CVE-2018-4192.
- Vulnerabilities that occur due to an engine-internal object or function being "leaked" to application code through some kind of logic issue. Examples include CVE-2016-6754, CVE-2017-2446, and CVE-2018-4299.
- Vulnerabilities stemming from incorrect optimizations in a JIT compiler. To this date, vulnerabilities have at least been found in the implementation of bounds-check elimination (e.g. CVE-2015-0817, CVE-2017-2547, and CVE-2018-0769), escape analysis (e.g. CVE-2017-5121, CVE-2018-0860), and type-check elimination (e.g. CVE-2017-11802, CVE-2018-17463, and CVE-2018-4233, discussed in more detail in section 2.3.2).

In addition to the described vulnerability classes, there exist a large number of miscellaneous vulnerabilities that currently cannot easily be assigned to any category. Future publications and research of JavaScript engine vulnerabilities might warrant the introduction of further categories.

The following two sections will describe two rather typical vulnerability classes in more detail.

2.3.1 CVE-2016-4622

The first vulnerability discussed here is CVE-2016-4622, a vulnerability in JavaScriptCore that is caused by an unexpected callback.

The vulnerability was present in the implementation of the `%Array.prototype.slice` builtin function in JavaScriptCore, the engine powering WebKit. The vulnerable source code is shown in listing 2.5. The function proceeds roughly as follows: first it retrieves

the length of the input array in line 9. Next it retrieves the arguments, converts them to integers, and clamps them to the range [0, length) in line 13 and 14. Finally, if the fast path can be used, it copies the bytes between `begin` and `end` into a newly created array without additional bounds checking in line 21 and 22. The issue here is that the conversion of the arguments to integers can execute JavaScript code of the attackers choosing due to JavaScript's conversion rules[13]. In particular, a function installed as the `valueOf` property on one of the arguments will be invoked at this point. In this callback, an attacker could now modify the length of the array, resulting in an out-of-bounds access when the function resumes. Listing 2.6 shows proof-of-concept code to trigger the bug. Execution of this code on a vulnerable instance yields an array with special double values such as 2.12199579146e-313. These correspond to pointer values that were read out-of-bounds behind the array. The resulting primitive can be used to construct a remote code execution exploit[14].

```
1  EncodedJSValue JSC_HOST_CALL arrayProtoFuncSlice(ExecState* exec)
2  {
3    JSObject* thisObj = exec->thisValue()
4                          .toThis(exec, StrictMode)
5                          .toObject(exec);
6    if (!thisObj)
7      return JSValue::encode(JSValue());
8
9    unsigned len = getLength(exec, thisObj);
10   if (exec->hadException())
11     return JSValue::encode(jsUndefined());
12
13   unsigned begin = argumentClampedIndex(exec, 0, len);
14   unsigned end = argumentClampedIndex(exec, 1, len, len);
15
16   ...;
17
18   if (/* can use fast path */ && isJSArray(thisObj)) {
19     if (JSArray* result =
20           asArray(thisObj)->fastSlice(*exec, begin, end - begin))
21       return JSValue::encode(result);
22   }
23 }
```

Listing 2.5: The vulnerable implementation of %Array.prototype.slice in JavaScriptCore, slightly modified for readability.

[13] https://www.ecma-international.org/ecma-262/8.0/index.html#sec-type-conversion
[14] https://github.com/saelo/jscpwn

```
1  var a = [];
2  for (var i = 0; i < 100; i++) {
3      a.push(i + 0.123);
4  }
5
6  var end = {
7      valueOf: function() {
8          a.length = 0;
9          return 10;
10     }
11 };
12
13 a.slice(0, end);
14 // [0.123,1.123,2.12199579146e-313,0,0,0,0,0,0,0]
```

Listing 2.6: Proof-of-concept code to trigger CVE-2016-4622.

2.3.2 CVE-2018-4233

The next vulnerability presented here is CVE-2018-4233, a JIT compiler optimization bug in the JavaScriptCore engine.

The optimization in question, often called *redundancy elimination*, is supposed to detect redundant type checks in the IR and eliminate those. Listing 2.7 provides an example of a redundant type check. Here, the second type check is redundant as the type of the argument cannot change between the first and the second check. Eliminating redundant type checks can be achieved by keeping track of previous checks on each path through the control flow graph and removing subsequent checks that are fully subsumed by the existing ones. Special care has to be taken as the types of objects can sometimes change in between two checks as the result of *side effects*. In these cases, the seemingly redundant check could not be safely removed. Such a situation is depicted in Listing 2.8. Here, the second `CheckHasProperty` operation in line 6 is still required as the type of the argument object can change due to side effects. In particular, the `ToNumber` operation in line 4 could trigger the execution of arbitrary JavaScript code in a similar way as described in 2.3.1, which in turn could change the type of the object by e.g. removing the property ”foo”. To detect these situations, the compiler requires knowledge about the side effects all operations in its IR. Failure to capture the side effects of an operation will lead to the removal of a type check and subsequently to a type confusion vulnerability.

```
1 v0 = LoadArgument 0
2 CheckIsInteger v0
3 v1 = IntegerAdd v0, 42
4 CheckIsInteger v0
5 v2 = IntegerMul v0, 42
```

Listing 2.7: An example for redundant type checks in an (imaginary) compiler intermediate representation.

```
1 v0 = LoadArgument 0
2 CheckHasProperty v0, "foo"
3 v1 = LoadProperty v0, "foo"
4 v2 = ToNumber v1
5 v3 = NumberAdd v2, 1
6 CheckHasProperty v0, "foo"
7 StoreProperty v0, "foo", v3
```

Listing 2.8: An example for a type check that seems redundant but is actually required as the type of the input can change due to side effects of the ToNumber conversion.

Such an issue is the cause for CVE-2018-4233. In particular, the compiler assumed that the `CreateThis` operation, responsible for creating the new object in a constructor call, would not result in any side effects. However, by wrapping the constructor in a Proxy[15], it is indeed possible to execute arbitrary JavaScript code during the execution of the `CreateThis` operation. By changing the type of an argument object, in this case from an array of floating-point numbers to an array of JavaScript values, it is possible to achieve a type confusion in the emitted machine code.

Proof-of-concept code for this vulnerability is shown in listing 2.9. Here, the JIT compiler assumes that the constructor function always receives an array with doubles as first argument. It guards this assumption with a type check at the beginning of the emitted machine code. However, the `CreateThis` operation happens after the type check of the argument object and invokes JavaScript through a Proxy callback when the `prototype` property of the constructor is retrieved. Changing the type of the argument array in the callback then causes a type confusion when the constructor function resumes and accesses the array.

As a result of executing the proof-of-concept code, the double value 3.54484805889626e-310, which is stored as 0x414141414141 in IEEE 754 format, is wrongly used as a pointer, resulting in an attacker-controlled crash due to an access violation when dereferencing the address 0x414141414141. A full exploit is available as well[16].

[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
[16] https://github.com/saelo/cve-2018-4233

```
1  function Constructor(a, v) {
2      a[0] = v;
3  }
4
5  var trigger = false;
6  var arg = null;
7  var handler = {
8      get(target, propname) {
9          // Potentially change the type of |arg| from
10         // array with doubles to array with anything.
11         if (trigger) {
12             arg[0] = {};
13         }
14         return target[propname];
15     },
16 };
17 var EvilProxy = new Proxy(Constructor, handler);
18
19 for (var i = 0; i < 100000; i++)
20     new EvilProxy([1.1, 2.2, 3.3], 13.37);
21 }
22
23 trigger = true;
24 arg = [1.1, 2.2, 3.3];
25 new EvilProxy(arg, 3.54484805889626e-310);
26 arg[0];         // Crashes here with an access violation
```

Listing 2.9: Proof-of-concept code to trigger CVE-2018-4233.

2.4 Fuzzing Basics

Fuzz-testing or fuzzing describe a group of related techniques for automatic vulnerability discovery. The central idea is that randomly generated input is repeatedly given to the application as input. The application is then monitored for faulty conditions such as memory access violations during processing of the generated input [SGA07].

Fuzz-testing approaches can broadly be divided into two main categories: generative approaches and mutation-based ones. These differ in the way they produce input data. Each of them will be discussed in more detail next.

2.4.1 Generative Fuzzing

In generative fuzzing, each input file is generated from scratch, often following a set of predefined rules. Commonly, grammar-based approaches are used, in which the set of all inputs is first defined through a context-free grammar [GKL08]. The generation of new inputs is then achieved by taking random production rules starting from an initial ”root” production.

An example grammar, specified in ABNF form [CO08], for a very limited subset of the JavaScript language is given in listing 2.10. A simple generative fuzzer would start with the

`Program` production rule and choose random productions until a certain recursion depth is reached, after which point it will always choose a terminating rule if available. As such, for the grammar given in listing 2.10, a sample such as `"var v42 = 7 + 3 + v13;"` could be produced.

```
1  program                = *statement
2  statement              = variable-declaration ";"
3  statement              =/ expression ";"
4  variable-declaration = "var" identifier "=" expression
5  expression             = integer-literal
6  expression             =/ add-expression
7  expression             =/ identifier
8  add-expression         = expression "+" expression
9  integer-literal        = 1*DIGIT
10 identifier             = "v" 1*DIGIT
```

Listing 2.10: Example grammar in ABNF form for a subset of the JavaScript language.

