OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers

Jiming Wang[1,2], Yan Kang[1,2], Chenggang Wu [1,2,3*], Yuhao Hu[1,2], Yue Sun[1,2], Jikai Ren[1,2], Yuanming Lai[1], Mengyao Xie[1], Charles Zhang[4], Tao Li[5], and Zhe Wang[1,2,3]

[1]SKLP, Institute of Computing Technology, CAS
[2]University of Chinese Academy of Sciences
[3]Zhongguancun Laboratory
[4]Tsinghua University
[5]Nankai University

Abstract

Just-In-Time (JIT) compiler is a core component of JavaScript engines, which takes a snippet of JavaScript code as input and applies a series of optimization passes on it and then transforms it to machine code. The optimization passes often have some assumptions (e.g., variable types) on the target JavaScript code, and therefore will yield vulnerabilities if the assumptions do not hold. To discover such bugs, it is essential to thoroughly test different optimization passes, but previous work fails to do so and mainly focused on exploring code coverage. In this paper, we present the first optimization path guided fuzzing solution for JavaScript JIT compilers, namely OptFuzz, which focuses on exploring optimization path coverage. Specifically, we utilize an *optimization trunk path* metric to approximate the optimization path coverage, and use it as a feedback to guide seed preservation and seed scheduling of the fuzzing process. We have implemented a prototype of OptFuzz and evaluated it on 4 mainstream JavaScript engines. On earlier versions of JavaScript engines, OptFuzz found several times more bugs than baseline solutions. On the latest JavaScript engines, OptFuzz discovered 36 unknown bugs, while baseline solutions found none.

1 Introduction

JavaScript engines are widely used in various applications, including web browsers, PDF readers and React Native [35] applications. To enhance execution efficiency, JIT compilers are introduced to JavaScript engines, which dynamically compile hot code during runtime to generate machine code. However, JavaScript is a dynamically typed language, and the variables' types are determined during runtime and can change as the code executes. This characteristic significantly amplifies the intricacy of JIT compilers, rendering them susceptible areas for vulnerabilities within JavaScript engines.

Fuzz testing currently stands as a primary methodology for uncovering vulnerabilities in software [31] [21] [29] [39]. For interpreters like JavaScript engines, there are a series of tailored fuzzing solutions. Some approaches focus on generating test cases satisfying JavaScript syntax checks, which involve code generation based on syntax rules [36], mutation using abstract syntax trees [20] [42], or a combination of both [41] [2]. Some other approaches focus on enhancing the validity and diversity of test case semantics, including [15] [32] [23] [8] [16].

In recent research, attention has been directed towards the JIT compiler, i.e., a core component of JavaScript engines. Some approaches focus on triggering more JIT compilation operations. Fuzzilli [14] introduces customized generation templates and mutation rules to trigger JIT, while FuzzJIT [43] enhances this approach by embedding test cases into a loop, further increasing the likelihood of triggering JIT compilation. Some other approaches focus on discovering JIT-specific vulnerabilities. JIT-picking [3] and FuzzJIT incorporate differential testing between interpreted executions and JIT executions to detect non-crash inconsistency bugs.

However, few of the these approaches could thoroughly explore the program paths of JIT compilers, leaving potential vulnerabilities uncovered. Generally, a JIT compiler has a series of optimization passes, performing different kinds of optimizations on the JavaScript code. As shown in Figure 1, each optimization pass comprises of an outer loop that iterates over instructions, basic blocks or loops of target JavaScript code and a loop body that performs specific code transformation. The loop body follows different program paths (denoted as *optimization paths*) to take different actions. In each optimization path, it will check the pattern of the target instruction/basic block/loop and hold some assumptions, and then apply specific optimizations to the code. The optimization may fail to insert appropriate bailouts or eliminate bailouts incorrectly, which will cause potential vulnerabilities. Thus, it is crucial to thoroughly explore all the optimization paths.

Note that, existing fuzzers often utilize edge coverage to explore target code, which could help explore some optimization paths but are unable to thoroughly explore all paths as discussed in [11,40]. Instead, after an in-depth analysis of JIT compilers, we have made the following observations, which

*Chenggang Wu is the corresponding author. wucg@ict.ac.cn.

could help fuzzers better explore optimization paths.

Observation 1: Entering the optimization pass is not equivalent to triggering the optimization (or the bugs in it). The target optimization bugs could be triggered only if specific conditions are met and certain optimization paths are executed. So, only exploring high-level optimization passes is not sufficient to uncover deep JIT bugs. Instead, *the fuzzer should cover as many optimization paths as possible*.

Observation 2: Edge coverage-guided fuzzing is not efficient at thoroughly exploring optimization paths. Note that, a program/optimization path consists of multiple connected edges. It is possible that, even if a fuzzer has covered all the connected edges of a path, the path itself is still not covered. In this case, the edge coverage guided fuzzer will skip any following test cases that only cover these connected edges (since they are not new), including test cases that cover the target path, which would miss the optimization path and bugs in it. So, *the fuzzer should consider the optimization path coverage*.

Observation 3: Different optimization paths are tested unevenly. Within an optimization pass, different optimization paths have different preconditions. Optimization paths whose preconditions are difficult to satisfy will be tested less in regular fuzzing settings, and are more likely to have uncovered bugs. So, *we need a new fuzzer which allocates more testing resources to the less-tested optimization paths*.

In this paper, based on these three observations, we propose the first optimization path guided fuzzing solution, named OptFuzz, to efficiently discover bugs in JavaScript JIT compilers. Specifically, we utilize optimization paths as feedback to guide the fuzz testing. Note that, although the optimization path is within the optimization pass' outer loop, it could also have inner loops, which will cause the number of optimization paths exploding. To mitigate this issue, we introduce the concept of *Optimization Trunk Path (OptPath)*, which simplifies the inner loops of optimization paths as if they will only be iterated once, and use OptPath coverage to approximate the optimization path coverage, and adjust the seed preservation and scheduling strategies of fuzzing respectively. Specifically, if a test case triggers a new OptPath, it will be preserved as a seed for further testing. Further, if a seed explores an OptPath that is less tested than other OptPaths, then this seed will be prioritized to schedule during fuzzing, e.g., more computational resources will be allocated to test this seed.

