FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities

Samuel Groß
Google
saleo@google.com

Simon Koch
TU Braunschweig
simon.koch@tu-braunschweig.de

Lukas Bernhard
Ruhr University Bochum
lukas.bernhard@rub.de

Thorsten Holz
CISPA Helmholtz Center for Information Security
holz@cispa.de

Martin Johns
TU Braunschweig
m.johns@tu-braunschweig.de

***Abstract*—JavaScript has become an essential part of the Internet infrastructure, and today's interactive web applications would be inconceivable without this programming language. On the downside, this interactivity implies that web applications rely on an ever-increasing amount of computationally intensive JavaScript code, which burdens the JavaScript engine responsible for efficiently executing the code. To meet these rising performance demands, modern JavaScript engines ship with sophisticated just-in-time (JIT) compilers. However, JIT compilers are a complex technology and, consequently, provide a broad attack surface for potential faults that might even be security-critical. Previous work on discovering software faults in JavaScript engines found many vulnerabilities, often using fuzz testing. Unfortunately, these fuzzing approaches are not designed to generate source code that actually triggers JIT semantics. Consequently, JIT vulnerabilities are unlikely to be discovered by existing methods.**

In this paper, we close this gap and present the first fuzzer that focuses on JIT vulnerabilities. More specifically, we present the design and implementation of an intermediate representation (IR) that focuses on discovering JIT compiler vulnerabilities. We implemented a complete prototype of the proposed approach and evaluated our fuzzer over a period of six months. In total, we discovered 17 confirmed security vulnerabilities. Our results show that targeted JIT fuzzing is possible and a dangerously neglected gap in fuzzing coverage for JavaScript engines.

I. INTRODUCTION

The modern Web is unimaginable without JavaScript (JS). Driven by powerful JavaScript frameworks such as AngularJS [1], React [8], or jQuery [5], modern web content is typically created entirely on the client side, rather than being delivered in the form of HTML [10], [40]. This evolution caused increasing performance issues for existing JS engines that relied on simply *interpreting* JS code. As a result and to enable a dynamic web experience, modern web browsers have aggressively moved towards *just-in-time (JIT) compilation and optimization* of JS code. While JIT engines provide desirable performance improvements, they make the execution of JS code significantly more complex and inherently expose a large attack surface. Software vulnerabilities based on JIT compiler faults are attractive to attackers because they provide powerful exploit primitives and typically allow code execution based on a single vulnerability. An attacker can chain a successful attack with an escape from the browser sandbox, gaining unauthorized privileges by luring a victim to a malicious website.

By its very nature, JavaScript as a programming language is flexible and dynamic. Because of this flexibility, JIT optimizations require assumptions about the global state of the engine, groups of related objects, or even a single object involved in the optimized code segment. Such assumptions must either be proven true or protected by complex runtime mechanisms that notify the engine when a previously made assumption is violated. Any assumption that turns out to be false but remains undetected during execution represents a significant vulnerability, such as in CVE-2018-4233 (see Section III-B for details), a bug in the JIT compiler of WebKit. Consequently, JIT compilation bugs should be a focus of software testing.

A popular method for finding bugs in complex software systems such as JavaScript engines is fuzz testing (*fuzzing* for short). Fuzzing involves testing software with many different inputs and evaluating how the software responds to those inputs. The underlying hope is to find corner cases in the software that lead to non-trivial crashes. An analyst can then further investigate these crashes to create a proof-of-concept for an exploit that may break out of the JavaScript sandbox. In the past, fuzzing was mainly used to find vulnerabilities in JavaScript engines, and several critical problems with JavaScript engines were found [28], [28], [36], [43]. However, previous fuzzing approaches targeted JavaScript engines without focusing on specific components [28], [31], [43] or focused only on the runtime API [30]. Such approaches can find a wide range of vulnerabilities, but more complex vulnerabilities that require the concurrence of multiple preconditions have rarely been discovered. In particular, JIT compilation vulnerabilities are precisely such a type of vulnerability.

For JIT optimization to occur at all, certain conditions must be met: The engine must frequently execute the code in question, and the code must behave predictably during the observation because only then the JIT compilation starts. These conditions imply that not only must the JS code be structured in a special way to emphasize the faults, but it must also be executed numerous times in a similar manner and then change its behavior in an unpredictable way in order for the JS engine to encounter an error. This behavior is difficult to

Network and Distributed Systems Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24290
www.ndss-symposium.org

reproduce for a fuzzer that generates code snippets more or less randomly via mutations.

Classic JS fuzzers [4] generate JS constructs and wrap every statement in `try-catch` blocks because they cannot guarantee semantic correctness. Unfortunately, a JIT compiler treats code wrapped in `try-catch` differently from code that is not wrapped, so many JIT errors elude these approaches. Other work [28], [36], [43] generates test cases from existing JS corpora. Relying on pre-existing corpora requires a sufficiently diverse set of vulnerabilities of a specific type to extrapolate similar ones. This requirement limits the ability to uncover vulnerabilities dissimilar to existing test cases. In conclusion, developing a targeted fuzzing approach to detect novel faults in JIT compilation is a challenge that has not yet been tackled.

In this paper, we address this research gap in fuzzing coverage and propose the first fuzzer that uses an intermediate representation (IR) that focuses on discovering just-in-time compiler vulnerabilities in JavaScript engines. Our IR allows us to generate new JavaScript programs without initial input corpora, targeting the JIT compiler. Furthermore, our IR allows the implementation of semantically meaningful mutation operations, such as splicing multiple input programs while rewiring instruction operands, a feature missing in common AST-based fuzzing approaches.

We implemented the proposed approach and performed a comprehensive evaluation of our prototype on the major JS engines Apple JavaScriptCore, Google V8, and Mozilla SpiderMonkey. We find that our fuzzer compares well with Superion, a state-of-the-art open-source fuzzer [43], on all engines. Furthermore, we show that Superion cannot achieve significant code coverage gains when provided with a comprehensive input corpus. In contrast, our approach performs well in different kinds of setups and we identified 17 security-critical vulnerabilities.

Contributions. In summary, our main contributions are:

- We present the design and implementation of an IR-based fuzzing approach targeting JIT vulnerabilities in JS engines of modern web browsers.
- In a comprehensive evaluation, we discover 17 security-critical vulnerabilities with our prototype implementation. A more detailed analysis of the identified vulnerabilities confirmed that most of the faults are indeed related to the JIT compiler.
- We perform a comprehensive comparison against modern browser fuzzers and find that our approach outperforms the state-of-the-art method called *Superion*.

II. Background and Related Work

Fuzzing is a popular research area that has received much attention in the past years. In the following, we briefly introduce this area and discuss work that is closely related to ours. Given the enormous scope of this area, we cannot provide a comprehensive overview of all related work and thus focus mainly on related work that improves JavaScript fuzzing. For an introduction and overview of the fuzzing research area, we refer the reader to surveys on fuzzing [34] and greybox fuzzing [44]. For a comprehensive listing of recent fuzzing publications, we refer the reader to an online repository that maintains a list of papers published in this area [9].

A. Fuzzing Overview

Fuzzing can be divided at a high level into several different approaches, which we briefly explain below. These general approaches provide a rough classification, and in practice, many hybrids are used, so that a clear separation is not always possible.

a) Generative Fuzzing: Generative-based approaches [7] *generate* each input from scratch, using generator functions that output the relevant data. The main advantage of the generative approach is that the produced inputs are syntactically correct by design, since the generator functions respect the underlying syntax expected by the program under test.

b) Mutation Based Fuzzing: Mutation-based approaches use seed files and manipulate them according to certain rules, and then continue with the slightly modified files as new seed files. Mutations can be random and arbitrary, such as bit/byte flips or randomized addition/deletion of message parts, or more targeted, such as replacing integers or strings with data points known to have had problems in the past (e.g., magic values such as MAX_INT or MIN_INT).

Guided Fuzzing: Guided fuzzing [17], [18], [26], [45] extends the approach used in mutation-based fuzzing and prunes the mutated files based on relevance according to some metric (e.g., coverage-guided fuzzing). A popular metric for pruning in language fuzzing is branch coverage. A fuzzer using branch coverage collects data about the branches of the executed target program and deletes/ignores new mutated files for further consideration if they have not discovered any new branches during their execution. This approach ensures that the fuzzing process retains some momentum and does not reach a dead end.

Structure Aware Fuzzing: Depending on the underlying program to be fuzzed, especially if it requires a specific syntax as input, e.g., an interpreter, it is challenging to generate valid mutations of the input files. In particular, highly structured input data [16], [21], [25], [31], [38] such as source code can be challenging, as random changes are likely to result in input data that is immediately rejected, preventing in-depth testing. To counter this problem, the fuzzer can be made aware of the required input structure. In this paper, we explore the use of an intermediate language for this purpose.

B. JavaScript Fuzzing

Several previous publications cover general fuzzing of browser engines for JavaScript or stand-alone JavaScript engines, but so far, there has been no publication that focused on vulnerabilities in JIT compilers. Consequently, there is a gap in JavaScript fuzzer coverage that we fill with our approach. Note that previous JavaScript fuzzing work dealt with using intermediate representations for fuzzing JavaScript or semantically correct fuzzing of JavaScript, two properties that our fuzzer has (and needs) for its success in fuzzing JIT

compiler vulnerabilities. Hence we discuss in the following how our approach relates to previous work in this area.