One of the main benefits of generative fuzzing is that the resulting samples will always be syntactically valid if the grammar is correct. Another benefit is the high degree of control over the produced samples by the grammar engineer. This might enable her to use domain-specific knowledge to tune the fuzzer by adding production rules for features that have historically proven dangerous. However, without some form of code emulation, it becomes hard to achieve semantic correctness of the generated samples. This is problematic as a runtime exception would stop further processing of the generated code. As such, a common solution is to wrap each generated statement or expression (or groups of such) into a try-catch block so that execution continues even in the event of a runtime exception.

2.4.2 Mutation-based Fuzzing

The idea behind mutation-based fuzzing is to start with a set of known-good input files and mutate those in a random manner. Possible mutations include bit and byte flipping, incrementing and decrementing integer values, and the insertion of predefined, "interesting" integer and string values. Mutation-based approaches perform well for programs that parse binary protocols, as mutations are relatively straightforward. However, input data which contains checksum fields or similar integrity checks can pose problems as processing of the generated input will often stop during integrity checking. A simple way to deal with these issues is to either modify the target program to ignore the checksum or to recompute the checksum after the mutations.

The success of mutation-based fuzzing is largely dependent on the quality of the initial corpus as well as the type of mutations that are performed.

When targeting scripting languages, a main challenge is to define sensible mutations, as bit-level mutations on the source code representation of the code are likely to result in syntactically invalid programs, which will be rejected early on during parsing. One approach is to mutate the AST which has successfully been implemented in the past, e.g. by Holler et al. [HHZ12]. With this, syntactic correctness is easy to achieve. Semantic correctness, on

the other hand, is not automatically achieved as a mutation of the AST could for example result in a variable being used before it is defined. This problem can be tackled either by using mutations that preserve semantic correctness in a majority of the cases, such as pure insertions of existing code from other samples, or by using try-catch constructs as previously described.

2.5 Guided Fuzzing

An extension to the basic fuzzing algorithm, in which the outcome of the generated samples is largely ignored, is guided fuzzing. Here, some form of feedback about the produced samples is used to improve future samples. This becomes easily possible when a mutation-based fuzzing approach is used as basis. In that case, samples that receive positive feedback will be kept for future mutations while ”uninteresting” samples will be discarded. In essence, guided fuzzing is then a genetic algorithm: mutations are performed on samples from the current generation and ”good” samples are kept for the next generation while others are discarded.

To perform well, guided fuzzing approaches require the use of sensible mutations which preserve the interesting features of existing samples while changing their semantic slightly. Furthermore, guided fuzzing requires a metric to determine whenever a sample has triggered ”interesting” behavior and should thus receive positive feedback. A generic and broadly used metric is *edge coverage*, in which case the algorithm is commonly referred to as coverage-guided fuzzing. With edge coverage, a sample is determined ”interesting” if it has discovered an edge in the control flow graph of the target program that has never, or less often, been discovered before. As an example, consider the imaginative excerpt of the control flow graph of a program given in listing 2.1. Assuming that previous samples caused the path `A->B->A->C->E` to be taken, then a sample that triggers the path `A->C->D->E` will be considered interesting and kept for future mutations as the edges `CD` and `DE` have never been observed before. However, a sample that caused the path `A->B->A->B->A->C->E` to be taken would not be considered interesting even though it resulted in a unique path through the program.

The intention behind edge coverage are twofold. For one, it naturally achieves code coverage of the target program as edge coverage subsumes basic block coverage. Additionally, it is able to distinguish between different state transitions in the tested program while largely avoiding issues like path explosion. As such, coverage-guided fuzzing is able to gradually generate complex but valid inputs for the program under test. Pseudocode for the general guided fuzzing algorithm is given in listing 2.11.

A popular tool that implements the coverage-guided fuzzing approach is afl [Zal15], which has been able to discover a multitude of previously unknown vulnerabilities and serves as a model for this work.

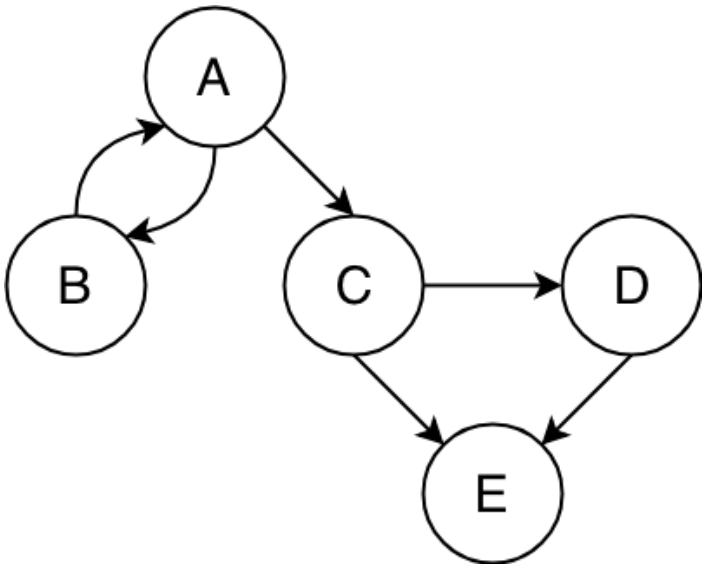

Figure 2.1 An exemplary control flow graph.

```
1  let C be a corpus of input samples
2  repeat indefinitely:
3      let S be a random sample in C
4      let S' be the result of applying a random mutation to S
5      let R be the result of executing S'
6      if R resulted in a crash:
7          save S' to disk
8      otherwise if R contains previously unseen edges:
9          mark new edges as seen
10         insert S' into C
```

Listing 2.11: The high-level guided fuzzing algorithm

3. Analysis

The chapter will define the goal of this work, then define a set of requirements that the developed system should meet to achieve said goal.

3.1 Goal of this Work

As motivated in chapter 1, this work is aimed towards the automatic discovery of vulnerabilities in JavaScript engines based on fuzz-testing. We note that, to be successful, this does not necessarily require the developed system to outperform competing approaches in some way. Rather, it has to be able to detect different vulnerabilities than existing fuzzers.

Given the recent success of coverage-guided approaches, we consider coverage to be a useful metric for interpreter fuzzing as well. As an example, JIT compiler engines generally require special casing in various situations. These would be discoverable with a coverage guided approach. Another benefit of coverage guidance is that specific components of the engine, or the browser as a whole, can be targeted by adjusting the coverage feedback, e.g. so it is only measured for the desired components. As such we believe that a coverage-guided approach could be able to discover new vulnerabilities.

The goal of this work is thus to provide an implementation of a coverage-guided fuzzing approach for JavaScript engines which is able to discover vulnerabilities inside the engine such as the ones previously presented in section 2.3.

3.2 Requirements

To achieve the defined goal, a number of requirements were defined which the developed approach had to satisfy.

First, we note that syntactically invalid programs will most of the time fail to reach past the parsing stage of the engine. While there might well be vulnerabilities in the parser, it is a relatively small and uncomplex component of the engine. As an example, in JavaScriptCore as of April 2018, the parser made up only approximately 3% of the engine's source code. As such, a requirement is that all generated programs have to be syntactically correct under all circumstances.

Second, given the goal of guided fuzzing, it is necessary to define sensible mutations to preserve most of the interesting features of existing samples during mutations. As the goal is to detect defects in core components of the engine, such as JIT compilers, it is necessary to define mutations that are able to change the control and data flow of a program, as that is what is ultimately processed by these components.

Finally, we note that ad-hoc solutions for dealing with semantically invalid programs, namely the use of try-catch constructs, influence the behavior of the engine, especially the JIT compiler, which might lead to missed vulnerabilities. We were able to confirm this suspicion with the testcase for CVE-2018-12386, discussed in section 6.3. The discovered testcase did indeed not result in a crash if the function body, or part thereof, was surrounded by try-catch statements. While this does not mean that it is impossible to trigger such a vulnerability with a fuzzer that wraps every statement inside a try-catch block, it shows that doing so in fact influences the processing of the engine in a way that can lead to missed vulnerabilities. As such, the third requirement is that our system is able to generate semantically valid programs in high number as to avoid the need for said try-catch constructs. It should, however, be noted that it is generally desirable to generate semantically invalid programs as well as vulnerabilities often occur due to unexpected internal exceptions being raised. Complete semantic correctness is thus not desirable.

To summarize, our requirements are as follows:

1. Emit syntactically valid samples
2. Define mutations to control and data flow of a program
3. Emit a high percentage of semantically valid samples

4. Coverage Guided JavaScript Fuzzing

This chapter explores a new approach towards coverage guided fuzzing for scripting engines. It starts out with an overview of the proposed system and a discussion about how it aims to satisfy the previously defined requirements, namely syntactic correctness, the use of a mutation-based approach, and a high degree of semantic correctness. Afterwards each of the relevant components of the presented approach is described in more detail.

4.1 Concept

The central idea to solving the described requirements is to use mutation-based fuzzing on JavaScript code. However, instead of mutating syntactic constructs like the AST or even the textual source code representation, we instead propose to mutate on a ”bytecode” level which is closer to the engines internal representation of the code. This is depicted in figure 4.1. In essence, the bytecode level resembles a control and data flow graph, which is what is ultimately required to reach the majority of the attack surface of a scripting engine while syntactic information, such as the AST, is largely discarded during the parsing step.