To demonstrate the effectiveness of our approach, we implemented a prototype of OptFuzz and evaluated it with four mainstream JavaScript engines: JavaScriptCore, V8, SpiderMonkey and Hermes. Our evaluation shows that OptFuzz is effective in discovering new bugs and exploring optimization paths. In terms of discovering bugs in the latest JavaScript engines, OptFuzz has newly discovered 36 bugs and 26 of the bugs have been confirmed or fixed with 3 CVEs assigned, while existing baselines found none. In terms of discovering bugs in earlier versions of JavaScript engines, we conducted two groups of repetitive experiments. In the short-term baseline comparison experiments, OptFuzz found 9×, 9×, 1.8× and 4.5× more bugs than Superion, DIE, Fuzzilli and FuzzJIT respectively. In the long-term baseline comparison experiments, OptFuzz found 6.3×, 6.3×, 2.1× and 9.5× more bugs than Superion, DIE, Fuzzilli and FuzzJIT respectively. Besides, OptFuzz can explore more OptPaths than baselines as well. Furthermore, ablation experiments have demonstrated the effectiveness of individual designs in Optfuzz. The contributions of this paper are outlined as follows:

- We present an in-depth analysis of the JavaScript JIT compilers and highlight three observations that can help fuzzers better test JIT compilers.
- We present the first optimization path guided fuzzing solution OptFuzz, which utilizes the optimization trunk path coverage as feedback to fine tune the seed preservation and seed scheduling strategy of the fuzzing process.
- We propose a strategy for extracting optimization trunk paths from JavaScript engines. The extracted optimization trunk paths can approximate the optimization paths while mitigating the issue of path explosion.
- We have conducted thorough experiments and reported 36 new bugs in the latest JavaScript engines: 17 in JavaScriptCore, 5 in SpiderMonkey, 1 in V8 and 13 in Hermes. 26 of these bugs have been confirmed or fixed.
- The prototype system is made publicly accessible at https://github.com/JimWongM/OptFuzz.

2 Background and Motivation

2.1 Optimizations in JavaScript Engines

During the execution of the JavaScript program in a JavaScript engine, it goes through stages of parsing into an abstract syntax tree, generating bytecode, and interpretation. The interpreter collects runtime information, such as type details and function calls. When a code snippet or function is executed multiple times and reaches a certain threshold, the JavaScript engine invokes the JIT compiler for optimization, generating machine code to accelerate execution speed.

The JIT compiler can apply various optimizations on the optimized JavaScript code, such as common subexpression elimination, constant folding, dead code elimination, loop-invariant code motion, strength reduction and so on. Compiler optimization is generally implemented using a sequence of optimizing transformations which take a program and transform it to produce a semantically equivalent output program that uses fewer resources or executes faster [46]. As the generated machine code is based on type inference, the compiler inserts guard codes, such as type checks or bound checks, to *bailout* to the interpreter when the inference fails during a

Table 1: Vulnerabilities in different optimizations.

Category	Vulnerability	Description
Instruction	CVE-2019-5857	Error in comparison of -0 and null.
	CVE-2021-30598	Invalid right shift operation optimization.
	CVE-2021-30599	Wrong optimization of bitfield checks.
	CVE-2019-1366	Error in handle opcode Decr_A and Sub_A.
Loop	CVE-2019-8518	wrong hoisting GetByVal leading to OOB.
	CVE-2019-8623	LICM leaves stack variable uninitialized.
	CVE-2019-8671	LICM leaves object property access unguarded.
	CVE-2020-0828	Type confusion when hoisting variable fails.
Function	CVE-2018-4233	Type confusion caused by abstract interpreter.
	CVE-2020-9802	Integer range optimization error caused by CSE.
	Bug240720	The result of integer range analysis is incorrect.
	CVE-2019-9810	Incorrect alias information leading to OOB.
	CVE-2019-17026	Incorrect alias information leading to OOB.
	CVE-2019-26950	UAF caused by wrong side-effect analysis.
	CVE-2021-21230	Incorrect range information.

specific execution. To improve the efficiency of the generated machine code, the compiler performs analysis on the code and eliminates the guard codes when it is confident that a certain inference will always hold. Some JavaScript engines also employ multiple levels of JIT optimizations, such as *BaselineJIT*, *DFG* and *FTL* [34] in JavaScriptCore and *O0*, *O1* and *O2* in Hermes. Higher optimization levels apply more aggressive optimizations, resulting in more efficient generated code.

2.2 Optimization-related Vulnerabilities

A Just-In-Time (JIT) compiler encompasses various optimization passes. The current landscape of compiler optimizations is characterized by a multitude of techniques, each with its own distinct focus. In this paper, we classify compiler optimizations into three categories based on the granularity of effect: instruction, loop and function as shown in Table 1.

Instruction Optimization. The essence of instruction optimization lies in the identification and replacement of specific instruction sequences to enhance the execution speed. This optimization technique operates within a narrow scope, typically targeting a small subset of instructions. Instruction optimization can introduce potential bugs. Inaccurate instruction replacement can result in incorrect results of variable values, leading to significant security vulnerabilities, such as CVE-2019-5857, CVE-2021-30598 and CVE-2021-30599.

Loop Optimization. Loop invariant code motion (LICM) is a prominent loop optimization technique, which aims to hoist loop-invariant code outside the loop body, thereby minimizing the number of times instructions are executed. It is crucial to note that when hoisting the code, the corresponding checks associated with the hoisted code must be relocated. Failure to do so may result in security vulnerabilities, such as CVE-2019-8518, CVE-2019-8623 and CVE-2020-0828.