One of the first works on this topic was presented by Holler et al., who proposed to use an abstract syntax tree (AST) as an intermediate representation [31]. During fuzzing, the subnodes of the AST are taken and replaced by nodes from other programs (or even newly generated code) and then translated back into the actual fuzzed language. Note that this process generates code that adheres to the syntax of the fuzzed language. However, no focus was placed on any particular bug class or semantic validity of the generated code. In our work, we do not use the AST as an intermediate representation. Instead, we develop our own intermediate language that represents a subset of JavaScript. Most importantly, it enables a focus on JIT errors and the semantic validity of the generated code.

Most recently, He et al. presented *SoFi* [29], a semantic-aware fuzzing approach. To ensure the validity of the generated test cases, the authors propose to use a fine-grained program analysis to identify variables and derive the types of these variables for mutation. In addition, SoFi uses an automatic repair strategy to fix syntactic and semantic errors in invalid test cases. Unfortunately, the full source code of SoFi is not publicly available, and hence we were not able to directly compare our approach to SoFi. Furthermore, we have reservations that the bugs discovered by SoFi are indeed security-critical (see Section VII-B2 for details).

Saxena et al. developed an intermediate language to canonicalize different JavaScript instructions (e.g., splitting a string) into a single action to improve fuzzing [39]. In this way, the small details of an implementation are abstracted into a higher-level, easier to handle representation. Their focus is on detecting client-side validation vulnerabilities in JavaScript applications, rather than the browser's JavaScript engine itself. Consequently, their fuzzing was based on the inputs of a JavaScript program rather than the inputs of a JavaScript engine (i.e., JavaScript code) itself. In a similar spirit, Hodován et al. created a graph-based representation of the JavaScript engine API and used this graph to generate input data for fuzzing [30]. However, they focus on the API provided by JavaScript engines and do not generate code beyond it. Consequently, the resulting code focuses on semantic correctness but is unlikely to detect JIT compiler bugs.

Montage, a fuzzer based on a neural network language model (NNLM), was proposed by Lee et al. [33]. They transform an AST into AST subtrees that can be used directly to train an NNLM. Using Montage, the authors found 37 bugs, including 3 CVEs. Although they found a JIT-related vulnerability, the overall approach is orthogonal to ours as they use machine learning on an AST. In contrast, we use predefined mutations on an IR. Moreover, they do not target JIT but perform extensive fuzzing of JavaScript engines in general.

Recently, Ta Dinh et al. presented *Favocado* [24], a fuzzer specializing on fuzzing binding layers in JavaScript code. They report that fuzzing such bindings requires both syntactic and semantic correctness to achieve the intended test area. This approach is similar to the challenges we faced in fuzzing the JIT-related code parts, which also requires high semantic and syntactic correctness. However, the targeted aspects of JavaScript engines are not comparable to ours, as we focus on software faults in JIT compilers.

To improve the semantic correctness of fuzzed inputs, Dewey et al. studied how *Constraint Logic Programming* (CLP) [23] can be used in this area. The authors use CLP to generate semantically valid code, which is a similar focus to ours, but they do not focus on JIT compiler vulnerabilities given that this class of software faults is particularly challenging to handle.

Wang et al. proposed *Skyfire*, a seed generation tool for fuzzing that requires a corpus of inputs and a grammar [42]. Based on this input, Skyfire learns a probabilistic context-sensitive grammar and uses this grammar to generate seed inputs. The authors show that their approach works well for highly structured languages, such as XML. However, they only provide preliminary results on JavaScript fuzzing, leaving future work to extend their approach "to better support more complex languages such as JavaScript and SQL." [42].

Han et al. presented *CodeAlchemist* [28], a generative fuzzer for JavaScript. Park et al. [36] proposed *DIE*, a novel method for exploiting hidden information, which they termed *aspects*, in input corpora. This method enables a fuzzer to generate more complex, and consequently more profound, test programs. They analyze the given input seed files and extract not only code snippets but also aspects of the code snippets, such as structure and runtime types. The proposed fuzzing method then uses this information to generate new code snippets containing the extracted aspects. Although the work features a type system similar to ours, the type information is applied to the AST layer instead of an IR. While, generally speaking, ASTs are capable of representing any valid JavaScript program, this abstraction layer is not ideal for implementing mutations. Analogous to code transformations used by modern compilers [32], [35], we apply our mutations on an IR layer. This design decision enables the implementation of semantically meaningful mutations that produce a high diversity of generated JavaScript programs, a crucial aspect for fuzzing.

In an orthogonal approach, Aschermann et al. proposed *Nautilus* [14], a multi-language fuzzer that combines an input grammar with code coverage. Mutations are applied at the AST layer, hence suffering from the aforementioned limitations. A more recent work on multi-language fuzzing called *Polyglot* [21] improves *Nautilus* by translating a seed corpus to a language-agnostic IR. Unfortunately, the mutations applicable to the IR are quite limited, e.g., not even basic language features such as variable definitions are amenable to mutation. In contrast, our specialization allows us to include highly specialized mutators and generators that specifically target code to trigger JIT routines. As a result, we find significantly more security-critical software vulnerabilities in real-world JIT engines used by major web browsers.

```
// addition function in C
int add(int a, int b) {
  return a + b;
}
// addition function in javascript
function add(a, b) {
  return a + b;
}
```

Figure 1: A simple addition function in C and JavaScript.

Regarding JavaScript-related security research without a specific focus on fuzzing, we refer to two recent survey papers [13], [41].

III. Just in Time Compiler Vulnerabilities

In this section, we first give a brief overview of JIT compilation for JavaScript in modern browsers, followed by a case study of a JIT vulnerability to illustrate technical challenges.

A. Just in Time Compilation

We use a small, intuitive example to give a brief introduction to the current approach to designing and implementing an efficient JS JIT compiler. More specifically, we explain the basic concept of a *mixed-mode JIT compiler architecture*, i.e., an interpreter as a baseline, followed by a sequence of successively higher optimizing JIT compilers. For a more detailed and comprehensive explanation of different JIT compilation approaches, including template- and trace-based JIT compilation, we refer the reader to common compiler and interpreter literature (e.g., [11], [15], [19], [20], [22], [27]).

JavaScript engines used in browsers contain a parser, a bytecode compiler, an interpreter, and usually a JIT compiler. When JavaScript code is first encountered, the parser of the engine constructs the corresponding AST, which is compiled to engine specific bytecode by the bytecode compiler [11]. This bytecode is consumed by an interpreter.

However, bytecode interpretation is slow due to the dispatching overhead as well as the numerous, often redundant, type checks being performed by each bytecode handler. If code is executed frequently, it is desirable to optimize the execution by compiling the JavaScript code to machine code and optimizing it on the way. An intuitive example of the challenges faced by a JavaScript JIT compiler in comparison with a classic ahead-of-time compiler (e.g., clang) can be seen in Figure 1: The given C code can directly be compiled into assembler code, as all required information is present. Contrary to C, JavaScript is dynamically typed and not all required information for machine code generation is present.

Consequently, a JIT compiler cannot trivially compile JavaScript code into machine code if performance is a requirement. The execution must first confirm the types used and then proceed according to the type at hand. Those types can range from primitive integers to highly complex objects, all exhibiting different behavior when used with the same functionality.

However, during execution, usage patterns become apparent. E.g., let us assume that the *add* operation is only observed being called with integers. Based on this observation, *speculative optimization* can be performed: The compiler compiles the JavaScript code specifically for the inferred type profile. Finally, type guards are added that represent the optimization type assumptions. The guards check that the given values are really of the assumed type (in our example, integers). As long as the guard holds, the code proceeds with the now optimized function. If the guard fails, the code 'bails out' and the execution of the JavaScript code returns to the interpreter, which executes the non-optimized, slower function. The resulting abstract assembler code would be similar to the compiled C code with the difference of containing the type guards and, as we are talking about integer addition, an overflow check.

We can summarize such an optimization into the following steps resulting in a compiled and optimized version of the function under scrutiny: (1) collect usage pattern data, (2) infer type patterns, (3) optimize the code for those types, (4) deploy type guards in front of the optimized code.

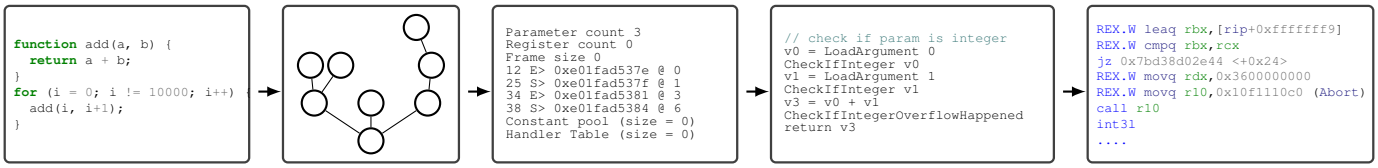
The reason why an engine does not immediately JIT compile the JavaScript code of a web page is twofold: First, the profiler has to collect execution information for JIT optimization to work. Second, JIT optimization is time-consuming because optimizing every aspect of a given JavaScript program might consume more time than is ever conserved by gains in execution speed.

To gather the required information before triggering JIT compilation, a profiler, which is part of the engine, collects execution information of the executed code. After reaching an internally specified threshold, the engine schedules the code for JIT compilation. Later executions directly call the optimized code instead of executing the function via bytecode interpretation.

Consequently, a typical modern JIT compiler pipeline consists of the steps visualized in Figure 2. (a) The engine translates the source code into an AST. (b) The engine compiles the AST into bytecode for a custom VM and executes this bytecode using the interpreter a number of times, during which type information is collected. (c) The engine passes the bytecode to the JIT compiler, which translates it into a compiler-specific IR. While the bytecode is designed for execution by the interpreter, the JIT IR is designed to facilitate the implementation of various program optimizations. (d) The JIT compiler optimizes the IR and adds type guards, which essentially add type information to the IR. (e) Finally, the JIT compiler lowers the IR to machine code, which is directly executed on the host CPU.