To achieve this, we define a custom intermediate language which we call FuzzIL. This enables us to define new mutation strategies that could not easily be implemented as AST mutations, such as the input mutator discussed in section 4.3.1. This satisfies the second requirement.

To satisfy the first requirement, the syntactical correctness of the produced samples, it is enough to guarantee that the conversion from our IL to JavaScript is always possible. This is trivial to guarantee as will further be discussed below.

To satisfy the third requirement, we first note that due to the relatively minor changes performed by each mutation, the chances of it turning a semantically valid program into an invalid one are relatively small. Further, all mutations are required to obey to a set of basic semantic correctness rules of the IL, enforcing such things as the definition of a variable before its usage. Finally, by avoiding the inclusion of semantically invalid programs, it is possible to keep the overall percentage of emitted invalid programs at an acceptable level. We slightly extend the definition of semantic correctness for our programs to reject samples

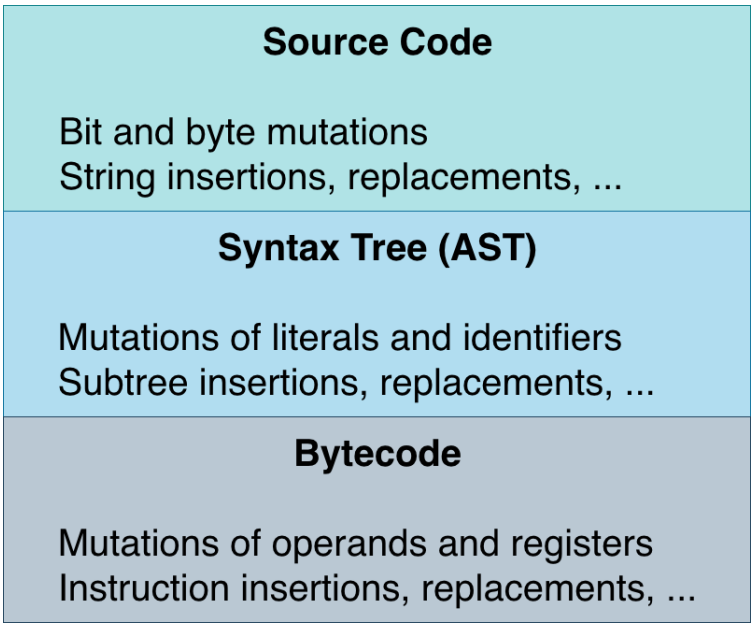

Figure 4.1 Different levels at which mutations can be applied for a scripting language.

that run for too long as well. As such we define a program to be *valid* if it neither results in a runtime exception nor a timeout but instead terminates successfully within the given timeframe.

4.2 FuzzIL

With FuzzIL we introduce a custom intermediate language (IL) that lends itself to easy mutations while allowing straight-forward conversion, from here on called *lowering*, to JavaScript code. A FuzzIL program consists of a list of *instructions*, each in turn consisting of an *operation* together with a list of input and output variables. Listing 4.1 shows an example FuzzIL program consisting of twelve instructions which compute the sum of the numbers from zero to nine and output it. Variables are identified through integers and are required to be numbered consecutively starting from zero in every program. Amongst others, this makes it possible to implement supporting data structures in a simple way, e.g. a set of variables as bitmap and a map of variables to some value type with a linear array.

Lowering of a FuzzIL program to JavaScript is possible in various ways. Listing 4.2 shows the result of a straight-forward lowering algorithm, which converts each instruction by itself. Listing 4.3 shows the result of a more elaborate lowering algorithm which replaces variable uses with the expression defining them in certain circumstances and omits variable definitions if these are not needed. While more difficult to implement and mostly irrelevant for the execution of the code, the second algorithm produces more human readable output, which in turn facilitates the analysis of both crashes and interesting programs to gain insights into the state of the fuzzer. Both algorithms have been implemented.

Many FuzzIL Operations are *parametric*. In the representation seen in listing 4.1, the parameters of an operation are enclosed in single quotes. Parameters include constants in operations such as `LoadInt` and `LoadString`, property and method names, and operators in binary and unary operations as well as in comparisons.

In FuzzIL, control flow is implemented by special block instructions. For each block type there exist at least two operations: one that starts the block and one that ends it. This can be seen in listing 4.1 with the For loop in lines 5 through 8. Each block has its own scope, meaning that variables defined inside of it are not visible to the outside. Furthermore, the input variables to the block instructions themselves, such as the condition variable in a `while` or `do-while` loop, have to be defined in the outer scope. This reflects the behavior of common programming languages.

A FuzzIL program is always in SSA form, meaning that a variable is assigned exactly once. SSA form facilitates the implementation of various code analysis algorithms, such as the define-use analysis. To produce behavior that requires the reassignment of a variable we introduce *Phi* operations, which in some ways work inverse to Phis in compiler construction theory: a variable defined through a *Phi* operation has an initial value (the second input to the Phi operation) but can be used as first input of a *Copy* operation, which will result in a reassignment of the variable. This can be seen in listing 4.1 in line 4 and 7.

The following invariants must hold for every FuzzIL program, and thus for every JavaScript program generated from it:

- Variables are numbered consecutively
- All input values to an instruction must be variables, there are no immediate values or nested expressions
- All variables must be defined before they are used, either in the current block or an enclosing one
- A block begin must eventually either be followed by the corresponding closing instruction or by an intermediate block instruction, such as a `BeginElse` for which the same holds true
- All inputs to block instruction must be defined in an outer block
- The first input to a `Copy` instruction must be the output of a `Phi` instruction

The deliberately strict form of a FuzzIL program makes it possible to define various mutations on a FuzzIL program. These will be described in the next section.

```
1  v0 <- LoadInt '0'
2  v1 <- LoadInt '10'
3  v2 <- LoadInt '1'
4  v3 <- Phi v0
5  BeginFor v0, '<', v1, '+', v2 -> v4
6      v6 <- BinaryOperation v3, '+', v4
7      Copy v3, v6
8  EndFor
9  v7 <- LoadString 'Result: '
10 v8 <- BinaryOperation v7, '+', v3
11 v9 <- LoadGlobal 'console'
12 v10 <- CallMethod v9, 'log', [v8]
```

Listing 4.1: An example FuzzIL program.

```
const v0 = 0;
const v1 = 10;
const v2 = 1;
let v3 = v0;
for (let v4 = v0; v4 < v1; v4 = v4 + v2) {
    const v6 = v3 + v4;
    v3 = v6;
}
const v7 = "Result: ";
const v8 = v7 + v3;
const v9 = console;
const v10 = v9.log(v8);
```

Listing 4.2: The example FuzzIL program 4.1 converted to JavaScript through a naive lowering algorithm.

```
let v3 = 0;
for (let v4 = 0; v4 < 10; v4++) {
    v3 = v3 + v4;
}
console.log("Result: " + v3);
```

Listing 4.3: The example FuzzIL program 4.1 converted to JavaScript through a more elaborate lowering algorithm which performs expression inlining if possible.

The following JavaScript language features are currently supported by FuzzIL:

- Number, string, and boolean literals with the `LoadInt`, `LoadFloat`, `LoadString`, and `LoadBoolean` operations
- Null and undefined values with the `LoadNull` and `LoadUndefined` operation
- Object and array literals with the `CreateObject` and `CreateArray` operations
- Function definitions with the `BeginFunctionDefinition` and `EndFunctionDefinition` operation
- Property access with the `LoadProperty`, `StoreProperty`, and `DeleteProperty` operation
- Computed property access, in which another variable is used as property name, with the `LoadComputedProperty`, `StoreComputedProperty`, and `DeleteComputedProperty` operations
- Function-, method- and constructor calls with the `CallFunction`, `CallMethod`, and `Construct` operations
- Binary operations, unary operations, and comparisons with the `BinaryOperation`, `UnaryOperation`, and `Compare` operation

- Various loops with the `BeginWhile`, `BeginDoWhile`, `BeginFor`, `BeginForIn`, `BeginForOf`, and the corresponding closing operations
- If-else constructs with the `BeginIf`, `BeginElse`, and `EndIf` operations
- Try-catch constructs with the `BeginTry`, `BeginCatch`, and `EndTryCatch` operations
- Other control flow features such as `Continue`, `Break`, and `Return`

4.3 Mutations

A mutation transforms an existing program into a new, slightly different program, while ideally preserving its interesting behavior. We propose the following mutations to perform modifications of control and data flow as well as to combine multiple programs into one:

- The input mutator, which changes input variables of instructions
- The operation mutator, which mutates parameters of operations
- The insertion mutator, which inserts newly generated code into a program
- The combine mutator, which combines two existing programs together
- The splice mutator, which inserts a part of an existing program into another one

Each one will be discussed in more detail in the following sections.

Additional mutation algorithms, like duplicating instructions and replacing output variables, have been evaluated but did not result in improved results. However, it is likely that further mutation algorithms, especially ones that target the control flow of a program, will lead to improved results. This remains subject of future work.

4.3.1 Input Mutator

This mutator is the prime way of mutating data flow in a program. It makes use of the fact that in FuzzIL, all inputs to an instruction are variables. As such the input mutator operates by selecting one or multiple instructions in the original program and replacing one of their inputs with a different, random variable. Listing 4.4 shows an example of this mutation. Here, the first argument of a function call has been replaced with a different value.

```
1 # Before Mutation
2 ...
3 v16 <- CallFunction v1, v6, v9, v3
4
5 # After Mutation
6 ...
7 v16 <- CallFunction v1, v12, v9, v3
```

Listing 4.4: Example application of the Input Mutator.