Function Optimization. Function optimization encompasses two key aspects: 1) optimizing based on the data flow information, and 2) optimizing the control flow structure. Common subexpression elimination (CSE) is a typical optimization based on data flow analysis. CSE replaces later expressions with the result of earlier ones. It is necessary to ensure that the two expressions are equivalent and can be replaced with each other. Otherwise, incorrect replacements can lead to security vulnerabilities, such as CVE-2020-9802. Moreover, CVE-2017-7117 is a vulnerability related to control flow structure optimization.

```
1  void convertToJump( BasicBlock* BB, BasicBlock* targetBB ) {
2      if ( CanMergeBlocks( BB, targetBB ) )
3          mergeBlocks( BB, targetBB ); //optimization
4      else
5          BB->replaceTerminal( Jump ); //optimization
6  }
7  void run( ) {
8      for ( BasicBlock* BB : graph ) {
9          if ( !BB )
10             continue;
11         switch ( BB>terminal( ) ) {
12             …
13             case Branch:
14                 …
15                 if ( BB->successor( 0 ) == BB->successor( 1 ) ) {
16                     convertToJump( BB, BB->successor( 0 ) );
17                     break;
18                 }
19             case Switch:
20                 for ( int i = 0; i < cases.size( ); i++ ) {
21                     // Analyze the target of cases
22                 }
23                 if ( cases.isEmpty( ) ) {
24                     convertToJump( BB, BB->fallThrough( ) );
25                     break;
26                 }
27                 …
28             default:
29                 break;
30         }
31     }
32 }
```

Figure 1: CFGSimplification Optimization.

2.3 Motivation

The aforementioned vulnerabilities shares a common cause: the optimization is triggered and there is an error in the optimization. Edge coverage is the main feedback of current fuzzers. The purpose of edge coverage-guided fuzzing is to improve edge coverage and explore more code, with the expectation of discovering vulnerabilities in the newly explored code. Through the analysis of the JIT compiler, we have found that merely exploring new code is insufficient for uncovering bugs in the JIT optimization passes.

3 Study of JIT Optimization

3.1 Optimization Path

In order to uncover vulnerabilities in the JIT compiler, we conducted an in-depth analysis of each optimization pass. We found that each optimization pass contains a large outer loop. This outer loop traverses the intermediate representation (IR) of the optimized JavaScript code at a specific granularity level (e.g., loops, basic blocks, and instructions). Within each iteration of the outer loop, the loop body code analyzes and pro-

cesses the IR code. Figure 1 depicts a schematic diagram of the CFGSimplification optimization in JavaScriptCore. Line 8 contains an outer loop that traverses each basic block of the optimized JavaScript code and triggers the optimization if a specific condition is met. The optimization being triggered actually means that the optimization pass executes to the location where a code transformation occurs (line 3 and line 5). In other words, the optimization pass executes a series of code blocks within the outer loop in a particular iteration, performing a code transformation on the optimized JavaScript code. Therefore, this paper introduces the concept of *Optimization Path*, which refers to a code execution path within the outer loop of an optimization pass, leading to the location where the code transformation occurs. The functionality of the optimization pass in the JIT compiler is manifested in these optimization paths within the outer loop, and different optimization paths can produce varying or identical results on the optimized JavaScript code.

The outer loop in line 8 generates paths of different lengths for different numbers of iterations, but the length of the optimization path is not affected by the iterations of this outer loop. In fact, if only the number of iterations of the outer loop is increased, but no new optimization path is generated within the loop, it is actually testing the same functionality in the optimization pass. When conducting fuzz testing, we should pay more attention to different optimization paths within the outer loop to test different functionalities.

3.2 Enter Optimization≠Trigger Optimization

During JIT compilation, the JIT compiler performs a variety of optimizations on the intermediate representation of JavaScript programs. Each optimization pass is executed. However, not every executed optimization pass will actually result in code transformation. In other words, the optimization paths in an optimization pass are not necessarily executed. Multiple prerequisites must be met for an optimization path to be executed, as engine designers often design optimizations specifically for certain patterns in the code.

For instance, consider the loop invariant code motion (LICM) optimization in JavaScriptCore. To trigger the code motion, the code must satisfy the following four prerequisites: 1) The presence of a loop preheader; 2) No write operation exists in the hoisted code; 3) The read region of the hoisted code cannot be modified by other code within the loop, and 4) The code itself is eligible for hoisting. If one of the prerequisites is not satisfied, the optimization cannot be triggered. Therefore, entering the optimization pass is not equivalent to triggering the optimization. Failure to trigger an optimization means that the true functionality of that optimization pass remains untested, thereby potentially overlooking any bugs present in that optimization pass.

We conducted fuzzing on JavaScriptCore using DIE [32] and Fuzzilli [14], two edge coverage-guided fuzzers. The experiment lasted for 72 hours. A total of 1370090 and 17406775 valid test cases were generated by DIE and Fuzzilli respectively. Out of these, 721074 and 8266477 test cases entered the LICM optimization pass. However, only 263908 and 3660644 test cases triggered LICM optimization, accounting for 36.6% and 44.3% of the test cases that entered the optimization pass. The primary cause of vulnerabilities in the JIT compiler is the triggering of incorrect optimizations. The current fuzzers inadequately test the code that triggers the optimization. Consequently, these approaches may overlook the vulnerabilities in the JIT optimization passes.

3.3 Overlooked Optimization Path

When designing an optimization pass, the engine designer will design different optimization techniques for different bytecodes, operand types, built-in functions, control flow structures, and data flow relationships. The optimization pass in Figure 1 incorporates specific optimization techniques to handle different control flow nodes (such as branch and switch). The code transformation occurs at line 3 and line 5. There are four distinct optimization paths in the schematic diagram. The code in line 20 has a loop. Different iterations of the loop will produce different optimization paths, we consider these optimization paths to be an optimization path in this section.

Different optimization paths within the same pass may reuse code from each other. The execution of some optimization paths (e.g. line 11→15→16→2→3, line 11→15→16→2→4→5 and line 11→23→24→2→3) will potentially cover all edges of other optimization paths (e.g. line 11→23→24→2→4→5). Consequently, in edge coverage-guided fuzz testing, even if a specific test case triggers a new optimization path, it may not be preserved as a seed due to its failure to reach new edges.