B. JIT Vulnerability Case Study

CVE-2018-4233 was one of the first vulnerabilities we discovered during the initial exploration of JIT compiler vulnerabilities. The JIT compiler tries to merge multiple type guards and fails to recognize that the type of the checked variable can change in-between.



(a) (b) (c) (d) (e)

Figure 2: High-level overview of the JIT compiler pipeline: (a) The engine gets the code and (b) translates it into an AST. (c) The AST is translated to bytecode that is executed by the compiler. (d) The JIT compiler translates bytecode into a JIT intermediate representation and optimizes it. (e) Finally, the JIT compiler lowers the IR to machine code.

```
function Constructor(a, v) {
  a[0] = v;
}

var trigger = false;
var arg = null;
var handler = {
  get(target, propname) {
    if(trigger) {
      arg[0] = {};
    }
    return target[propname];
  },
};
var EvilProxy = new Proxy(Constructor, handler);

for(var i = 0; i < 100000; i++) {
  new EvilProxy([1.1, 2.2, 3.3], 13.37);
}

trigger = true;
arg = [1.1, 2.2, 3.3];
new EvilProxy(arg, 3.54484805889626e-310);
arg[0];
```

Figure 3: Proof-of-concept code to trigger CVE-2018-4233.

1) Guard Redundancy Removal: The JIT code deploys guards (Section III-A) to ensure that all type assumptions made during compilation indeed hold at runtime. Missing any violated assumption may have severe consequences, ranging from crashes to exploitable vulnerabilities. Depending on the code, however, guards can be redundant. The JIT compiler can remove a redundant check to further optimize the code. To ensure that guards are redundant, the JIT compiler analyzes the code between guards for potential side effects. This analysis can be faulty, as was the case for CVE-2018-4233. A call to function deemed side-effect free could cause a user-defined JavaScript callback to be invoked, which in turn could change the type of a variable.

2) The Concrete Vulnerability: The compiler assumed that the $CreateThis$ operation, responsible for creating a new object in a constructor, would not result in any side effects. However, this assumption is violated by wrapping the constructor in a Proxy.

By being able to change 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.

Figure 3 shows a proof-of-concept to trigger this behavior. The JIT compiler assumes that the constructor function always receives an array with doubles as the first argument. It guards this assumption with a type check at the beginning of the emitted machine code. However, the $CreateThis$ operation is executed 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 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, is wrongly used as a pointer, resulting in an attacker controlled crash due to an access violation when dereferencing the address.

IV. Method

Fuzzing for JIT compilation vulnerabilities is an area that has not yet been explored in detail (see Section II) and requires special considerations concerning semantic correctness (see Section III). In this section, we describe our approach to fill the current gap in fuzzing JIT compilers for vulnerabilities. We start by defining a set of requirements that we deem necessary to successfully fuzz JIT compiler engines (Section IV-A) and then show how a fuzzer based on mutations of a custom IR can satisfy them (Section IV-B).

A. Requirements

1) Syntactic Correctness: The parsers of JavaScript engines are simple and easy to understand compared with the rest of the code base, and are not of interest to us. Additionally, the parser does not influence the JIT compiler. Consequently, our fuzzing approach needs to target the components *behind* the parser. Aiming at these components requires the syntactic correctness of emitted programs. Since the parsing phase rejects syntactically invalid examples, we ensure syntactically correct programs.

2) Guided Fuzzing: JIT compilers are deeply embedded within a JavaScript engine. The first element of the engine to get into contact with the code is the parser, then the interpreter, and only when the code is executed with the correct pattern is the JIT compiler triggered. To reach that deep into the engine, we require feedback to generate increasingly complex inputs stressing different features, eventually reaching the JIT compiler.

3) Semantic Correctness: As explained in Section III, JIT compiler optimization is only triggered in cases of repeatedly and reliably executed code. For such executions to occur, we need semantic correctness of the emitted code. Exceptions prevent the execution of subsequent code and JIT compilation altogether, because the engine does not execute the code often enough. Commonly, fuzzers work around this issue by wrapping every generated statement with a try-catch block. This approach does succeed in ensuring execution of subsequent code after the execution encountered an exception. Unfortunately, this substantially alters the program's semantics, as an additional control flow is introduced. Therefore, a JIT compiler treats the generated example differently than if no try-catch statements had been inserted. In fact, a JIT compiler cannot perform many optimizations when the control flow graph is fragmented, as with inserted try-catch blocks. We confirmed this assumption by adding try-catch constructs into programs found during fuzzing, which afterward stopped triggering the defect in most cases. Therefore, it becomes clear that a central requirement for successful fuzzing of JIT compilers is the ability to produce semantically correct code with a high likelihood.

4) Semantic Code Mutations: We determined that we want to use feedback and require semantic correctness for a successful fuzzing framework. One essential component is still missing: the underlying semantics of the code. A JIT compiler deals only with the semantic properties of the code, such as control and data flow. This is due to the JIT compiler commonly operating on its own IR of the bytecode without any knowledge of the initial AST and thus syntax. Consequently, it is desirable to perform mutations on that level and incorporate the feedback given from guided fuzzing.

The simplest way to use feedback is by using a mutation-based fuzzing approach. With this process, a fuzzer can dynamically add samples to the corpus that result in new coverage, and mutate them further in the future.

Existing mutation-based interpreter fuzzers, such as LangFuzz, mutate syntactic elements of the code, using representations such as the AST [31]. However, syntactic elements are irrelevant to the component targeted by our approach, the JIT compiler. Further, the AST can be ambiguous. Consequently, solely semantic mutations are more challenging to implement as immediate mutations could result merely in syntactic changes to the program and not semantic ones.

Figure 4 shows an example where two code snippets with different AST express the same computation. An AST-based mutation could simply be the transformation between those two code snippets. To counter this issue, we opted to use an intermediate representation that is close to the representation used by the compiler. Mutations on such an IR avoid semantically meaningless mutations and increase fuzzing effectiveness. By performing a different set of mutations on an IR, we can detect different defects faster. We note that an AST mutation could be restricted to counter meaningless mutations but would then effectively become an IR on its own.

```
// verbose code
function foo(b) {
  const a = 42;
  const c = a + b;
  return c;
}
// concise code
function foo(b) {
  return b + 42;
}
```

Figure 4: Example of a source code snippet showing a verbose and a concise version performing the same semantic operation.

```
v0 <- LoadInt  0
v1 <- LoadInt 10
v2 <- LoadInt  1
v3 <- Phi v0
BeginFor v0, <, v1, +, v2 -> v4
    v6 <- BinaryOperation v3, +, v4
    Copy v3, v6
EndFor
v7 <- LoadString 'Result: '
v8 <- BinaryOperation v7, +, v3
v9 <- LoadGlobal 'console'
v10 <- CallMethod v9, 'log', [v8]
```

Figure 5: An example of an IR program with a highlighted slice of the program.

B. An Intermediate Representation Designed for Fuzzing

As explained in the previous section, our fuzzing approach uses its own IR. Thus, we centered our fuzzer's design around the idea of mutating code in a custom intermediate representation (IR), then translating the IR code to JavaScript for execution. We designed our IR to our requirements as stated in Section IV-A:

IR Design: In our IR, a program consists of a list of instructions, each in turn consisting of an operation together with a list of input and output variables. Figure 5 shows an example program which computes the sum of the numbers from zero to nine. Note that IR operations can be *parametric*. Parameters include constants in operations, property and method names, the operators of binary and unary operations, as well as comparisons.

We implemented the control flow using special block instructions, for which at least a starting and an ending block exist. Our IR uses static single-assignment (SSA) form [12], [37], meaning that any variable has exactly one assignment. SSA form facilitates the implementation of a define-use analysis that we employ later on. It also increases the reliability of type inference and simplifies code generation because, e.g., output values will always be assigned to a new SSA variable. Reassignments of JavaScript variables are possible through a `Phi` operation that produces an output reassignable through a `Copy` instruction. We give a complete list of implemented IR operations in Appendix C, together with a description of the JavaScript language features that they cover.

Required Invariants: In addition, we require that the following five invariants must hold for every program in our IR, and thus for the JavaScript programs generated from it:

- *All inputs are variables:* All input values to an instruction must be variables. There are no immediate values or nested expressions. This enables more straightforward reasoning about the data flow of a program and facilitates mutations to it.
- *Variables are defined before use:* To reduce possible semantic errors, all variables must be defined before being used, either in the current block or an enclosing one.
- *No open semantic blocks:* A block beginning must eventually be followed either by the corresponding closing instruction or by an intermediate block instruction, such as a `BeginElse` for which the same holds true. This is necessary to guarantee syntactic correctness.
- *Inputs of blocks defined outside:* All inputs to block instructions must be defined in an outer block, reflecting the variable definition rules of JavaScript
- *Usage of Phi:* To preserve SSA semantics, the first input to a `Copy` instruction must be the output of a `Phi` instruction.

Lifting the IR to JavaScript: We lift a program from our IR to JavaScript by first translating each instruction in isolation. As a next step, we inline expressions when possible to create more human-readable code.