Special handling is required to ensure that the first input to a `Copy` instruction is always a variable produced by a `Phi` instruction and to ensure that the inputs to a closing block instruction are selected from the outer scope.

4.3.2 Operation Mutator

This mutator is responsible for mutating parameters of operations. This includes the constants used by the `LoadInteger`, `LoadString`, `LoadBoolean`, and `LoadFloat` operations, but also the comparator used in comparisons, the property or method name used in property operations and method calls, as well as the actual operation in a binary or unary operation.

4.3.3 Insertion Mutator

This mutator is the only way that new code can be generated and is in essence a small generative component. It makes use of a list of predefined *code generators*: functions that emit FuzzIL code into the mutated program.

A multitude of code generators have been implemented, ranging from fundamental code generators to produce an integer literal or call a function to specific ones that produce an instance of a TypedArray[1] or cause a previously defined function to be JIT compiled by repeatedly invoking it.

The insertion mutator allows us to start with an empty corpus instead of having to import existing code, as new code will be generated by this mutator.

4.3.4 Combine Mutator

The combine mutator, as well as the splice mutator discussed next, allow the combination of two existing programs. The combine mutator is the simpler one of the two, inserting another program in full at a random position in the current program. This only requires the renaming of all variables in the inserted program and the following code of the outer program to avoid collisions of variable numbers between the two. Listing 4.5 shows an example of this mutator. Here the first program has been inserted into the second one after the first instruction.

```
1  # Program 1
2  v0 <- LoadString 'bar'
3  v1 <- CreateObject 'foo', v0
4
5  # Program 2
6  v0 <- LoadInt '1337'
7  v1 <- LoadGlobal 'console'
8  v2 <- CallMethod v1, 'log', [v0]
9
10 # Combined program
11 v0 <- LoadInt '1337'
12 v1 <- LoadString 'bar'
13 v2 <- CreateObject 'foo', v1
14 v3 <- LoadGlobal 'console'
15 v4 <- CallMethod v3, 'log', [v0]
```

Listing 4.5: Example application of the Combine Mutator.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays

4.3.5 Splice Mutator

The splice mutator works similar to the combine mutator in that it inserts existing code at a random position in the mutated program. However, instead of inserting another program in full, it will insert a group of related instructions which is called a *slice*. The key property of a slice is that each variable that is used by any instruction in the slice is also defined by an instruction in the slice. During insertion, variable numbers will be changed in the same way as for the combine mutator.

A slice of a program is found through the following algorithm: first select a random instruction of the program, then recursively select all instructions whose output variables are used by the current selection. When all input variables are defined, copy all selected instruction in their previous order into the mutated program.

4.4 Scoring

As described in section 2.5, guided fuzzing inevitably requires a metric to score every sample. We chose edge coverage, a broadly used and fairly universal metric, for our implementation. With edge coverage, a sample is considered interesting if it reaches a previously unseen edge in the control flow graph. In contrast to many existing fuzzers such as afl [Zal15], we do not take edge counts into account. This is due to the observation that every edge can likely be triggered multiple times by wrapping the generated code into a loop. Edges can thus be represented as a bitmap with one bit for each edge. If at least one previously unseen edge is triggered by a sample, that sample is regarded ”interesting” and added to the corpus after refinement as described in the next section.

4.5 Refinement

Before adding newly found samples to the corpus or saving crashing samples to disk, a set of *refinement* steps is performed. Refinement currently consists of three steps: a check for deterministic behavior of the sample, followed by *minimization*, and finally *normalization* of the sample.

4.5.1 Determinism

The first step of the refinement phase is to check whether the new sample behaves in a deterministic way. A sample that resulted in a crash is deemed deterministic if it crashes during a second execution, while a sample that triggered new coverage edges is deemed deterministic if all newly discovered edges are triggered again during a second execution.

Determinism of samples is relevant for both crashes and interesting samples. For crashes, a non-deterministic crash might be a false positive, for example related to a low-memory situation of the system as a whole. In the current implementation, non-deterministic crashes are still saved to disk but marked as non-deterministic in the filename. For interesting samples, non-determinism essentially implies that the following refinement steps, namely minimization, will not, or only to a limited degree, be successful. As such non-deterministic samples that triggered a new edge are currently discarded.

4.5.2 Minimization

The main part of the refinement process is the *minimization* phase in which the sample's size is reduced as much as possible while still preserving its original, interesting behavior. This is necessary as mutations can only grow the sample in size, never shrink it. As such, the average program size would drastically increase over time without minimization, resulting in lower performance. Furthermore, the amount of instructions that are not relevant for a programs behavior would grow as well, decreasing the effectiveness of various mutations.

The minimization phase is implemented as a fixed-point iteration of different minimization algorithms. Each of them in turn performs a set of *reductions*: atomic modifications to the current program resulting in a new program that is, in some way, smaller than the original one. After one such reduction is performed, the resulting program is executed to verify whether it still behaves in the original way, meaning either crashing or triggering all of the previously unseen edges. If its behavior changed, the program is reverted to its previous state, otherwise the changes are kept.

It should be noted that minimization in general is an expensive process. In fact, during the first several million iterations, it is common to see the number of executions spent on minimization to outnumber the number of executions of newly generated programs. As the rate of newly discovered edges decreases over time, this effect disappears and the number of executions spent on minimization usually accounts for only around one percent of the total executions after ten to hundred million iterations.

The different minimization algorithms currently used are discussed next.

4.5.2.1 Generic Instruction Remover

The first minimization algorithm is also the most generic one. It tries to remove non-block instructions that do not contribute to the desired behavior.

The general idea is to iterate backwards through the program and remove every instruction whose output is not used in the following code. Each deletion of an instruction must be performed as a single reduction. As such, this minimizer will require $O(n)$ program executions, where n is the number of instructions in the program.

4.5.2.2 Input Reducer

One issue that remains is that instructions with a variable number of inputs, such as function calls or array literals, can potentially prevent a large number of unnecessary instructions from being removed during the generic minimization. Listing 4.6 shows an example of such a situation. Here, the first four instruction are all required by the function call in line 6, preventing them from being removed if the call itself is necessary, even though the function in question does not make use of the argument.

To solve this, a second minimization algorithm is used which removes inputs of instructions whose operation can take a variable number of inputs, namely the `CreateObject`, `CreateArray`, `CallFunction`, `CallMethod`, and `Construct` operations. As some of the inputs might be required, each input must be removed as a separate reduction.

```
1 v0 <- LoadInt '42'
2 v1 <- LoadString 'foo'
3 v2 <- LoadBool 'true'
4 v3 <- CreateArray v0, v1, v2
5 v4 <- LoadGlobal 'Math'
6 v5 <- CallMethod v4, 'random', v3
```

Listing 4.6: An unused argument to a function call preventing a number of other instructions from being removed during minimization.

The core issue is, however, not limited to instructions with a variable number of arguments. It is, for example, often possible to replace an addition operation with an integer literal as only the type of the output is relevant. However, fully assessing the impact of this more generic issue and designing an algorithm to resolve it remain subject of future work.

4.5.2.3 Block Reducer

So far only non-block instructions have been considered. Removal of block instructions is, however, equally important. There appears to be no general algorithm that is capable of removing block instructions. Instead, special handling is required for the different types of blocks, namely loops, if-else constructs, and try-catch blocks.

When dealing with loops, the goal is to determine whether the loop itself is relevant or only the instructions in the loop body. For that it suffices to remove only the loop itself (more specifically, the two instructions that start and end the loop block) and leave the loop body in place, which would now execute only once. However, special treatment has to be given to `break` and `continue` instructions inside the loop body: without a surrounding loop, these would be syntactically invalid, preventing the removal of the loop. As such any `break` or `continue` instruction inside the loop body, which are not part of a nested loop, must be removed as well.

If-else constructs are currently implemented by three instructions: `BeginIf`, `BeginElse`, and `EndIf`. Similar to loops, the if-else construct itself might not be required, only the body of either of the blocks. As such it would be possible to first remove everything but the body of the If block, then, if that did not succeed, remove everything but the body of the Else block. However, it is likely that the body of the block that is not relevant has already been fully removed by previous reductions. As such it suffices to remove all the three block instructions at once.

Similar to if-else constructs, try-catch constructs are supported through the `BeginTry`, `BeginCatch`, and `EndTryCatch` operations. In case the try-catch construct itself is not required to reach the interesting behavior, the following possibilities exist:

1. Only the body of the Try block is required
2. Only the body of the Catch block is required, which is reached through some operation in the Try block that throws an exception
3. The first part of the Try block, until the instruction that raises an exception, is required as well as the entire body of the Catch block

The first case is resolved by removing only the three block instructions, as in that case, the body of the Catch block will likely have been removed already. The second case is resolved by removing the three block instructions as well as the entire body of the Try block. This is necessary as the group of instructions in the Try block that cause an exception to be raised might be arbitrarily large. Consider listing 4.7 for an example of such a situation that was found during testing. The third case is currently only resolved heuristically: the three block instructions as well as the last instruction in the Try block, which is assumed to be the one raising an exception, are removed. This would, however, not work for the example in listing 4.7 if there were some relevant code before the loop in line 2.

```
1 try {
2     for (let v16 = 0; v16 < 27; v16 = v16 + -2473693327) {
3         const v17 = Math(v16,v16);
4      }
5  } catch {
6      some_interesting_code();
7  }
```

Listing 4.7: A group of instructions that trigger an exception and thus transfer execution to the Catch block.