In order to explore the seed preservation for current fuzz testing, we manually identify the start and end of the optimization paths and mark them in the source code. We conducted fuzzing on JavaScriptCore using Fuzzilli and pay close attention to the execution of optimization paths for each test case. If a test case executes a new optimization path but is not preserved as a seed, then we consider that this optimization path has been **missed**. Over the fuzzing of 72 hours, a total of 10880237 valid test cases have been generated. These test cases covered 1520 optimization paths, but missed 180 optimization paths, thereby limiting the opportunity for thorough testing on those specific optimization paths and potentially overlooking underlying vulnerabilities.

3.4 Imbalanced Testing for Optimization Path

There are various optimization paths in an optimization pass, some of which have easily satisfied conditions, while others have more challenging conditions. Taking the optimization in Figure 1 as an illustration, in order to execute to line 13, the

JavaScript program must encompass syntactical structures, such as *if* statements, *for* loops,*while* loops, or other constructs capable of generating the intermediate representation *Branch*. Similarly, to execute to line 19, the JavaScript input program must contain a *switch case* syntax structure. To invoke *convertToJump* at line 16, it is imperative for the two successor blocks of the IR *Branch* to be identical, thereby facilitating the transformation of Branch into Jump. Furthermore, to invoke *convertToJump* at line 24, the cases of the switch must be empty. These optimization paths have different preconditions, leading to different extents of being tested.

Edge coverage-guided fuzzing still enables partial testing of optimization paths. However, experimental results reveal an imbalance in the testing of these paths using Fuzzilli. During 72-hour fuzzing, the number of tests conducted for the four optimization paths in Figure 1 is 11, 1, 83, and 0 respectively. It is observed that optimization paths with easily satisfied conditions undergo more testing, while those with challenging conditions are tested less. From a vulnerability discovery perspective, fuzz testing should prioritize the testing of the less-tested optimization paths.

4 Design

The goal of this paper is to test the optimization passes in the JIT compilers. The analysis above indicates that when performing fuzz testing on the JIT optimization passes, it is crucial to not only test up to the optimization pass, but also to comprehensively test the optimization paths within that pass.

4.1 Optimization Trunk Path

To further explore the optimization paths, we can employ them as feedback to guide the seed preservation and scheduling process of fuzz testing. Nonetheless, nested inner loops persist within the optimization paths, and different iterations of these nested loops will result in different optimization paths. Thus, utilizing the optimization paths as feedback for fuzzing still poses the challenge of explosion in the number of seeds.

We conducted a statistical analysis of 29 optimization passes in JavaScriptCore, revealing that their inner loops exhibit a branch count ranging from approximately 0 to 6, with 86% of them having no more than 2 branches. The logic of these inner loops is simple, often traversing information such as operands, metadata, and use points of the intermediate representation. In fuzz testing, even without specifically targeting the testing of branches within the inner loop, these branches will be adequately tested. Our experiments show that 64.4% of the branches are tested more than 1,000 times, and 88.5% of the branches are tested more than 100 times during the 72-hour fuzz testing. Longer tests lead to more testing of these branches. While multiple iterations do not significantly aid in exploring new combinations of code blocks within the inner loop, they do create paths of varying lengths and exacerbate the path explosion issue. Consequently, this paper excludes nested loops in the optimization paths.

Based on the above analysis, this paper introduces the concept of *Optimization Trunk Path (OptPath)* and utilizes OptPath to approximate the optimization path. *Trunk* means the main woody stem of a tree as distinct from its branches and roots. The OptPath in this paper represents a path within the outer loop, which ignores the nested inner loop. The starting point of OptPath is the entry point of one iteration of the outer loop, and the end point is the tail code of the same iteration. For example, the sequence of line 9→11→19→23→24→2→3→30→31 in Figure 1 is an OptPath where we ignore the inner loop of line 20. Notably, the sequence of line 9→10→31 is also an OptPath. Unlike the optimization path, this particular OptPath does not trigger the optimization; instead, it proceeds to the branch that results in an early exit. We will discuss this in subsection 4.4.

4.2 Seed Preservation

During fuzz testing, OptFuzz records the OptPaths that the test case executes, and if a test case triggers a new OptPath, it is preserved as a seed. When the JavaScript engine executes to the start of an OptPath, it starts recording the path. When execution reaches the end point of an OptPath, it stops recording the path and performs a hash operation on each basic block of the recorded path to obtain a hash value. Different OptPaths will result in different sequences of basic blocks, which will generate different hash values without accounting for hash collisions. After the execution of a test case, OptFuzz can determine whether new OptPaths have been reached by examining the presence of new hash values.

Numerous research studies [4,11,14,32,40,42] have demonstrated the significance of incorporating edge coverage feedback into fuzz testing to explore new code in the target program. In this paper, we propose an approach that combines edge coverage and OptPaths to comprehensively explore new code through edge coverage, thereby traversing different optimization passes. Additionally, we leverage OptPaths feedback to test optimization paths within the optimization pass. We maintain separate seed corpus for edge coverage and OptPaths. Initially, the seeds are mostly preserved by edge coverage, and in order to explore more code quickly, the seeds preserved by edge coverage are mainly used for mutation at this time. As the fuzz test progresses, the proportion of seeds preserved by the OptPaths gradually increases. Once the number of seeds preserved by the OptPaths surpasses a certain threshold, the seeds preserved by the OptPaths are enabled. Subsequently, the fuzz testing alternates between seeds preserved by edge coverage and seeds preserved by OptPaths.

The interesting test cases may include code that is not related to the new edge or the new OptPath. To minimize the impact of such code, we employ test case trimming techniques. This involves reducing the test cases while ensuring they

still reach the new edge or the new OptPath. If a test case reaches both the new edge and the new OptPaths in different optimization passes, we generate n distinct seeds based on different trimming criteria. Specifically, we trim out a seed based on the new edge, and trim out different seeds for the new OptPaths in different optimization passes. These seeds are then placed into separate seed queues. In this way, we maintain a seed queue for edge coverage and also maintain respective seed queues for different optimization passes.