C. Mutating the IR

We designed the mutations in such a way that they modify the central aspects of a program expressed in our IR. In particular, we achieve the following four goals by our mutations:

- Mutation of the data flow between instructions (Input Mutation, Generative Mutations)
- Mutation of the computation performed by instructions (Operation Mutation)
- Mutation of the control flow of the program (Combine Mutation, Generative Mutation)
- Combination of aspects from two different programs (Combine Mutation)

In the following, we describe how these goals can be achieved via different kinds of mutations.

1) Input Mutation: The input mutation is a simple mutation to the data flow of a program. We replace one SSA input to an instruction with a different one. This causes the instruction to operate on another value at runtime, potentially yielding different results.

2) Operation Mutation: The operation mutation consists of selecting a random parameterized instruction and changing one of its parameters. For example, we change constant values, cause the program to use different methods or properties, or replace binary or unary operations.

3) Combine Mutation: The combine mutation combines parts of different programs into a new one: In the simple version of the mutation, we insert a program in full at a random position in the second program. This requires renaming of variables in the inserted program to avoid collisions of variable names, which is, however, trivially possible.

The more complex version of the mutation only inserts a part of an existing program into the second one. The mutation selects a random instruction and, recursively, all instructions whose outputs are also used as inputs. We then copy the resulting *slice* into another program. Figure 5 shows an example slice of the program. This slice could then simply be copied into a different program as it is self-contained.

However, this mutation does not alter any existing data flow, as the SSA variables of the two input programs are not mixed. To merge the data flows of the two input programs, an input mutation would need to happen afterwards.

4) Generative Mutation: The generative mutation simply inserts newly generated code, which makes use of existing values, at random positions into an existing program. For this purpose, we implemented several code generator functions that emit short code snippets. Overal, we implemented one simple code generator for every language feature of the IR as well as a small number of special code generators to either trigger JIT compilation or stress historically error-prone features.

Note that this mutator also ultimately makes it possible to commence from an initially empty corpus, as the mutator generates new code and programs increasing the code coverage, which our fuzzer then adds to the corpus dynamically.

D. Achieving a High Likelihood of Semantic Correctness

The invariants we impose on the IR—each being preserved by every mutation—avoid some trivial semantic errors, such as the use of a variable before it is defined. Those restrictions are, by themselves, not sufficient to ensure semantic correctness over the generated corpus. We added three additional measures to improve the semantic correctness.

1) Allowing only a valid corpus: We achieve an additional degree of semantic correctness by ensuring that only semantically valid samples are added to the corpus at runtime. In order to achieve that, it is necessary to not only record coverage information during every execution, but also whether the program terminated abnormally due to an uncaught runtime exception. In all supported engines, this is possible through the exit code, which will generally be zero if no uncaught exception was raised, and nonzero otherwise.

2) Only performing small changes: A pivotal insight to further improve the likelihood of semantic correctness is that each mutation only has a small probability of turning a valid (in the semantic sense) program into an invalid one. This is due to the fact that each mutation is either inherently semantically correct (combination mutations) or only affects the program in a minor way (input mutation, operation mutation, generative mutations).

3) A lightweight type system: Our final step to increase the semantic correctness of the generative component is a custom type system, as type errors are a significant source of semantic errors. For this, we implemented a lightweight abstract type inference engine. The inference engine, which can statically

approximate the runtime types of an SSA variable. This information is then used to avoid generating trivially invalid code constructs, such as method calls on non-objects or function calls on non-callable objects. In order to not limit the diversity of code that the fuzzer can produce, other mutations generally ignore type information.

We designed the type system to be as simple as possible, yet powerful enough to enable inference of the possible operations that can be performed on the value at runtime. The basic types supported are $T_{integer}$, T_{float}, T_{string}, $T_{boolean}$, $T_{function(signature)}$, $T_{constructor(signature)}$, $T_{object([properties],[methods])}$, $T_{undefined}$, and $T_{unknown}$

These types can be combined using the union operator, $t1|t2$, expressing that a value is one of multiple types. For example, the outcome of the addition operator in JavaScript can generally be a number or a string and would as such be represented as $T_{integer}|T_{float}|T_{string}$.

Further, two types can also be combined using the merge operator, $t1 + t2$. Such a merged type expresses that a value is two or more types *at the same time*. An example for this would be strings in JavaScript, as they expose properties and methods to the user. The type system thus represents them as $T_{string} + T_{object}$. Finally, the type system can also model the list of properties and methods of an object as well as the signatures of functions and constructors.

The abstract type inference engine has simple rules that determine the output types of every operation and operates on a static model of the runtime environment, containing type information for every built-in object. Whenever two or more alternative control-flow paths merge, we combine the variable states, using the union operator. The execution semantics differ slightly between JavaScript and our inference engine, e.g., the inference engine does not have a concept of prototypes as they exist in JavaScript. While simplifying the implementation, this leads to errors in the static type approximation. However, in practice, these turned out to be unproblematic, as the type approximation is used conservatively by the code generators.

As the static type inference system is merely a performance optimization, it can be disabled entirely, in which case the types of all variables will become $T_{unknown}$ and code generators will generate truly random operations. In practice, we found that the correctness rate varied between 50% and 75%.

E. Fuzzing on an Intermediate Code Representation

Our fuzzing approach generally follows the standard design of a mutation-based fuzzer. In every iteration, the fuzzer selects a program P from the existing corpus (seeded with a single-line JavaScript program) and mutates it randomly to produce a new program P_m. The fuzzer then lifts P_m to JavaScript code, which is subsequently executed on the target engine while gathering coverage statistics, e.g., through Clang's sanitizer-coverage feature.

If execution of P_m increases the coverage of the target program, P_m is regarded as interesting and kept for future mutations. However, as our mutations can only increase a program in size but never shrink it, it is necessary to minimize P_m before adding it to the corpus. Otherwise, the size of the programs in the corpus would keep increasing and slow down fuzzing. Minimization is naively possible through a fixpoint iteration that successively attempts to remove instructions while ensuring that the resulting program still exhibits the same coverage increase. As the distance between two interesting programs is often larger than a single mutation can bridge, we mutate a program multiple times consecutively. However, to prevent the unnecessary investment of resources, the last mutation is reverted if it produced an invalid program. Pseudocode for the high-level fuzzing algorithm is given in Algorithm 1.

Algorithm 1: Scheduling and mutating samples in FUZZILLI.

```
Corpus ← [];
Corpus.add(genSeedProgram());
while True do
    P ← Corpus.randomElement();
    for N do
        P_m ← mutate(P);
        exec ← execute(lift(P_m));
        if exec.returnStatus == crash then
            saveToDisk(P_m);
        else if exec.returnStatus == normal then
            P ← P_m;
            if newCoverage(exec) then
                P_min ← minimize(P);
                Corpus.add(P_min);
```

V. Experiment

We implemented the fuzzer design outlined in the previous section in the Swift programming language in a tool called Fuzzilli. We used this prototype implementation for the evaluation and ran it against the instrumented JavaScript engine code of three state-of-the-art JavaScript engines: Google V8, Apple JavaScriptCore, and Mozilla SpiderMonkey.

A. Fuzzing Time Frame

The vulnerabilities reported in this section are the results of a consecutive series of fuzzing sessions spanning six months. Each session lasted for around one week and used around 500 CPU cores. For each session, either the most recent source code version at that point or, if available, the source code of the current beta release was used. We ran the fuzzing on Google Compute Engine (GCE) and predominantly used multiple N1-standard-4 machine type machines (4 CPUs, 15 GB of RAM). Further, we chose to use preemptible instances to decrease the costs.

B. Setup

We compiled the target engines as standalone binaries, without the web browser bindings. Additionally, we modified the engines to support the target interface that requires

Table I: Summary of the found security vulnerabilities for Google V8, Apple JavaScriptCore (JSC), and Mozilla SpiderMonkey (SM). The table also presents more information about each vulnerability along two taxonomies (effect of the vulnerability and the time a vulnerability may occur). Finally, we list the age of the vulnerability in months.

Engine	Issue	Type	Runtime	Age	Description
SM	CVE-2019-9813	Type Safety	✓	18	Incorrect update to inferred property types
SM	CVE-2019-9791	Type Safety	✓	>36	Incorrect type inference for constructors entered via OSR
SM	CVE-2019-11707	Type Safety	✓	20	Incorrect return type inference of `Array.prototype.pop`
V8	Issue 944062	Type Safety	✓	1	`ReduceArrayIndexOfIncludes` fails to add Map checks
V8	Issue 944865	Type Safety	✓	8	Invalid value representation in V8
JSC	CVE-2019-8671	Type Safety	✓	9	LICM leaves object property access unguarded
JSC	CVE-2019-8765	Type Safety	✘	>6	GetterSetter type confusion during DFG compilation
V8	Issue 939316	Spatial Safety	✘	4	Optimizing `Reflect.construct` causes Map pointer OOB
JSC	CVE-2019-8518	Spatial Safety	✓	9	LICM moves array access before bounds check causing OOB
JSC	CVE-2019-8622	Temporal Safety	✓	>36	`doesGC()` incorrectly models behavior of StringObjects
JSC	CVE-2019-8672	Temporal Safety	✓	30	JSValue use-after-free in ValueProfiles
JSC	CVE-2019-8558	Temporal Safety	✘	>30	CodeBlock use-after-free due to dangling Watchpoints
JSC	CVE-2019-8623	Uninit Data	✓	24	LICM leaves stack variable uninitialized
JSC	CVE-2019-8611	Uninit Data	✓	4	Optimization incorrectly removes assignment to register
SM	CVE-2019-9792	Misc	✓	4	Leaks `JS_OPTIMIZED_OUT` magic value to script
SM	CVE-2019-9816	Misc	✓	>36	Unexpected ObjectGroup in ObjectGroupDispatch operation
V8	Issue 958717	Misc	✓	4	Incorrect interaction between DCE and inlining

communication between our fuzzer and the JavaScript engine over a set of communication pipes. Further, we lowered the JIT compilation thresholds to trigger JIT compilation earlier, thus speeding up the fuzzing. In general, we set the thresholds such that roughly 100 executions of a function would cause it to be compiled. This threshold allows a sufficient number of iterations for the engine to collect type information, while speeding up the fuzzing. Modifying the threshold is a common technique deployed by previous fuzzing solutions [3]. Finally, we compiled the engines in a custom debug configuration with optimizations enabled for performance reasons. The debug configuration includes a number of internal assertions, which are removed in release builds for performance reasons.