4.5.2.4 Special-Purpose Minimizers

While the presented minimizers are generally able to minimize a given program significantly, there are some code patterns that cannot be removed with the generic approaches. These code patterns, from here on called *artifacts*, can cause the generated programs to grow unnecessarily large as they, once introduced in a program, will remain in all programs derived from it later on. Listing 4.8 shows a simple example of such an artifact that occurs when using a lowering algorithm that is capable of inlining expressions. In this case, the logical *or* operation might not be required. However, removing just the logical *or* will result in the function call suddenly becoming live code. This will raise an exception, causing the reduction to fail.

This particular issue can be resolved with a special-purpose minimizer that removes the logical *or* as well as its right-hand side. However, a more generic solution is to not perform expression inlining when lowering samples intended for execution to avoid the occurrence of dead code in the first place.

```
1 # FuzzIL
2 v0 = LoadInt '42'
3 v1 = LoadInt '43'
4 v2 = CallFunction v1
5 v3 = BinaryOperation v0 '||' v2
6
7 # Lowered to JavaScript
8 const v0 = 42 || (43)();
```

Listing 4.8: A code artifact that cannot be removed by the existing minimizers.

Another kind of artifact commonly observed are nested function calls such as shown in listing 4.9. This type of code can often be reduced simply to `perform_interesting_action(42)`. The way to achieve this is by implementing *inlining*: functions that are called exactly once are discarded and the calling instruction is replaced with the function body. Function parameters are replaced by the arguments while the return value is replaced with a Phi variable which is initially set to undefined, then reassigned at each return instruction in the function body. Return instructions are then discarded as well.

There are likely further artifacts, which, if they occur frequently, will require special handling during minimization.

```
1  function v0() {
2      function v1() {
3          function v2() {
4              function v3(v4) {
5                  perform_interesting_action(v4);
6              }
7              v3(42);
8          }
9          v2();
10     }
11     v1();
12 }
13 v0()
```

Listing 4.9: Nesting of functions can build up over time and requires inlining of functions to resolve.

4.5.3 Normalization

Normalization is the final step of the refinement process. During normalization, variables are renamed so they are numbered consecutively again, an invariant of every FuzzIL program. This is necessary as instructions, and thus output variables, might have been removed during minimization, leaving holes in the space of used variable numbers in the program.

4.6 Fuzz-testing with FuzzIL

After the central mechanisms of our proposed system have been presented in the preceding sections, this section now discusses the overall architecture. The general scheme follows the one described in section 2.5.

Mutation-based fuzzing requires an initial corpus of test cases. In our current implementation, the initial corpus, however, only contains the program shown in listing 4.10. This is possible due to the insertion mutator, discussed in section 4.3.3, which is able to generate new code from scratch. It is likely that a larger initial corpus, possibly containing programs from various regression tests, could lead to improved results. This would require a lifter to convert JavaScript to FuzzIL and remains the subject of future work.

```
1 v0 <- LoadGlobal "Object"
2 v1 <- CallFunction v0
```

Listing 4.10: The (arbitrarily chosen) seed program in the initial corpus.

The central fuzzing loop repeatedly chooses a random program from the current corpus and mutates it. However, it is often unlikely that a single mutation will suffice to discover new behavior. Moreover, some mutations tend to work well together: an input mutation might be most effective if it can combine the data flow of two separate programs that were previously combined into one program by a splice or combine mutation. As such, a fixed number of mutations are applied consecutively before a new program is selected from the corpus. Through measurements of the achieved coverage after one hundred million of iterations, it was determined that the number of consecutive mutations on a single sample, referred to as `N` in listing 4.11, should be somewhere between five and fifteen. If that number is too low, it becomes less likely to discover new behavior. On the other hand, if that number is too high, many iterations will be "wasted" on programs that have already lost their interesting behavior due to a previous mutation.

The next step of the fuzzing loop is then to execute the mutated program and evaluate the result. Execution can generally result in one of the following outcomes, which are processed differently:

- Execution resulted in a crash. In that case the sample is minimized and stored to disk.
- Execution terminated unsuccessfully, either through a runtime exception or a timeout. In that case the generated program is invalid and is discarded.
- Execution triggered previously unseen control flow edges. In that case the program is deemed interesting and is, after refinement, inserted into the corpus.
- Otherwise the program is valid but causes no new behavior. In that case the currently mutated program is replaced with the newly generated one.

Listing 4.11 summarizes the core fuzzing algorithm.

Programs that have been selected for mutation a certain number of times (currently 128) are subject to eviction from the corpus. However, a minimal size of the corpus (currently 1024) is enforced as well to prevent the corpus from depleting.

```
1  let C be an empty set of FuzzIL programs
2  insert an initial program into C
3  repeat indefinitely:
4      let P be a random program in C
5      repeat N times:
6          let P' be the result of applying a random mutation to P
7          let R be the result of executing P'
8          if R resulted in a crash:
9              minimize and normalize P'
10             save P' to disk
11         otherwise if R terminated unsuccessfully:
12             continue
13         otherwise if R contains previously unseen edges:
14             if P' behaves deterministically:
15                 mark the new edges as seen
16                 minimize and normalize P'
17                 insert P' into C
18         otherwise:
19             set P to P'
```

Listing 4.11: High-level fuzzing algorithm.

5. Implementation

The proposed system was implemented in the Swift[1] programming language, with some parts implemented in C. The implementation is around 7000 lines of code (LOC) in size and currently supports the Linux and macOS operating systems. This section will highlight some of the noteworthy aspects of the implementation.

5.1 Operations and Instructions

Internally, the distinction between Instructions and Operations allows Operations to be shared between multiple programs as they do not contain any program-specific data such as variable names. Operations are thus implemented as Swift classes, meaning that they have reference semantic and are automatically reference counted to avoid memory leaks. They store a set of flags, such as whether they represent the start or end of a block, as well as the number of input and output variables. Further, as some operations are parametric, the parameters are also stored in child classes. Listing 5.1 shows the implementation of the Operation class while listing 5.2 shows the `LoadInteger` subclass. Variables are implemented as simple 16-bit integers for low memory usage while Instructions are implemented as a pointer to an Operation together with an array storing both in- and outputs. The number of in- and outputs of each instruction is defined by its Operation.

[1] https://swift.org

```
class Operation {
    let flags: OperationFlags

    // The number of input variables to this operation
    let numInputs: Int

    // The number of newly created variables in the current scope
    let numOutputs: Int

    // The number of newly created variables in the inner scope
    // if one is created by this instruction
    let numInnerOutputs: Int
}
```

Listing 5.1: The fields of the Operation class.

```
class LoadInteger: Operation {
    let value: Int

    init(value: Int) {
        self.value = value
        super.init(numInputs: 0, numOutputs: 1,
            flags: [.isPrimitive, .isParametric])
    }
}
```

Listing 5.2: The fields and constructor of the LoadInteger class.

5.2 Program Construction and Mutation

A mutation algorithm is expected to receive a program as input and return a mutated copy of the program as output. Mutations are then performed while copying the input program.

To facilitate the construction of programs, and thus the implementation of mutation algorithms, a `ProgramBuilder` class has been implemented. At its core, it provides an API to construct a program by adding instructions to it. Further, it manages the variable space of the program being built and provides access to them. Finally, it takes care of renaming variables when appending instructions from a different program. As such, the code shown in listing 5.3 suffices to concatenate two programs together.

```
let program1 = ...
let program2 = ...

let b = ProgramBuilder()
b.append(program1)
b.append(program2)

print(b.program)
```

Listing 5.3: Example usage of the ProgramBuilder class to concatenate two programs. This includes renumbering variables of the second program to avoid collisions.

5.3 Type System

A lightweight type system and type tracking mechanism has been implemented which assigns a type to every variable in a program during construction. This is purely an optimization to avoid producing obviously invalid constructs such as performing a function call on a primitive type such as a number. Currently, the following types are implemented: Unknown, Integer, Float, String, Boolean, Object, and Function. These correspond to the output of the operations `LoadInt`, `LoadFloat`, `LoadString`, `LoadBoolean` and `Compare`, `CreateObject` and `CreateArray`. The Unknown type is given to all other output variables, in particular the result of any function or method call and property loads. It would be possible to compute the types of certain binary and unary operations if the input types are known. Furthermore it would be possible to determine types of the result of some function calls as well. Further improvements to the type tracking remain subject of future work.