4.3 Seed Scheduling

In the process of fuzz testing, some OptPaths whose preconditions are easy to satisfy are given more opportunities to be tested, while OptPaths whose preconditions are difficult to satisfy are given much less opportunities to be tested. Seed scheduling based on OptPaths aims to make those OptPaths which are tested less get more chance to be tested. To identify the OptPaths that require more testing, we keep track of the cumulative number of times each path is tested.

We designed a two-tier seed queue for seed scheduling. When a test case executes a new OptPath, the test case is preserved as a seed, which is trimmed and put into the primary queue. We maintain a primary queue for each optimization pass. The secondary queue is utilized to store the prioritized seeds. This is achieved by selecting a number of seeds from each primary queue in ascending order of the number of times the OptPath has been tested. These selected seeds are then stored in the secondary queue. Subsequently, seeds are extracted from the secondary queue one by one for mutation. Once all the seeds from the secondary queue have been mutated, seeds are again selected from each primary queue.

4.4 Discussion

Early Exit Branches in OptPaths. In this paper, we employ OptPaths as an approximation of the optimization paths. OptPaths and optimization paths are not entirely equivalent. There are some exit branches before an optimization is triggered. The OptPath also records these exit branches. The analysis in subsection 3.2 emphasizes that the triggering of an optimization requires the fulfillment of various conditions. It is not an easy task to execute an optimization path, but easier to encounter an early exit branch. Furthermore, early exit branches are more likely to be tested during fuzz testing. In this paper, we propose a seed scheduling strategy based on OptPaths. This strategy favors scheduling OptPaths that have been tested less frequently, while exit branches that have been tested more frequently are not selected for mutation. As the testing progresses, these OptPaths containing exit branches do not excessively consume computational resources. On the other hand, individual vulnerabilities, such as CVE-2019-8623 [37], also require execution of the exit branch before incorrect optimizations are triggered. Hence, exit branches do not contradict our objective of testing the optimization path. In fact, the execution of early exit branches can even aid in finding potential vulnerabilities in specific scenarios.

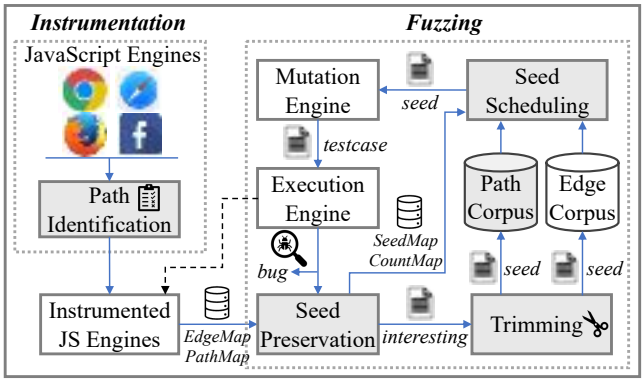

Figure 2: Workflow of our prototype system.

Ignored Inner Loops. The OptPath ignores the nested inner loops. Generally, the number of iterations within inner loops exceeds that of outer loops by 1-2 orders of magnitude, and the control flow structure in inner loops is relatively simple with fewer branches. Consequently, when outer loops undergo frequent testing, a majority of paths within inner loops also receive more comprehensive testing. There may be a few under-tested paths in the inner loops that require a more fine-grained feedback for enhancement, but implementing such feedback could introduce additional overhead that we will investigate in future work.

The Order of Edges in OptPaths. To record the path, AFLFast [4] performs hashing on the set of visited edges without considering the order of the edges. For OptPath, in tandem with its definition, different sets of visited edges are likely to represent different edge orders, so hashing the set of visited edges may be adequate. However, considering the order when recording the paths tends to yield more accurate results. This is because two test cases with the same set of visited edges may execute edges in different orders. Hashing the set of visited edges will miss some interesting test cases. Therefore, OptPath takes into account the order of edges and records the entire path.

5 Implementation

5.1 Workflow

Figure 2 shows the workflow of our fuzzing system. The system consists of two parts: instrumentation and fuzzing. The instrumentation module identifies the OptPaths and instruments the JIT compiler. The fuzzing module is built on Fuzzilli [14]. We have enhanced Fuzzilli's evaluation module to incorporate a dedicated evaluation feature for OptPaths. Additionally, we have integrated a trimming strategy specifically targeting OptPaths into Fuzzilli's trimming module. To implement seed scheduling based on OptPaths, we have intro-

duced a two-tier seed queue within Fuzzilli's Corpus module. We reuse Fuzzilli's mutation module and execution module. Table 2 presents the lines of code (LoC) of each component.

Table 2: The lines of code of each component.

Module	Component	LoC	Language
Instrumentation	Path Identification	1281	C++
	Instrumented Code	209	C++
Fuzzing	Seed Preservation	155	Swift, C
	Seed Scheduling	342	Swift
	Trimming	419	Swift

5.2 Instrumentation Module

This module inserts code into the JavaScript engine to collect information on edge coverage and OptPaths. We have established two bitmaps, namely *EdgeMap* and *PathMap*, to record the edge coverage and OptPaths information during the execution of test cases. The insertion process consists of two steps. The first step is to identify the various types of basic blocks in the OptPath. We implement an LLVM [25] Pass named *OptPathAnalysis*. The *LoopInfoWrapperPass* is able to analyze all the loop information in the target program, from which *OptPathAnalysis* obtains the structure of each loop as well as the loop nesting relationships in the optimization passes. We mark the header [26] of an outer loop as the starting point of an OptPath, the latch basic block [26] of a outer loop as the end point of an OptPath, and the basic blocks between the header and latch as the OptPath basic blocks. We exclude the basic blocks inside the nested loops.