We enable assertions, as they help with detecting exploitable defects that do not immediately materialize as memory safety violations. As an example, CVE-2019-8622 was discovered through an assertion failure. The JIT compiler assumed that a specific operation could never cause a garbage collection (GC) to happen. However, during execution, the lowered operation did invoke APIs that can, under some circumstances, trigger GC. This situation can then be exploited by first triggering a GC at a specifically chosen time and then deliberately crafting JavaScript code so that a now freed JSObject is afterward accessed in the JIT compiled code.

Both of these steps are necessary to cause memory safety violation. As both steps require a non-trivial amount of specifically crafted code, they are unlikely to be directly found through fuzzing, but the indicators for such a violation are detected by fuzzing.

C. Results

Many of the vulnerabilities found first materialized as failed assertions or null pointer dereferences. Afterward, we performed manual triaging of the crashes to determine if they were security-critical and exploitable. While some observed crashes were clearly exploitable given the failed assertion or crash condition, others first required substantial analysis to determine exploitability. All identified bugs were reported to the developers in a coordinated way. We consider all defects listed to be exploitable and either received a CVE or Chrome Internal Issue number. Consequently, the issues were subsequently fixed by the developers.

1) Type Classification: Table I shows a comprehensive summary of all 17 vulnerabilities found which resulted in a CVE or internal issue number being assigned to us. All the identified vulnerabilities were related to JIT compilation and span across issues such as invalid bounds check removal, incorrect type inference, and register misallocation issues.

2) Age Determination: We also determined the age of the vulnerability by compiling old versions of the software and verifying whether the found test case triggers a crash. For JavaScriptCore and V8, we used git's bisecting feature to find the oldest revision. For Firefox, we used old official releases and tested on those.

As the test case might not trigger on a different version for unrelated reasons, or might even trigger another bug, the age results could contain inaccuracies, but are generally conservative. In cases where we could not determine the age of an issue dynamically as described above, we resorted to manual source code analysis to try to determine when the vulnerable code was introduced. As this is more error prone than compiling and testing the code, the results here are less certain.

VI. Classification of the found Vulnerabilities

To systematize the found vulnerabilities, we first describe the root causes that lead to each vulnerability, followed by two taxonomies. The first classifies the vulnerabilities according to the effect and the second by the time the vulnerability occurs. A tabular overview of classified vulnerabilities is shown in Table I.

A. Root Causes

We identified three common root causes: Optimizations that move code, incorrect modeling of runtime execution semantics, and faulty deletion of type checks. Root causes not fitting this categorization are grouped as miscellaneous.

1) Code Motion: A common optimization consists of moving code fragments in the program (e.g., loop-invariant code motion). However, if this is done incorrectly, previously safe code fragments become unsafe (3 vulnerabilities).

2) Incorrect Modeling: Issues may arise from faulty modeling of runtime execution semantics, such as whether an operation has side effects or could trigger garbage collection (2 vulnerabilities).

3) Incorrect Type Inference: A central optimization of JIT compilers is the inference of runtime type information, which allows type checks to be omitted. Whenever an incompatible value is stored in a property with associated type information, that type information has to be updated because JIT compilers rely on it to omit runtime type checks (4 vulnerabilities).

4) Misc: Not all found vulnerabilities shared a common underlying issue. This could be because they are "one-off" bugs or simply because no other similar vulnerabilities were found with which they could have formed a category (9 vulnerabilities).

Furthermore, the discovered vulnerabilities can be differentiated according to their *effect* and *time of impact*. Next, we briefly explain these two categories.

B. Classification by Effect

We determined that there were four common clusters (and a misc cluster) of different types of effects most found vulnerabilities could have:

1) Type Safety Violations: All vulnerabilities that cause some kind of type confusion (7 vulnerabilities).

2) Spatial Memory Safety Violations: All vulnerabilities that cause spatial memory corruption, such as out-of-bounds accesses to heap allocated memory blocks (2 vulnerabilities).

3) Temporal Memory Safety Violations: All vulnerabilities that cause temporal memory safety violations, e.g., due to usage of previously freed memory (3 vulnerabilities).

4) Usage of Uninitialized Data: All vulnerabilities that use uninitialized data, such as reading a pointer value from an uninitialized location in the stack (2 vulnerabilities).

5) Misc: All vulnerabilities that do not fit into any of the preceding categories (3 vulnerabilities).

C. Classification by Time of Impact

There are two different times a vulnerability may occur, either during runtime or during compile time.

1) Runtime: All vulnerabilities in this category are logical compiler flaws, possibly leading to memory corruption triggering in the emitted machine code at runtime (14 vulnerabilities).

2) Compile Time: This category includes "classic" memory corruption bugs as well as compiler-specific ones, which, lead to memory corruption during compilation (3 vulnerabilities).

VII. Efficacy of Our Method

Due to the non-deterministic nature of fuzzing, the objective properties of different fuzzing methods are difficult to compare. Another obstacle to performing a meaningful comparison is the different goals and design principles of fuzzers. We designed our approach called Fuzzilli specifically for finding JIT vulnerabilities and specialized accordingly. Moreover, our method does not require an input corpus and can generate new JavaScript code on its own. Other JavaScript fuzzers, such as Superion [43] or SoFi [29], have a more general design goal and require an input corpus. Consequently, the context of the design must be considered when comparing them.

To evaluate the efficacy of our method, we perform both a *descriptive* and an *empirical* analysis. We begin our descriptive evaluation by analyzing both the objective *generality* and *quality* of our approach. Then we present an empirical study in which we investigate the impact of our different generators, measure the code coverage of our method, and compare it to Superion [43]. To allow future replication, we make our code and artifacts openly available online.

A. Descriptive Efficacy Evaluation

During our experiments, we discovered and reported multiple previously unknown vulnerabilities in all three major JavaScript engines and were assigned corresponding CVE or Issue numbers (see Table I). These findings show that our approach achieves *generality* in terms of applicability, i.e., we did not optimize for a specific engine or benchmark and had a positive security impact on different highly relevant JavaScript engines. Furthermore, all three engines have been continuously tested by vendor-specific fuzzing infrastructures and by third-party fuzzers. Due to these prior efforts, JavaScript engines are generally considered well-tested software. Nevertheless, we found 17 vulnerabilities that eluded competing fuzzers, with some of the vulnerabilities introduced more than three years ago. This shows that our fuzzer is a significant *qualitative* improvement for JavaScript JIT fuzzing.

B. Empirical Efficacy Evaluation

Our empirical efficacy evaluation is two-fold. First, we analyze what effect each of our different generators had and how many mutations we are able to achieve across time for the three major JavaScript engines.

Second, to show that our fuzzer is competitive in the context of overall JavaScript fuzzing, we conduct an empirical evaluation against Superion [43]. This fuzzer represents the current state of the art in JavaScript fuzzing and the authors have shown that it outperforms other approaches in several dimensions.

1) Generator Effect Analysis: We ran our fuzzer five times for 24 hours against the three JavaScript engines SpiderMonkey, JavaScriptCore, and V8. During these runs, we logged the mutations resulting in code coverage increases. Figure 6 shows the results for JavaScriptCore and SpiderMonkey, the plot for V8 is shown in Appendix A due to page restrictions. We can observe that generative and input mutations

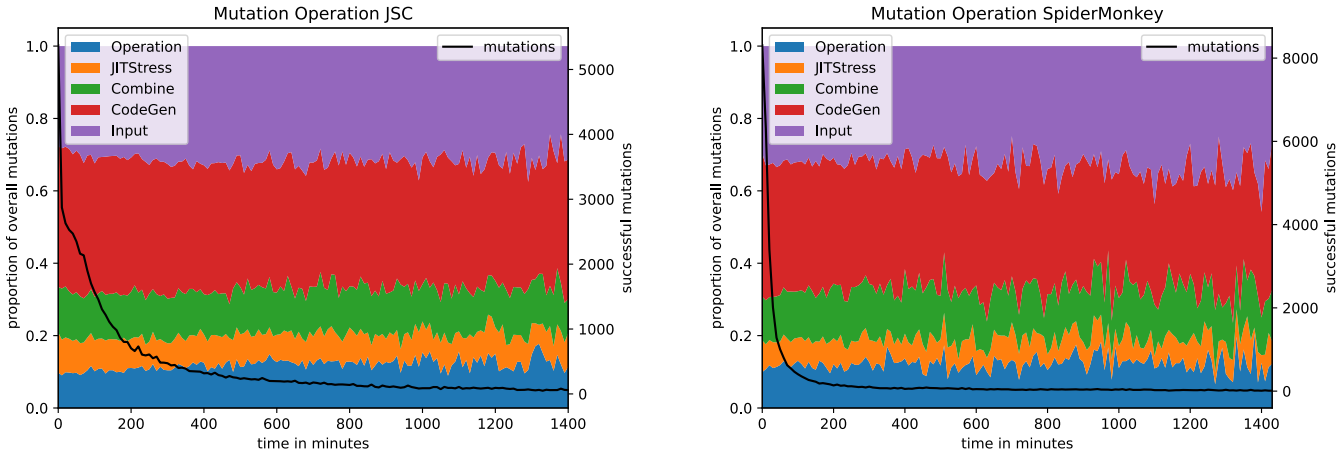


Figure 6: Temporal analysis of the proportion of our different mutation strategies for JSC and SpiderMonkey, respectively. Measured at 10 minute intervals.

were the most significant contributors when generating new samples. Combine and operation mutations are the next two influential mutations. Interestingly, explicitly stressing the JIT was the smallest contributor during our empirical analysis. These results show that when fuzzing for JIT vulnerabilities, there is no need to focus entirely on JIT-related mutations. A relatively small but constant effort suffices in practice. An in-depth explanation of the different mutation strategies can be found in Section IV-C.