5.4 Code Generators

As mentioned in section 4.3.3, a number of code generators are used by the insertion mutator to generate new code. Listing 5.4 gives an example of such a code generator. The code generator receives a program builder instance containing the currently constructed program and adds a number of instructions at the current position in the program. In this particular case it generates code to set the prototype of a random object to another object. As such, it would generate JavaScript code similar to the one shown in listing 5.5. This code generator also shows how the type tracking, performed during program construction, can be used by clients of the `ProgramBuilder` class. The type `MaybeObject` is a combination of the Unknown and the Object type.

```
struct PrototypeOverwriteGenerator: CodeGenerator {
    func generate(_ b: ProgramBuilder) -> Bool {
        let obj = b.randVar(ofType: .MaybeObject)
        let proto = b.randVar(ofType: .MaybeObject)
        b.storeProperty(proto, as: "__proto__", on: obj)
        return true
    }
}
```

Listing 5.4: An example for a code generator as used by the insertion mutator.

```
1 v42.__proto__ = v17;
```

Listing 5.5: Exemplary JavaScript code generated by the code generator in listing 5.4.

5.5 Coverage

Coverage measurements were implemented using a shared memory region between the fuzzer process and the JavaScript engine process as well as compiler instrumentation to collect edge coverage information. The target program was instrumented using clangs sanitizer-coverage feature, available through the `-fsanitize-coverage=trace-pc-guard` command line option. This causes all branches in the control flow graph to be instrumented with a call to a special function `__sanitizer_cov_trace_pc_guard` which receives a pointer to an integer which uniquely identifies the edge. In our case edges were numbered from 1 through N. The implementation of `__sanitizer_cov_trace_pc_guard` is given in listing 5.6. It computes the index of the edge in the shared memory bitmap and sets the corresponding bit. Afterwards, it deactivates the edge by setting its number to zero. This is merely a performance improvement to avoid setting the same bit multiple times. Compiler instrumentation naturally induces a certain overhead in execution speed. In our measurements, this overhead was less than a factor of two of the original execution speed.

```
1 extern "C" void __sanitizer_cov_trace_pc_guard(uint32_t *guard) {
2   if (!*guard) return;
3   uint32_t index = *guard - 1;
4   shmem->edges[index / 8] |= 1 << (index % 8);
5   *guard = 0;
```

Listing 5.6: Implementation of the `__sanitizer_cov_trace_pc_guard` which is responsible for collecting coverage information and writing it to a shared memory region.

5.6 Program Analysis

To support various mutation and minimization algorithms, a number of program analyses have been implemented. These include:

- A define-use analysis which is able to provide the definition of every variable in a program in $O(1)$ time. This is for example used to implement a lowering algorithm that is capable of inlining expression.
- A type analysis implementing the type tracking mechanism described in section 5.3.
- A scope analysis which keeps track of currently visible variables through a stack of active scopes, each containing a list of variables which were defined in them. This is used for efficient access to currently visible variables during program construction.
- A dead-code analysis which determines whether the current position in the program is in unreachable code, e.g. because it is preceded by a Return operation. This is used by Mutators that insert new or existing code into a program so they can avoid producing dead code.

Execution Strategy	Execution Time	Execution Overhead
Unoptimized	1009ms	9ms
Forkserver	1007ms	7ms
REPRL	1001ms	1ms

Table 5.1 Comparison of the execution overhead of three different execution mechanisms.

5.7 Program Execution

When running the generated scripts in the target engine, execution speed is crucial for the performance of the fuzzer as a whole. As such, two mechanisms to reduce the overhead of launching a new process have been implemented: a forkserver, as used by e.g. afl [Zal15], and another mode called REPRL, which is similar in nature to the in-process fuzzing performed by e.g. libFuzzer[2].

The idea behind a forkserver is to save the large overhead of the `execve` or similar system call. This is done by adding a piece of code into the target program to run after initialization is completed but before any input is processed. This code will wait for commands from the fuzzer process and then perform a `fork` system call for every generated input. The child process can then immediately process the input without performing its process initialization anew.

The other mode, called read-eval-print-repeat-loop, or REPRL for short, tries to reuse an existing process for multiple inputs. In essence it modifies the engine such that it will read input code from a predefined file descriptor and execute it, then reset its internal state and await the next program. This, too, avoids a large part of the initialization overhead.

The performance of the different approaches was evaluated by executing a script that would wait one second, then measuring the total execution time. The measurements were performed on JavaScriptCore at commit b4992ab, compiled with the `-O3` optimization level, and averaged over one hundred executions. The results are shown in table 5.1. As can be seen, the REPRL mechanism reduces the overhead the most, to about one millisecond.

5.8 Infinite Loops

Another performance problem are infinite loops, or in general loops that run for too long. Even though the code generators used by the insertion mutator rarely produce infinite loops, the input and operation mutator commonly cause such constructs to appear.

Infinite loops are mostly dealt with through a low timeout. An experimental feature to avoid infinite loops has been implemented in the form of *loop guards*: instructions that are placed at the beginning of function and loop bodies and which interrupt execution after a certain number of iterations. They are lowered to code such as shown in listing 5.7. As long as the guards do not trigger, they have minimal impact on the control and data flow of the program. However, they can clearly still result in different behavior of e.g. a JIT compiler. As such they are disabled by default.

[2]https://llvm.org/docs/LibFuzzer.html

```
/* Original FuzzIL program
...
BeginWhile v7
  LoopGuard
  ...
EndWhile
...
*/

...;
var guard3 = 0;
while (v7) {
  if (++guard3 > 10000)
    break;        // or alternatively throw an exception
  ...;
}
...;
```

Listing 5.7: A LoopGuard instruction at the beginning of a loop and the resulting JavaScript code.

5.9 Crash Deduplication

Basic support for crash deduplication, the problem of detecting when multiple crashes have the same root cause, was implemented as well. This was achieved in the same way as is done in afl [Zal15], namely through a dedicated edge bitmap only used for crashing samples. Each crashing testcase is then automatically minimized, as described in section 4.5.2, before edge coverage is measured and compared to this dedicated bitmap. If no new edges were reached the testcase is regarded a duplicate.

6. Evaluation

Due to the difficulty of comparing the quality of fuzz-testing approaches, as noted by Klees et al. [KRC+18], as well as the small number of publicly available alternative approaches and lack of benchmark suites to compare fuzzer performance we focus our evaluation mainly on newly discovered vulnerabilities, indicating that the presented approach is capable of finding defects in otherwise fuzz-tested and audited software. This follows the advice given by Klees et al.

As such, this chapter first discusses our evaluation methodology, then presents a number of newly discovered vulnerabilities. Afterwards, one of the discovered vulnerabilities, CVE-2018-12386, is discussed in more detail to highlight the path from a crashing sample to a proof-of-concept exploit. Finally, different types of vulnerability that are hard to detect with the presented approach are discussed and possible improvements suggested.

6.1 Methodology and Execution

Evaluation was performed during development on a Ubuntu 16.04 machine equipped with 64GB of RAM and a quad-core CPU with hyperthreading (Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz). Tests usually ran between 10 and 100 million iterations, with execution speed usually between 50 and 200 iterations per second. We acknowledge that this is generally too little: a thorough evaluation would require at least 30 tests per target, as suggested by Klees et al. [KRC+18]. Further, each test should run until there is no significant increase in coverage anymore, e.g. until the rate of newly found edges within a predefined number of iterations reaches zero. This will require multiple weeks of fuzzing on a machine comparable to ours and was thus not possible to do in the scope of this work.

To improve the execution speed and increase the chances of detecting faulty conditions we used non-standard configuration options and performed some modifications to the target engines. These will be discussed next.

JIT compilation usually only happens after the target function has been invoked thousands of times, which slows down automatic testing. As such, the target engines were configured to trigger JIT compilation much sooner, either by modifying the source code or by using

existing runtime flags. However, it is generally not desirable to compile a function as soon as possible, i.e. after a single invocation, as relevant type information might not fully be available. This would then cause the JIT compiler to behave differently. As such, the engines were configured to JIT compile a function after 10 to 100 invocations.

All of the target engines feature numerous internal assertions which are enabled in debug configurations of the software. These assertions are generally useful for detecting vulnerabilities as they tend to detect faulty conditions that otherwise do not result in memory corruption or otherwise externally observable misbehavior. However, the debug configuration of the software commonly lacks compiler optimization, which degrades performance significantly. As such, the target engines were built in a non-standard configuration in which all debug assertions were enabled but the software was otherwise built in a release configuration. AddressSanitizer[1] and similar compiler instrumentations were used as well but did not yield an increase in the number of detected faults.

Some vulnerabilities, e.g. use-after-free vulnerabilities, require garbage collection to trigger. As such, engines were configured to expose a function to JavaScript code which performed a full garbage collection. Furthermore, engines were configured to always perform garbage collection after executing a program from the fuzzer.

During testing our implementation was able to discover around 50 unique faults across all four major engines. These will be discussed in the next section.

6.2 Discovered Vulnerabilities

Most of the faults encountered during our tests were related to internal assertions which are enabled in debug configurations of the software. These do, however, not necessarily imply the existence of a security vulnerability. In general, a failed debug assertion can be the result of

- an internal error which is potentially security critical, thus, when exploited, can lead to remote code execution or leakage of sensitive information, or
- an internal error that causes observable misbehavior of the engine, such as incorrect results being computed, but is not a security issue, or
- an invalid assertion which rejects a condition that is in fact valid.

As such the identified faults have to be analyzed manually to determine whether they are the result of an underlying security vulnerability. This is often a non-trivial endeavor as it requires a good understanding of the surrounding code and the internals of the respective engine. Analysis of a subset of the identified faults resulted in the discovery of four previously unknown vulnerabilities which are presented in table 6.1. Section 6.3 will further explore one of the found vulnerabilities.