In the second step, we instrument the JavaScript engine by leveraging Clang Sanitizer Coverage [6] to insert calls to the function *trace_pc_guard(uint32* guard, int flag)* before each basic block. The first argument of this function, *guard*, receives the unique ID of the basic block. The second parameter *flag* indicates the attribute information of the basic block analyzed by *OptPathAnalysis*. The Pseudo code of this function is shown as Algorithm 1.

start is a global variable that is initially false, indicating that the start of an OptPath has not yet been encountered. *path* is used to record the OptPath. After entering the function *trace_pc_guard*, the initial action involves indexing the *Edgemap* bitmap according to the basic block ID and subsequently setting the corresponding unit. Next, according to the attribute *flag* of the basic block, the OptPath information is collected. If *flag* of the current basic block is 1, it means that the current basic block is the start point of an OptPath, so *start* is set to true, and the ID of the currently executed basic block is recorded into *path*. If *flag* is 0 and *start* is true, the basic block is added to *path*. If *flag* is 2, it means that the current basic block is the end point of an OptPath, then the ID of the current basic block is added to *path*, and a hash value is calculated for all the basic blocks in *path*. Then, based on the hash value, the bitmap *PathMap* is updated. After that, *start* is set to false and *path* is cleared.

Algorithm 1 trace_pc_guard

Input: guard: ID of the basic block
Input: flag: attribute information of the basic block

```
1:  UpdateEdgeMap( guard )
2:  if flag == 0 and start then
3:      path ← RecordPath( guard )
4:  end if
5:  if flag == 1 then
6:      start ← true
7:      path ← RecordPath( guard )
8:  end if
9:  if flag == 2 then
10:     path ← RecordPath( guard )
11:     hash ← ComputeHash( path )
12:     UpdatePathMap( hash )
13:     start ← false
14: end if
```

5.3 Fuzzing Module

The fuzzing module includes sub-modules for mutation, execution, seed preservation, trimming, and seed scheduling. The mutation submodule adopts mutation strategies from Fuzzilli. The execution module initiates the instrumented JavaScript engine and provides it with test cases. Throughout the execution process, the JavaScript engine dynamically updates the *Edgemap* and *PathMap*. The seed preservation submodule updates *CountMap* based on *PathMap* to accumulate the test number for each OptPath, and determines whether the test case is interesting based on *Edgemap* and *PathMap*. If a test case executes to new edges or unexplored OptPaths, the test case is *interesting*, and the relationship between the new OptPath and the test case is recorded into *SeedMap*. The trimming submodule removes redundant statements from interesting test cases. After trimming, the test cases that still trigger new edges are added to *Edge Corpus*, while those that lead to new OptPaths are added to *Path Corpus*. In *Path Corpus*, we maintain a seed queue for each optimization pass. In seed scheduling submodule, seeds in *Path Corpus* are selected based on *CountMap* and *SeedMap* to prioritize the testing of OptPaths that are currently tested less often while seeds in *Edge Corpus* are randomly selected.

Previous work such as FuzzJIT and JIT-picker detects non-crash bugs in JavaScript engines through differential testing. In this paper, we also develop a similar sanitizer named DiTing. DiTing captures bugs by comparing the results of variables executed in the interpreter and each optimization level.

6 Evaluation

To validate the effectiveness of our approach, this section evaluates OptFuzz and answers the following four questions:

Q1. Can OptFuzz find bugs in the JavaScript JIT compil-

ers? How does OptFuzz perform in terms of bug findings comparing to the state-of-the-art fuzzers? (subsection 6.2)

Q2. What is the distribution of OptPaths across optimizations? Can our approach explore more OptPaths against state-of-the-art fuzzers? (subsection 6.3)

Q3. What is the impact of individual designs in OptFuzz on bug finding? Do these designs make sense? (subsection 6.4)

Q4. How does OptFuzz perform in terms of code coverage against the state-of-the-art fuzzers? (subsection 6.5)

6.1 Experiment Setup

Baseline: We compare OptFuzz with four state-of-the-art fuzzers including Superion [42], DIE [32], Fuzzilli [14], and FuzzJIT [43]. JIT-Picker [3] and SoFi [16] are not compared for the following reasons. JIT-Picker contribute to the detection of non-crash bugs in the engines using differential testing, which is orthogonal to our work. SoFi focuses on improving the validity and diversity of test cases and is not open source.

Initial Seed Corpus: Fuzzilli does not require an initial seed corpus at startup. Fuzzilli first enters the generation mode at runtime, and uses the developer-defined JavaScript code generators to generate test cases. If a test case executes to new edges, it will be preserved as a seed. If Fuzzilli generates 100 consecutive test cases that cannot execute to new edges, it switches to the mutation mode. OptFuzz and FuzzJIT are built upon Fuzzilli and hence do not necessitate an initial seed corpus. In contrast, DIE and Superion do require an initial seed corpus. We utilize the seeds generated by Fuzzilli in the generation mode as the initial corpus for Superion and DIE.

Test Target: We choose four JavaScript engines, namely V8 [13] in Google Chrome, JavaScriptCore [1] in Apple Safari, SpiderMonkey [28] in Mozilla Firefox, and Hermes [10], developed by Facebook. These engines are the current mainstream JavaScript engines and have been tested in depth by security researchers and OSS-Fuzz [12]. It is worth noting that Hermes is a lightweight JavaScript engine that does not have a JIT compiler, but there are numerous optimization passes in Hermes which can be tested by OptFuzz.

Environment: We performed our experiments on Intel Xeon Gold 5218 2.30GHz (32 cores) machines with 132GB RAM. The OS version is Ubuntu 20.04.3. Each campaign is assigned 10 cores.

Parameter Settings: In the early stages of fuzzing, we use seeds preserved by edge coverage to explore more code. We limit the use of seeds preserved by OptPaths only after the number of seeds in the *Path Corpus* reaches a certain threshold. After multiple experiments in subsection 6.4, we set this threshold to 300 for JavaScriptCore and V8, 200 for SpiderMonkey, and 20 for Hermes.

6.2 Bug Finding Ability

To answer Q1, we utilized OptFuzz to test four latest versions of JavaScript engines for six months and record the number of bugs found by OptFuzz. The results are presented in subsubsection 6.2.1. To demonstrate OptFuzz's superiority in bug finding, we compared the number of bugs found in earlier versions of JavaScript engines with baselines in subsubsection 6.2.2. The comparison experiments are divided into short-term (72 hours) and long-term (45 days) experiments.