There is also no noticeable difference across the different engines. The distribution of the successful mutators stays within the same proportions. This observation also holds for the number of mutation per minute that result in novel samples, which converge towards zero for both JSC and SpiderMonkey. However, we find that the initial decline is slower for JSC than for SpiderMonkey.

2) Comparison with SoFi: We would have preferred to include a direct comparison with the recently published SoFi [29] approach. Unfortunately, the authors neither published the full source code nor provided the code when contacted. We analyzed the results reported in Table 2 of the paper and found that the discovered "bugs" in the three relevant JavaScript engines SpiderMonkey, JavaScriptCore, and V8 do not seem to represent actual security-critical vulnerabilities. For example, the first four bugs reported for SpiderMonkey are marked as invalid by the developers and do not represent a vulnerability at all. The fifth report is a duplicate. Similarly, the "bug" reported for JavaScriptCore is marked as invalid by the developers, too. We were puzzled by this analysis of the reported results and the unavailability of the source code. Unfortunately, the concerns could not be resolved in a direct exchange with the authors, so no direct comparison was possible.

3) Comparison with Superion: A widely-used metric to compare fuzzers is code coverage, as it shows how much of an engine is reached and consequently tested. We opted to leverage branch coverage as our metric. As our fuzzer specializes in JIT fuzzing, we also compare the JIT-specific coverage. As noted above, our targeted engines are JavaScriptCore, V8, and SpiderMonkey, given that they are used in modern web browsers. The exact command line flags for each fuzzer JS engine are shown in Appendix B.

Setup: We ran each fuzzer five times for 24 hours on 100 cores of a Xeon Gold 5320 CPU with 256GB RAM using Ubuntu 22.04. Fuzzilli and Superion instances were deployed in virtual machines with 2GB of RAM for each of the 100 instances. For both fuzzers, corpus sharing was enabled.

Used Corpus: Our fuzzer does not use an input corpus, whereas Superion does require a corpus. This is a caveat when trying to perform a fair and objective comparison, as the corpus might determine the quality and final coverage of the fuzzing results and progress. A further hindrance is that Superion does not publish its corpus. We opted to use the publicly available DIE corpus [2] as input for Superion. To measure the impact of the start corpus on the success of Superion, we additionally evaluated on randomly chosen sub-corpora of DIE. We generated these sub-corpora by successively adding random samples until reaching 17% branch coverage, which is roughly half the coverage yielded by the entire DIE corpus. Each sub-corpus was used for a separate evaluation. We acknowledge that not using the original input corpus might result in worse results than previously reported. However, the fact that a fuzzer works well without a specific body is a characteristic that a fuzzer must have to be generic.

Evaluation: To evaluate the code coverage, we split the collected sample files into sets for each minute of fuzzing. Each set was evaluated against the corresponding llvm-cov [6] instrumented engine. We merged the resulting coverage data for each set across time, starting with the time-wise first set for Fuzzilli and the coverage of the input corpus for Superion.

The total coverage encompasses the coverage of the whole engine denoted by the llvm-cov report file as "TOTAL". To

Table II: Final mean branch coverage results. DIE Coverage denotes the coverage already reached by the provided DIE corpus prior to fuzzing. The first number represents the full corpus the second, the randomly selected sub-corpora.

Engine	DIE Coverage	Superion Coverage	Our Coverage
JSC	44.56% / 17%	46.71% / 28.14%	43.72%
JSC JIT	56.78% / 34.98%	57.67% / 49.35%	59.22%
V8	35.17% / 17%	36.15% / 23.13%	30.64%
V8 JIT	53.33% / 38.82%	54.22% / 47.93%	53.47%
SM	36.49% / 17%	38.96% / 23.74%	30.53%
SM JIT	59.28% / 43.82%	60.50% / 52.69%	56.27%

gain the JIT specific coverage, we extracted and averaged the reported coverage for each file potentially involved in JIT compilation, using regular expressions on the individual file paths[1].

Results: We evaluated the branch coverage and averaged it across our five runs. Superion improves the initial *overall coverage* of the full DIE input corpus by 2.15%, 0.98%, and 2.47% for JavaScriptCore, V8, and SpiderMonkey, respectively. For the 17% coverage sub-corpus, Superion improves the coverage by 11.14%, 6.13%, and 6.74%. Our fuzzer reached a final coverage of 43.72%, 30.64%, and 30.53%. Concerning *JIT specific coverage*, Superion improves by 0.89%, 0.89%, and 1.22%. For the 17% coverage sub-corpus the improvements are 14.37%, 9.11%, and 8.87%. Our fuzzer reached a final coverage of 59.22%, 53.47%, and 56.27%.

We also evaluated the line coverage specifically for JavaScriptCore, as this is the intersection of evaluated engines by us and Wang et al., as well as their reported metric [43]. Using the full DIE Corpus with an initial line coverage of 52.01%, Superion improves by 1.44%. For the partial corpus, Superion improves by 11.49%. Our fuzzer reaches a line coverage of 49.51%. For JIT-specific coverage, Superion improves the full DIE corpus with an initial coverage of 64.58% by 0.53%. For the partial corpus, with an initial coverage of 44.10%, Superion improves by 14.13%. Our fuzzer reaches a JIT specific line coverage of 65.60%.

The plots showing branch coverage over time are given in Figure 7. A tabular overview of the raw branch coverage results can be found in Table II. Figure 8 visualizes the comparison along line coverage.

Discussion: Line and branch coverage are similar for JavaScriptCore, leading to the assumption that either metric is interchangeable. Overall, Superion outperformed our fuzzer when provided with the full DIE corpus. The final coverage of Superion was 3%, 6%, and 8% larger than our final coverage. However, the coverage that was achieved in addition to the start corpus was marginal, and the DIE corpus itself already reached a higher coverage than the final coverage of our fuzzer. Consequently, the better coverage cannot be attributed to Superion.

[1]JSC:'. $*$ $/(b3|ftl|dfg|assembler|jit)/.*$'$,$V8:'. $*$ $src/compiler/.*$', SpiderMonkey:'. $*$ $js/src/jit/.*$'

An interesting observation is that Wang et al. [43] reported a line coverage increase of Superion for WebKit/JSC from 52.4% to 78.0%, an additional 25.6% of coverage. However, using the DIE corpus with an initial line coverage of 52.01% only lead to a final line coverage of 53.45%, an addition of merely 1.44% (ref. Figure 8). Our assumption is that this is due to the difference in input corpora and should be considered a demonstration of the effect different input corpora can have on the final coverage.

We strictly outperformed Superion when it was only provided with the reduced corpus. Our fuzzer reached an additional coverage of 15%, 7%, and 7% for JavaScriptCore, V8, and SpiderMonkey. In contrast to the large corpus, Superion could noticeably improve on the coverage again, showing the impact an initial corpus can make on the results. However, those improvements are of questionable benefit, as they are a subset of the already existing larger DIE corpus, which largely consists of browser vendor test cases. Thus, no overall improvement in testing of JavaScript engines has been done.

Concerning JIT specific coverage, we outperformed Superion on a full DIE corpus for JSC by 1.6% and got outperformed by 0.8% and 3.8% for V8 and SpiderMonkey. But again, the initial coverage by the start corpus itself was already higher than our final coverage and the added coverage by Superion was marginal. However, this demonstrates that we are able to outperform or compete with Superion in terms of JIT focused fuzzing even if Superion is supplied with a comprehensive start corpus. The reduced start corpus lead to similar results for JIT coverage as it did for full coverage.

C. Lessons Learned

We were able to compete with and, for JSC, even outperform Superion when it comes to JIT coverage even when providing Superion with the full DIE corpus. Surprisingly, when looking at the overall coverage, Superion barely improved on the full DIE corpus. Concerning a reduced initial corpus, we strictly outperformed Superion in both JIT and general coverage.

Finally, we were unable to reproduce the reported 25.6% improvement of line coverage for JSC even though we also started at an initial line coverage of 52%. This is perplexing to us, as this suggests that the corpus used by Wang et al. had significant and reachable coverage gaps that could be filled but that do not exist in the DIE corpus. As the original Superion corpus has not been published, we were unfortunately not able to try and reproduce their original results. As a consequence, we emphasize that future fuzzing research has to provide any initial corpus to enable reproduction, and more emphasis should be put on reproduction of previous fuzzing results with different, possibly novel, corpora to estimate how well a fuzzer generalizes across different corpora.

Concerning the influence of different mutation strategies, we showed that a small but constant effort to stress the JIT suffices to be able to focus on JIT vulnerabilities.