[1] https://github.com/google/sanitizers/wiki/AddressSanitizer

Identifier	Description
WebKit Bug 185328[a]	In JavaScriptCore, a bug in the JIT compiler's special handling of the Number.isInteger function. This would lead to a register being clobbered with controlled data during execution of the generated machine code. This vulnerability was reported while the vulnerable code was only in pre-release software. As such, no CVE was assigned for it. The bug was fixed with git commit eeb3a1d26b2dc2490e91fb2f897ddaa84fb3dfc4.
CVE-2018-4299[b]	In JavaScriptCore, a bug in the implementation of Proxy.apply. This bug would cause an engine-internal object, namely a JSLexicalEnvironment object, to be leaked to unprivileged JavaScript. As this object was only expected to be used in very specific ways, it was possible to cause memory corruption by using it in unintended ways. This resulted in an exploitable condition. The vulnerability was fixed with git commit 31fc3561a2b3e8c44b1ca123c418618b23a9d817.
CVE-2018-4359[c]	A bug in the JIT compiler's handling of floating point multiplication in certain special cases which would cause invalid machine code to be emitted. It was not investigated whether it was possible to obtain enough control over the emitted code to produce unintended but valid machine code instructions. This would likely lead to an exploitable situation. The vulnerability was fixed with git commit fb37d4711220140e3f24a6d34d4669fcb409da87.
CVE-2018-12386[d]	In Spidermonkey, a bug in the register allocator which would result in a register holding an unexpected value. This could be abused to cause a type confusion when executing the generated machine code, resulting in an exploitable condition. This vulnerability is further discussed in section 6.3. The vulnerability was fixed with git commit c0e48637d27b853d4a600b3c81d897b201421390.

[a] https://bugs.webkit.org/show_bug.cgi?id=185328
[b] https://www.zerodayinitiative.com/advisories/ZDI-18-1081/
[c] https://bugs.webkit.org/show_bug.cgi?id=187451
[d] https://www.mozilla.org/en-US/security/advisories/mfsa2018-24/#CVE-2018-12386

Figure 6.1 Four vulnerabilities that were identified with the presented approach. All vulnerabilities were initially discovered as assertion failures in a debug configuration of the software which were then manually analyzed.

6.3 Case Study: CVE-2018-12386

This section describes in more detail one of the identified vulnerabilities, CVE-2018-12386, a bug in the register allocator in one of the JIT compilers inside Firefox. It serves as an example of a vulnerability for which the initial sample resulted in a failed assertion in debug builds but no observable misbehavior in release builds, and which was subsequently manually modified to result in an exploitable condition.

The initial sample that triggered a failed assertion in debug builds is shown in listing 6.1. Running this code in a debug build of the Spidermonkey engine resulted in the following failed assertion: `"Assertion failure: *def->output() != alloc, at firefox/js/src/jit/RegisterAllocator.cpp:222"`. The vulnerability was analyzed by a small team and an exploit was written for it. It was then presented at the hack2win competition[2], during which it was disclosed to Mozilla and then quickly fixed in Firefox 62.0.3.

```
1  function f() {
2      function g() {}
3      let p = Object;
4      for (; p > 0; p = p + 0) {
5          for (let i = 0; i < 0; ++i) {
6              while (p === p) {}
7          }
8          while (true) {}
9      }
10     while (true) {}
11 }
12 f();
```

Listing 6.1: The initial sample for CVE-2018-12386 which was found during fuzzing.

The vulnerability occurs due to a special combination of control and data flow caused by the loops and essentially leads to a scenario in which the wrong value is stored in a register. This can be abused to cause a type confusion by loading a value of type X into a register that is expected to contain a value of type Y. Listing 6.2 demonstrates this. The code triggers the register misallocation in such a way that after the third loop in the function, the register that is expected to store parameter a now instead contains parameter b. The following code compiles to a store of a double value to the first argument. However, due to the register misallocation, it ends up writing a controlled double value (in this case the double value represented by 0x414141414141) to an object containing a pointer at that offset. As such, subsequent access to said property on the corrupted object leads to an attacker-controlled crash. Confusing different types of objects, such as ArrayBuffer objects, allows the construction of a memory read and write primitive. This in turn enables the execution of arbitrary code by corrupting control flow data inside the process.

[2] https://blogs.securiteam.com/index.php/archives/3765

```
1  // Generate objects with inline properties
2  for (var i = 0; i < 100; i++)
3      var o1 = {s: "foo", x: 13.37};
4
5  for (var i = 0; i < 100; i++)
6      var o2 = {s: "foo", y: {}};
7
8  function f(a, b) {
9      let p = b;
10     for (; p.s < 0; p = p.s)
11         while (p === p) {}
12     for (var i = 0; i < 10000000; ++i) {}
13     // Sets b.y to 0x414141414141 due to register misallocation
14     a.x = 3.54484805889626e-310;
15     return a.x;
16 }
17
18 f(o1, o2);
19 f(o1, o2);
20 f(o1, o2);
21 o2.y;        // Crashes at 0x414141414141
```

Listing 6.2: Proof-of-concept exploit for CVE-2018-12386 demonstrating a type confusion.

6.4 Potential Improvements

While the presented system is successful in discovering faults in scripting engines, it potentially has difficulties detecting specific types of vulnerabilities. In particular, callback related issues such as the ones shown in section 2.3.1 can be hard to discover with coverage-guided fuzzing. To understand why, it is necessary to look at the call stack during the execution of an unexpected callback. An exemplary call stack of such a situation is shown in figure 6.2. Here, the vulnerable function calls a helper function to perform a type conversion which then eventually triggers the callback. This is a common pattern for this type of vulnerability and is a problem for coverage guidance, as triggering the callback will likely not result in additional edge coverage if the helper function has previously performed the callback, as is likely the case for fundamental functions such as `JSValue::toNumber`. As such, a sample that managed to trigger a callback during execution of the vulnerable function would be discarded, preventing further mutations of it which could potentially trigger the vulnerability. A common workaround for this sort of issue is to use aggressive inlining during compilation of the target program: if the function that ultimately triggers the callback, in this case `JSValue::toNumber`, is inlined in the vulnerable function, a new control flow edge would be discovered when the callback is triggered and the sample would be kept for future mutations. During testing, the engine under test has been compiled with the second highest optimization level (-O3) which performs aggressive inlining. Observing the corpus during execution has shown that the implementation is in fact able to discover various situations in which a callback is triggered during execution of a builtin function implemented in C++. However, thoroughly evaluating the impact of more aggressive inlining remains the subject of future work.

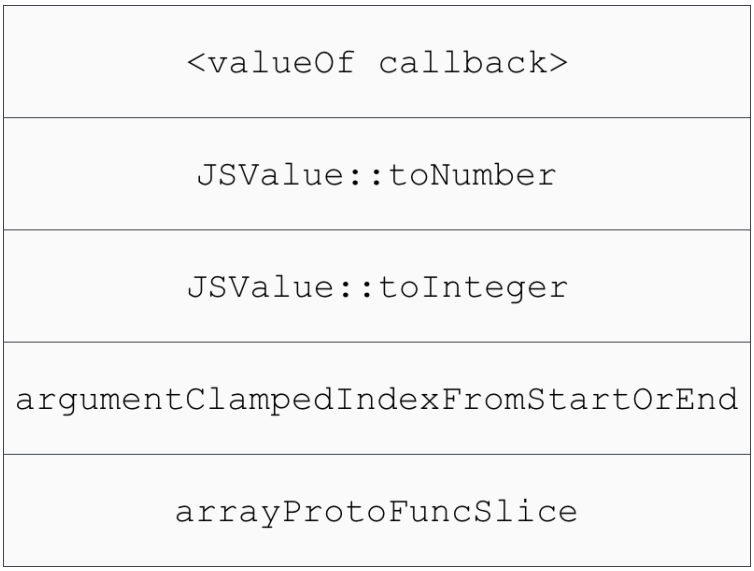

Figure 6.2 The call stack during the unexpected callback when triggering CVE-2016-4622 as described in section 2.3.1.

Similarly, vulnerabilities that require a non-trivial amount of additional logic to turn a faulty condition into an observable crash or assertion failure, such as the vulnerability described in section 2.3.2, can be hard to discover in an automatic fashion as well. With CVE-2018-4233, the vulnerable condition would occur as soon as a proxy that intercepts property loads was installed on a constructor function which was then JIT compiled. However, to obtain an observable fault, it was further necessary to modify the type of a function argument during an intercepted property load in a way that would lead to a type confusion in the compiled code afterwards. One possibility to improve the chances of detecting such issues is to instrument the engine to force a crash as soon as a faulty condition is observed. As an example, it might be possible to add runtime checks to the generated machine code that ensure that in fact no arbitrary JavaScript code has been executed as result of an IR operation that was deemed side effect free. This remains subject of future work as well.

Another weakness of the presented approach is that, due to the nature of coverage guided fuzzing, it focuses more on control flow and less on data flow of the program under test. As an example, integer overflow related issues can be hard to detect with this approach as there is no feedback for different integer values. A common workaround in this case is to include a list of ”dangerous” integers that commonly trigger overflows and to emit them more frequently.