6.2.1 Results of Real-word Bugs

To explore the bug finding ability of OptFuzz, we deployed OptFuzz on four servers and tested four engines for six months. Eventually, we discovered 36 new bugs, 26 of which were confirmed by the vendor with 4 security-sensitive bugs including 3 CVEs assigned. In V8, OptFuzz found 1 new bug and 3 known bugs. In JavaScriptCore, OptFuzz found 17 new bugs (including 2 security-sensitive bugs) and 1 known bug. In SpiderMonkey, OptFuzz found 5 new bugs. In Hermes, OptFuzz found 13 new bugs, including 2 security-sensitive bugs. The detailed descriptions of the bugs are shown in Table 19.

OptFuzz mainly found vulnerabilities due to JIT optimization errors, including integer range overflow (e.g. CVE-2023-38595), OOB (e.g. Issue 256022), UAF (e.g. CVE-2023-38542), and uninitialized memory (e.g. CVE-2023-46140). With differential testing system *DiTing*, OptFuzz was able to find some semantic bugs, e.g. Issue 257949, Issue 263647, and Issue 1200. Out of the 41 identified bugs, 31 are JIT-related, while some have root causes unrelated to JIT, such as Issue 256507 and Issue 263954. Despite extensive long-term and numerous short-term fuzz tests, Fuzzilli did not identify the aforementioned bugs. Upon analysis, we find that the triggering test cases for these bugs are generated through the mutation of seeds in *Path Corpus*. Only the test cases that trigger the new OptPaths are included in *Path Corpus*. In other words, OptFuzz preserved the parent seeds of Issue 256507 and Issue 263954 by OptPaths. Then, OptFuzz mutates the parent seeds to produce test cases that trigger Issue 256507 and Issue 263954.

6.2.2 Comparison with Baselines

To demonstrate OptFuzz's superiority in bug finding, we compared the number of bugs found with baselines in earlier versions of JavaScript engines. We conducted a two week-long fuzz testing on the latest version of JavaScript engines using Superion, DIE, Fuzzilli and FuzzJIT with *DiTing* system. However, they did not uncover any bugs. These fuzzers has previously identified a significant number of bugs and security vulnerabilities over the past few years. To ensure a fair comparison with baselines, we opt to use earlier versions of the engines for our comparison experiments. The versions of each engine are detailed in Table 3.

For JavaScriptCore, V8 and Hermes, we connect our own differential testing system *DiTing* to Superion, DIE, Fuzzilli, and OptFuzz, and for SpiderMonkey, we use SpiderMonkey's own differential testing system. FuzzJIT has its own differential testing system which does not support testing of Hermes.

Table 3: Versions for different engines.

JavaScript Engine	Commit ID	Time	Code Lines
JavaScriptCore	7e4859	2020.1.31	629591
SpiderMonkey	1b9955	2022.1.10	1511250
V8	ed0a85	2021.11.5	1245655
Hermes	2d8c88	2023.8.30	303928

Table 4: Number of bugs found by each fuzzer in 72 hours. Data in the parentheses indicate Mann-Whitney U Test results *p-value*. A value less than 0.05 indicates the result is statistically different between OptFuzz and the selected fuzzer.

	Superion	DIE	Fuzzilli	FuzzJIT	OptFuzz
JavaScriptCore	0 (**0.002**)	1 (**0.029**)	2 (**0.016**)	1 (**0.007**)	4
SpiderMonkey	0 (1.000)	0 (1.000)	0 (1.000)	0 (1.000)	0
V8	1 (0.067)	1 (1.000)	1 (1.000)	1 (1.000)	1
Hermes	0 (**0.001**)	0 (**0.001**)	2 (**0.002**)	–	4
Total Num	1	2	5	2	9

Short-term Experiments. In the short-term experiment, each fuzz testing trial lasts 72 hours and is repeated five times. The results of the short-term experiments are depicted in Table 4. OptFuzz demonstrates a clear advantage over JavaScriptCore and Hermes. In particular, Optfuzz found 2× more than Fuzzilli on JavaScriptCore and Hermes. The performance of each fuzzer does not vary on SpiderMonkey and V8. This is probably because 72 hours of testing time is not be sufficient for SpiderMonkey and V8.

Long-term Experiments. The long-term experiment lasts for 45 days. The results are illustrated in Table 5, showing that OptFuzz is capable of discovering more bugs than other baselines. In V8, both Fuzzilli and OptFuzz discover two bugs. They identify one common bug, while each find a distinct second bug. The bug uncovered by Fuzzilli pertain to the *ArrayBuffer maxByteLength* and is unrelated to the JIT compiler, whereas the bug uncovered by OptFuzz is associated with the JIT compiler's verifier [22]. In long-term experiments, OptFuzz found 6.3×, 6.3×, 2.1× and 9.5× more bugs than Superion, DIE, Fuzzilli and FuzzJIT respectively. Most of the bugs found in earlier versions of the engines have been fixed in the latest version. Therefore, we do not report these bugs to the engine vendors and omit them from Table 19. However, OptFuzz still found a bug in the earlier version that still exists in the new version, which is Issue 265978.

Table 5: Number of bugs for each fuzzer in 45 days.

	Superion	DIE	Fuzzilli	FuzzJIT	OptFuzz
JavaScriptCore	0	2	2	1	6
SpiderMonkey	0	0	1	0	4
V8	1	1	2	1	2
Hermes	2	0	4	–	7
Total Num	3	3	9	2	19

Table 6: The number of OptPaths explored by OptFuzz for optimization passes in JavaScriptCore within 24 hours.