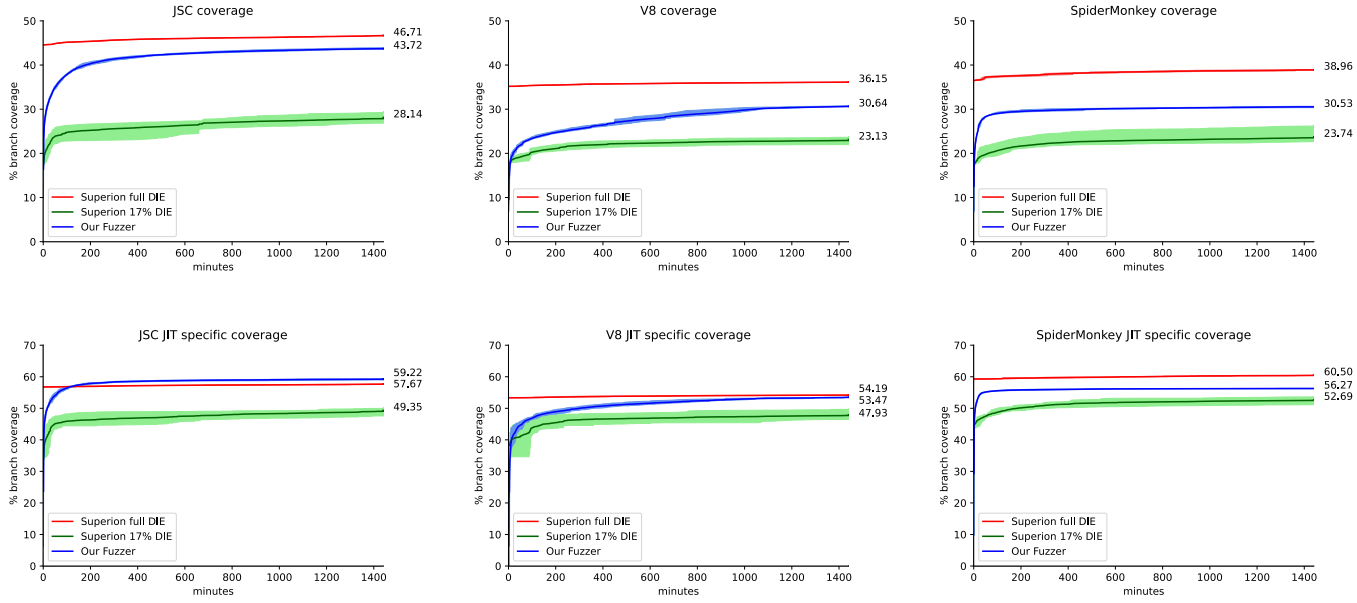


Figure 7: The first row shows the comparison of overall coverage, the second row shows the JIT specific coverage. The engines are shown in the order JavaScriptCore, V8, and SpiderMonkey. The main line displays the mean across all runs, whereas the shaded area denotes the confidence band ranging from minimum value to maximum value.

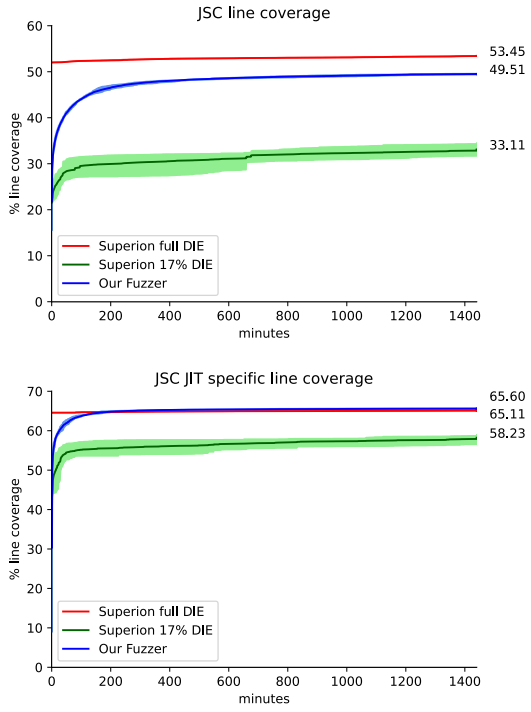


Figure 8: Line coverage evaluation for the overall engine (top) and concerning JIT specific files (bottom).

VIII. Conclusion

During the course of this paper, we showed how JIT compilation can lead to serious vulnerabilities and why current fuzzing approaches are insufficient to detect such vulnerabilities. We proposed filling this fuzzing gap by our novel approach of generating semantically correct code, leveraging an IR with a proportion of JIT focused mutation strategies.

We implemented our approach in the swift programming language and ran a 6-months experiment on 500 cores against V8, SpiderMonkey, and JavaScriptCore. During this test time frame we discovered 17 previously unknown vulnerabilities. Those vulnerabilities were on average at least 16 months old and were therefore also overlooked by the fuzzers of the respective vendors and researchers.

To foster research and strengthen the security of JS engines, we will open source our code.

We also performed a descriptive and empirical analysis of our fuzzer. In our descriptive analysis we showcased how our fuzzer generalizes well across different engines and was able to find previously unknown vulnerabilities, showing its qualitative improvements of the state of the art. Our empirical analysis showed that a constant, but limited, mutation focus for the JIT is deployed by our fuzzer. Furthermore, the empirical analysis showed that we are able to outperform or compete with the state of the art fuzzer Superion when it comes to JIT focused fuzzing even when providing Superion with a comprehensive start corpus. When reducing the start corpus, Superion was unable to outperform us, neither for general nor for JIT focused coverage across all three engines. Those results underline the importance to test fuzzer leveraging

input corpora across multiple different corpora to judge the generality. We also call for all future fuzzing research to be required to publish not only source code but also used evaluation corpora.

However, our fuzzing methodology is not complete as there is still room to improve type information, e.g., by instrumenting the emitted code, to be more exact, which would allow for even more targeted code generation. Furthermore, our set of mutations is limited. Increasing the set of mutations to include focused mutations for control flow could yield even more deeply hidden vulnerabilities. Also the set of special features can still be increased, as JavaScript is a complex and feature rich language and less popular features might contain yet undiscovered issues.

Availability

All of our research artifacts are available online at https://github.com/evaluating-fuzzilli-for-js-jit-fuzzing and the fuzzer itself is available at https://github.com/googleprojectzero/fuzzilli.

Acknowledgements

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, by the German Federal Ministry of Education and Research (BMBF, project KMU-Fuzz – 16KIS1523), and by the European Union's Horizon 2020 research and innovation program under grant agreement No 101019206.

References

[1] AngularJS - Superheroic JavaScript MVW Framework. [software], https://www.google.com/search?client=safari&rls=en&q=angular+js&ie=UTF-8&oe=UTF-8. visited 2022-07-29.

[2] DIE corpus. https://github.com/sslab-gatech/DIE-corpus.git. visited 2022-04-09.

[3] GitHub: funfuzz shell flags. https://github.com/MozillaSecurity/funfuzz/blob/master/src/funfuzz/js/shell_flags.py#L99. visited 2022-07-29.

[4] Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. visited 2022-07-29.

[5] jQuery: The Write Less, Do More, JavaScript Library. [software], https://jquery.com. visited 2022-07-29.

[6] llvm-cov - emit coverage information – llvm 16.0.0git documentation. https://llvm.org/docs/CommandGuide/llvm-cov.html. visited 2022-07-27.

[7] Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/. visited 2022-07-30.

[8] Reactjs. https://reactjs.org/. visited 2022-06-07.

[9] Recent papers related to fuzzing. https://wcventure.github.io/FuzzingPaper/. visited 2022-07-29.

[10] Web technology for developersjavascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. visited 2022-07-29.

[11] Alfred Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[12] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1988.

[13] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5):66:1–66:36, September 2017.

[14] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[15] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, jun 2003.

[16] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[19] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, 2009.

[20] Stefan Brunthaler. Inline caching meets quickening. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[21] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *IEEE Symposium on Security and Privacy*, 2021.

[22] Anshuman Dasgupta. *Tailoring traditional optimizations for runtime compilation.* Rice University, 2007.

[23] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *ACM International Conference on Automated Software Engineering (ASE)*, 2014.

[24] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. Favocado: Fuzzing the binding code of javascriptengines using semantically correct test cases. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[25] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[27] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[28] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[29] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[30] Hodová, Renáta, and Ákos Kiss. Fuzzing javascript engine apis. In *International Conference on Integrated Formal Methods - Volume 9681*, 2016.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, 2012.

[32] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[33] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *USENIX Security Symposium*, 2020.

[34] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[35] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179. Citeseer, 2003.

[36] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy*, 2020.

[37] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1988.
[38] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In *USENIX Security Symposium*, 2021.
[39] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.
[40] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security. In *USENIX Security Symposium*, 2017.
[41] Kwangwon Sun and Sukyoung Ryu. Analysis of javascript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, August 2017.
[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy*, 2017.
[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2019.
[44] Pengfei Wang and Xu Zhou. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *corr/arxive*, 2020.
[45] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

APPENDIX A
V8 GENERATOR EFFECT PLOT

Figure 9 shows the fraction of applied mutations over time in V8. We observed that the distribution is similar to SpiderMonkey and JavaScriptCore (see Figure 6). Note that the total number of mutations is lower from the beginning, but does not converge to zero towards the end, suggesting that V8 would benefit from longer fuzzing runs as it does not exhaust its exploration paths as quickly.