7. Related Work

While automatic discovery of vulnerabilities in software that parses binary formats has been well studied, similar research for dynamic language interpreters is lacking. Further, most systems for automatic vulnerability discovery in JavaScript engines are not publicly available, as they are often either a source of revenue for the developer in the form of bug bounty programs, or are developed internally by browser vendors as part of the secure development life cycle [HL06] to improve the security of the product. Some of the research that has been published will be discussed in this chapter. Existing approaches are divided into ones based on static analysis, generative fuzzing approaches, as well as mutation based fuzzing approaches.

A system to statically analyze code of the binding layer of a web browser has been designed by Brown et al. [BNW+17]. In a browser, the binding layer makes it possible to access APIs that are implemented in the browser's native language, commonly C++, to be accessed from JavaScript. This involves converting JavaScript types to native types in both ways. The number of available APIs in a browser, commonly called Web APIs, has steadily grown over time[1]. The presented approach statically checks for various predefined vulnerability code patterns in the binding layer, such as for example unsafe type casts performed on JavaScript values. While the approach presented in this work is based on dynamic analysis, and is thus not directly comparable to a static analysis approach, it is more general, as it is able to target the engine itself, the binding layer, as well as the functionality that is exposed through the binding layer. This can be achieved by adjusting the coverage feedback, e.g. by only measuring coverage for the component that should be targeted. Furthermore, static analysis usually suffers from the fact that it can only detect vulnerabilities following a predefined pattern while fuzz-testing approaches can potentially discover new types of vulnerabilities.

There exist a number of generative fuzzers for JavaScript and in general web browsers. These systems follow more or less strictly the general architecture described in section 2.4.1. They are often targeted at the browser itself, thus generating HTML and CSS as well as code to interact with various Web APIs, and less on the core JavaScript engine. One example of

[1] https://developer.mozilla.org/en-US/docs/Web/API

such a fuzzer is Domato [Fra17], which has discovered a number of vulnerabilities across different browsers. Another popular generative fuzzer that is specifically targeted towards JavaScript engines is jsfunfuzz [Rud07]. It is able to cover a large part of JavaScript's language features and appears to be widely used. However, with generative approaches it is generally non-trivial to generate semantically valid samples as well as interesting control flow patterns, as it would often require some form of emulation of the code as well as precise knowledge of the runtime environment. As such, try-catch constructs are often used, which, however, influence the behavior of the tested engine. A different generative approach was presented by Hodovan et al. [HK16]. Their approach is based on a graph-based representation of the JavaScript environment which connects different language entities in the graph through different types of edges such as property edges, prototype edges, or return value edges. This graph is then used to produce semantically valid expressions with a high probability.

While it is possible to mutate the textual representation of code, mutation-based approaches have thus far been explored mostly on the abstract syntax tree (AST) of a program. An example for such a system was presented by Holler et al. [HHZ12], who mutate the AST of a program and replace nodes with compatible subtrees, called fragments, from other programs. Similar work has been performed by e.g. Guo et al. [Guo17] who perform AST mutations on JavaScript queries of the mongoDB database[2]. Vegallam et al. [VRHB16], similarly mutate the AST through replacements of nodes with subtrees from other samples or newly generated code. They further apply a guided fuzzing approach, however, they do not use coverage as a metric but instead score samples based on factors such as execution time and code complexity as measured by e.g. cyclomatic complexity [McC76]. This work sets itself apart by applying mutations to a different representation of a program as well as using coverage as a metric for a guided fuzzing approach.

[2]https://www.mongodb.com/

8. Summary and Future Work

This work introduced a new approach for automatically detecting security vulnerabilities in JavaScript interpreters using coverage guided fuzzing. For that, three requirements which the developed system had to meet were identified: first, it must be able to produce syntactically valid programs at all times. Second, it must be based on mutations so that guided fuzzing becomes possible. Third, it has to be able to generate semantically valid programs in high numbers to avoid the need for try-catch constructs.

The presented approach satisfies all three requirements. It is based on mutational fuzzing, however, instead of mutating syntactic structures of a program such as the abstract syntax tree, a new intermediate language, FuzzIL, was developed and mutations were defined on it. This makes it possible to more directly mutate the control and data flow of the program. A FuzzIL program can then easily be converted to JavaScript which guarantees syntactical correctness. By nature of a mutation-based approach, generated samples will often be semantically valid if the initial program was valid as well. As such, semantical correctness is achieved by discarding invalid programs: ones that either raise a runtime exception or trigger a timeout. Lightweight type tracking has additionally been implemented to further increase the correctness rate.

An initial implementation of the proposed system was able to discover multiple crashes and assertion failures across the major JavaScript engines. A preliminary analysis of the discoveries yielded four previously unknown and potentially exploitable vulnerabilities, including an exploitable bug in the register allocator of the just-in-time compiler inside Spidermonkey, the engine powering the Firefox web browser.

Throughout this work, room for future improvements has been identified. In particular, the ability to include existing JavaScript code, e.g. from regression tests, in the initial corpus is expected to improve the obtained results. Further, extending FuzzIL's coverage of JavaScript language features, for example by implementing asynchronous functions, should be equally important. Moreover, additional mutation algorithms that specifically target control flow could have a positive impact. An example of such would be a mutation that wraps existing code into a control flow construct such as a loop or moves existing code around between block borders.

Valuable insight could also be gained from a thorough evaluation of the impact of various parameters and configuration options used in the current implementation. These include parameters for the number of mutations, the maximum size of the corpus or the minimal number of mutations performed on a program. However, this will first require a definition of completeness of a fuzzing session, as a change of parameter could result in slower progression of the fuzzer but ultimately yield better results.

Another interesting topic of research is to evaluate whether coverage guidance on the machine code emitted by JIT compilers could be useful. This could for example help to discover vulnerabilities that only manifest for specific inputs to the compiled code. In the meantime, it is possible to work around this by defining mutations that specifically call existing functions with new inputs or changed environments.

Finally, an extensive evaluation across all major engines and with predefined conditions should be performed once a more mature implementation is available.

Bibliography

[AU77] Aho, Alfred V and Jeffrey D Ullman: *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[BJ78] Baker Jr, Henry G: *Actor Systems for Real-Time Computation.* Technical Report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1978.

[BNW+17] Brown, Fraser, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala and Deian Stefan: *Finding and preventing bugs in JavaScript bindings.* In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 559–578. IEEE, 2017.

[CFR+91] Cytron, Ron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman and F Kenneth Zadeck: *Efficiently computing static single assignment form and the control dependence graph.* ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4):451–490, 1991.

[CO08] Crocker, Dave and Paul Overell: *Augmented BNF for syntax specifications: ABNF.* Technical Report, 2008.

[DG94] Doligez, Damien and Georges Gonthier: *Portable, unobtrusive garbage collection for multiprocessor systems.* In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–83. ACM, 1994.

[DL93] Doligez, Damien and Xavier Leroy: *A concurrent, generational garbage collector for a multithreaded implementation of ML.* In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123. ACM, 1993.

[Fra17] Fratrić, Ivan: *Domato fuzzer.* https://github.com/googleprojectzero/domato, 2017. Accessed: 2018-10-28.

[GES+09] Gal, Andreas, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff et al.: *Trace-based just-in-time type specialization for dynamic languages.* ACM Sigplan Notices, 44(6):465–478, 2009.

[GKL08] Godefroid, Patrice, Adam Kiezun and Michael Y Levin: *Grammar-based whitebox fuzzing.* In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[Guo17] Guo, Robert: *MongoDB's JavaScript fuzzer.* Communications of the ACM, 60(5):43–47, 2017.

[HCU91] Hölzle, Urs, Craig Chambers and David Ungar: *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches.* In *European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[HG12] Hackett, Brian and Shu-yu Guo: *Fast and precise hybrid type inference for JavaScript.* ACM SIGPLAN Notices, 47(6):239–250, 2012.

[HHZ12] Holler, Christian, Kim Herzig and Andreas Zeller: *Fuzzing with Code Fragments.* In *USENIX Security Symposium*, pages 445–458, 2012.

[HJ84] Halstead Jr, Robert H: *Implementation of Multilisp: Lisp on a multiprocessor.* In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17. ACM, 1984.

[HK16] Hodován, Renáta and Ákos Kiss: *Fuzzing JavaScript Engine APIs.* In *International Conference on Integrated Formal Methods*, pages 425–438. Springer, 2016.

[HL06] Howard, Michael and Steve Lipner: *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[JL96] Jones, Richard and Rafael Lins: *Garbage collection: algorithms for automatic dynamic memory management*, volume 208. Wiley Chichester, 1996.

[KRC+18] Klees, George, Andrew Ruef, Benji Cooper, Shiyi Wei and Michael Hicks: *Evaluating Fuzz Testing.* In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[McC76] McCabe, Thomas J: *A complexity measure.* IEEE Transactions on software Engineering, (4):308–320, 1976.

[Rud07] Ruderman, Jesse: *JSFunFuzz fuzzer.* https://github.com/MozillaSecurity/funfuzz, 2007. Accessed: 2018-10-28.

[SGA07] Sutton, Michael, Adam Greene and Pedram Amini: *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[VRHB16] Veggalam, Spandan, Sanjay Rawat, Istvan Haller and Herbert Bos: *Ifuzzer: An evolutionary interpreter fuzzer using genetic programming.* In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[Zal15] Zalewski, Michał: *American Fuzzy Lop (AFL) fuzzer.* http://lcamtuf.coredump.cx/afl/, 2015. Accessed: 2018-10-28.