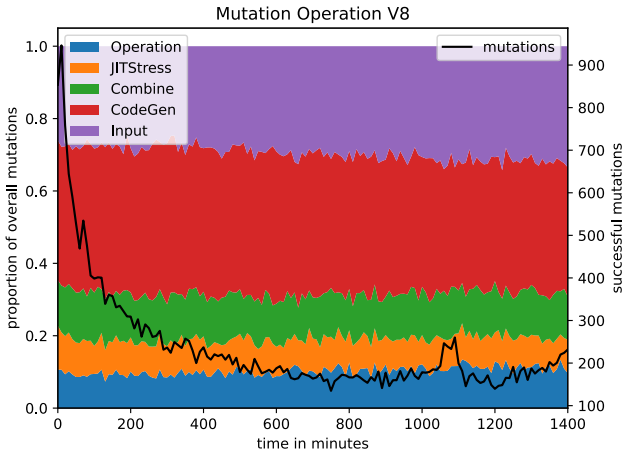


Figure 9: Temporal analysis of the proportion of our different mutation strategies for V8, measured at 10 minute intervals.

APPENDIX B
JS ENGINE FLAGS

The following listings summarize the command line flags used during the experiments to start the code coverage evaluation. As Superion was fed with the DIE corpus, we supplied the evaluation with the required setup JavaScript files $jsc.js$, $ffx.js$, and $v8.js$. Note that Superion did benefit from changes in commandline parameters, similar to Fuzzilli. We made sure that all fuzzers are evaluated in a fair and objective way.

SpiderMonkey:

a) Fuzzilli:

```
--fuzzing-safe
--no-threads
--baseline-warmup-threshold=10
--ion-warmup-threshold=100
--ion-check-range-analysis
--ion-extra-checks
```

b) Superion:

```
--fuzzing-safe
--no-threads
--fast-warmup
-f /chakra.js -f /ffx.js -f /jsc.js -f /v8.js
```

JavaScriptCore:

c) Fuzzilli:

```
--validateOptions=true
--thresholdForJITSoon=10
--thresholdForJITAfterWarmUp=10
--thresholdForOptimizeAfterWarmUp=100
--thresholdForOptimizeAfterLongWarmUp=100
--thresholdForOptimizeSoon=100
--thresholdForFTLOptimizeAfterWarmUp=1000
--thresholdForFTLOptimizeSoon=1000
--validateBCE=true $file
```

d) Superion:

```
--useConcurrentJIT=false
--useConcurrentGC=false
--thresholdForJITSoon=10
--thresholdForJITAfterWarmUp=10
--thresholdForOptimizeAfterWarmUp=100
--thresholdForOptimizeAfterLongWarmUp=100
--thresholdForFTLOptimizeAfterWarmUp=1000
--thresholdForFTLOptimizeSoon=1000
-f /chakra.js -f /ffx.js -f /jsc.js -f /v8.js
```

V8:

e) Fuzzilli:

```
--expose-gc
--fuzzing
--allow-natives-syntax
--interrupt-budget=1024
```

f) Superion:

```
--expose-gc
--fuzzing
--predictable
-f /chakra.js -f /ffx.js -f /jsc.js -f /v8.js
```

APPENDIX C
OPERATIONS AND GENERATORS

Table III: Overview of the operations implemented in our IR and the corresponding JavaScript language feature that they cover.

Operation	Covered JavaScript Language Feature
Nop	Empty statement (does nothing)
LoadInteger	A number literal containing an integer value
LoadFloat	A number literal containing a floating point value
LoadString	A string literal
LoadBoolean	A boolean literal
LoadUndefined	The undefined value
LoadNull	The null value
CreateObject	An object literal
CreateArray	An array literal
CreateObjectWithSpread	An object literal possibly using spread syntax
CreateArrayWithSpread	An array literal possibly using spread syntax
LoadBuiltin	Variable access (builtin objects are accessible through global variables)
LoadProperty	Property access using the dot notation
StoreProperty	Property access using the dot notation
DeleteProperty	Property access using the dot notation
LoadElement	Property access using the bracket notation (with a constant integer as property name)
StoreElement	Property access using the bracket notation (with a constant integer as property name)
DeleteElement	Property access using the bracket notation (with a constant integer as property name)
LoadComputedProperty	Property access using the bracket notation
StoreComputedProperty	Property access using the bracket notation
DeleteComputedProperty	Property access using the bracket notation
TypeOf	The typeof operator
InstanceOf	The instanceof operator
In	The in operator
BeginFunctionDefinition	A plain function
Return	The return statement
EndFunctionDefinition	A plain function
CallMethod	A method call
CallFunction	A function call
Construct	A constructor call
CallFunctionWithSpread	A function call possibly using spread syntax
UnaryOperation	A unary operation
BinaryOperation	A binary operation
Phi	Variable definition/assignment
Copy	Variable assignment
Compare	A comparison operation
BeginWith	A with statement
EndWith	A with statement
LoadFromScope	Variable access (in a with statement, properties of the context object become local variables)
StoreToScope	Variable access (in a with statement, properties of the context object become local variables)
BeginIf	If statement
BeginElse	If statement
EndIf	If statement
BeginWhile	While loop
EndWhile	While loop
BeginDoWhile	Do-While loop
EndDoWhile	Do-While loop
BeginFor	For loop
EndFor	For loop
BeginForIn	For-In loop
EndForIn	For-In loop
BeginForOf	For-Of loop
EndForOf	For-Of loop
Break	Break statement
Continue	Continue statement
BeginTry	Try-Catch statement
BeginCatch	Try-Catch statement
EndTryCatch	Try-Catch statement
ThrowException	Throw operation

Table IV: A complete list of all code generators used and a brief description. If a generator emits a block, that block is filled with the result of another code generator invocation.

Name	Description
IntegerLiteralGenerator	Loads a random integer
FloatLiteralGenerator	Loads a random floating point number
StringLiteralGenerator	Loads a random string
BooleanLiteralGenerator	Loads a random boolean
UndefinedValueGenerator	Loads the undefined value
NullValueGenerator	Loads the null value
BuiltinGenerator	Loads a reference to a random builtin
ObjectLiteralGenerator	Generates an object literal
ArrayLiteralGenerator	Generates an array literal
ObjectLiteralWithSpreadGenerator	Generates an object literal using spreading syntax
ArrayLiteralWithSpreadGenerator	Generates an array literal using spreading syntax
FunctionDefinitionGenerator	Defines a new function
FunctionReturnGenerator	Generates a return statement
PropertyRetrievalGenerator	Loads a random property on an existing value
PropertyAssignmentGenerator	Stores an existing value as random property on an existing value
PropertyRemovalGenerator	Deletes a random property of an existing value
ElementRetrievalGenerator	Loads a random indexed element of an existing value
ElementAssignmentGenerator	Stores a random value as indexed element on an existing value
ElementRemovalGenerator	Deletes a random indexed element from an existing value
TypeTestGenerator	Performs the JavaScript typeof operator on an existing value
InstanceOfGenerator	Performs the JavaScript instanceof operator on existing inputs
InGenerator	Performs the JavaScript in operator on existing inputs
ComputedPropertyRetrievalGenerator	Loads a computed property of an existing value
ComputedPropertyAssignmentGenerator	Stores a computed property of an existing value
ComputedPropertyRemovalGenerator	Removes a computed property from an existing value
FunctionCallGenerator	Calls an existing function with existing values as arguments
FunctionCallWithSpreadGenerator	Calls an existing function using the JavaScript spreading syntax
MethodCallGenerator	Calls a random method on an existing value
ConstructorCallGenerator	Performs a constructor call on an existing function
UnaryOperationGenerator	Performs a random unary operation on an existing value
BinaryOperationGenerator	Performs a random binary operation on two existing values
PhiGenerator	Creates a phi variable with an existing value as initial value
ReassignmentGenerator	Reassigns an existing phi variable to a different, existing value
WithStatementGenerator	Generates a JavaScript with statement
LoadFromScopeGenerator	Inside a with statement, load a property of the context object
StoreToScopeGenerator	Inside a with statement, store a property of the context object
ComparisonGenerator	Generates a random comparison operation of two existing values
IfStatementGenerator	Generates an if-else statement
WhileLoopGenerator	Generates a while loop
DoWhileLoopGenerator	Generates a do-while loop
ForLoopGenerator	Generates a for loop
ForInLoopGenerator	Generates a for-in loop
ForOfLoopGenerator	Generates a for-of loop
BreakGenerator	Generates a break statement
ContinueGenerator	Generates a continue statement
TryCatchGenerator	Generates a try catch statement with the result of another generator as bodies
ThrowGenerator	Throws an existing value as exception
WellKnownPropertyLoadGenerator	Loads one of the well-known Symbol properties of an existing value
WellKnownPropertyStoreGenerator	Stores to one of the well-known Symbol properties of an existing value
TypedArrayGenerator	Constructs a JavaScript typed array
FloatArrayGenerator	Constructs a regular JavaScript array containing only floating point numbers
IntArrayGenerator	Constructs a regular JavaScript array containing only integers
ObjectArrayGenerator	Constructs a regular JavaScript array containing objects
PrototypeAccessGenerator	Retrieves the prototype of an existing value
PrototypeOverwriteGenerator	Changes the prototype of an existing value to another existing value
CallbackPropertyGenerator	Installs a valueOf or toString callback on an existing value
PropertyAccessorGenerator	Generates a property getter and setter on an existing value
ProxyGenerator	Generates a JavaScript proxy object
LengthChangeGenerator	Stores a numeric value as .length property on an existing value
ElementKindChangeGenerator	Stores an object value as indexed element into an existing value