Optimization Pass	NP	STD	ANT	MNT
ConstantFolding	173	19.9	7740	83
ObjectAllocationSink	118	3.2	468	96
IntegerRangeAnalysis	32	3.3	2717	227
TypeCheckHoist	20	0.7	26170	884
CFGSimplification	16	0.4	11896	501
StrengthReduction	136	12.4	7106	129
ArgumentsElimination	64	3.6	333	89
IntegerCheckComb	14	0.8	4892	255
ValueRepReduction	2	1.0	18	131
VarargsForwarding	25	2.8	1065	158
CSE	52	1.7	4126	105
OSRAvailabilityAnalysis	46	5.9	12713	475
B3ReduceDoubleToFloat	56	17.9	176	74

NP indicates the number of OptPaths in an optimization.
STD denotes the standard deviation of NP.
ANT indicates the average number of times each OptPath is tested.
MNT indicates the median number of times each OptPath is tested.

6.3 Exploration of OptPaths

6.3.1 Number of OptPaths in Different Optimizations

To evaluate OptFuzz's exploration of OptPaths, we ran OptFuzz on JavaScriptCore, V8, SpiderMonkey, and Hermes. During fuzzing, we record the number of OptPaths explored by OptFuzz in different optimization passes. Each trial lasted 24 hours and was repeated 5 times. Table 6 gives the number of OptPaths explored by OptFuzz in JavaScriptCore within 24 hours. Table 17 and Table 18 give the number of OptPaths explored in V8, SpiderMonkey and Hermes respectively. NP indicates the number of OptPaths explored in an optimization pass. The results of five trials are averaged to obtain NP. STD denotes the standard deviation of NP. ANT indicates the average number of times each OptPath is tested. MNT indicates the median number of times each OptPath is tested. For each trial, the median is calculated first. The medians of five trials are averaged to obtain MNT.

The experimental results show that there is no explosion in the number of OptPaths. Furthermore, it is observed that the number of OptPaths varies across various optimization passes. Some optimization passes are designed with different optimization techniques for different instruction opcodes and operands, data flow relationships, and control flow structures, so the number of OptPaths in these optimization passes is high. Take *StrengthReduction* optimization pass as an example. The optimization pass designs different optimization techniques based on different operators (Add, Sub, And, etc.) and operand types (Int32, Int64, String, etc.). While some optimization passes, for example, *ValueRepReduction* only optimizes for specific intermediate representation *ValueRep* and *DoubleRep*, so the number of OptPaths is low.

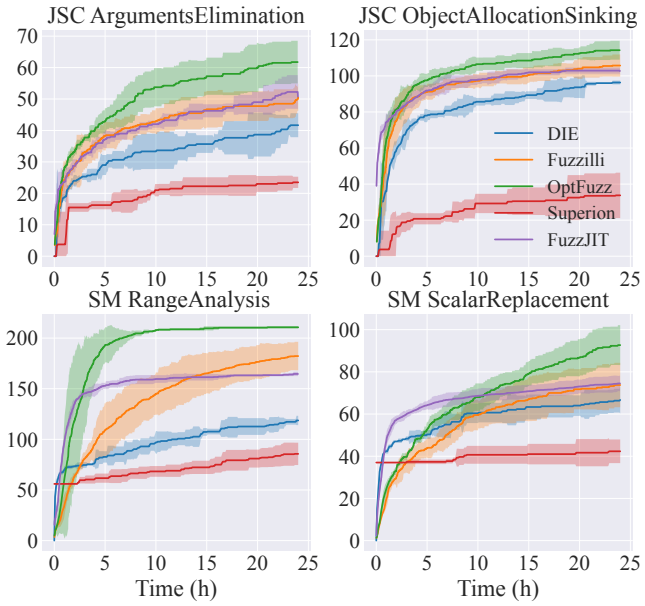

Figure 3: The number of OptPaths explored by Superion, DIE, Fuzzilli, FuzzJIT and OptFuzz in JavaScriptCore (JSC) and SpiderMonkey (SM) within 24 hours.

6.3.2 Comparison with Baselines

To demonstrate the advantages of OptFuzz in exploring OptPaths, we ran OptFuzz, Superion, DIE, Fuzzilli, and FuzzJIT on JavaScriptCore and SpiderMonkey respectively. Each trial lasted 24 hours and was repeated 5 times. To collect the number of OptPaths explored by DIE, Superion, Fuzzilli, and FuzzJIT, we modified these fuzzers to incorporate a new bitmap to dynamically record the number of OptPaths explored. This is analogous to capturing all test cases generated by the fuzzers. Figure 3 gives the number of OptPaths explored by different fuzzers in the *Arguments Elimination* and *Object Allocation Sinking* pass of JavaScriptCore and *Range Analysis* and *Scalar Replacement* pass of SpiderMonkey. The solid lines represent mean and the shades around lines are confidence intervals.

Compared to OptFuzz, Superion, DIE, Fuzzilli and FuzzJIT were able to explore 38.6%, 67.5%, 86.0% and 81.9% of the OptPaths, respectively. OptFuzz can explore more OptPaths than other fuzzers. This is because different OptPaths within the same optimization pass have varying conditions but share significant similarities. Seeds in *Path Corpus* can keep new OptPaths intact. By mutating these seeds, it becomes easier to generate test cases that trigger more new OptPaths in the same optimization pass.

DIE and Superion explore fewer OptPaths compared to Fuzzilli. This is because Fuzzilli is designed with templates and mutation rules specifically aimed at triggering JIT, enabling faster execution into the optimization pass. Superion explores fewer OptPaths than DIE due to its trimming strategy, which retains the code that triggers new edges but prunes the code that triggers new OptPaths. DIE does not perform syntax-valid trimming, allowing the code related to triggering OptPaths to remain. This provides more opportunities to generate test cases that trigger different OptPaths. Fuzzilli and FuzzJIT explore a similar number of OptPaths, which is because FuzzJIT is implemented based on Fuzzilli. In the early stages of fuzz testing, FuzzJIT explores faster, this is because FuzzJIT designs a template where each test case is embedded in a loop, resulting in a higher probability of triggering JIT.

6.4 Ablation Experiment

To answer Q3, this section performs ablation experiments to illustrate the effectiveness of our design. The experiment evaluates OptFuzz on JavaScriptCore in four parts including seed preservation, seed scheduling, trimming strategy and threshold. We will discuss switching strategy in subsection A.3. Table 7 shows the number of bugs discovered by different designs. Each trial lasted 72 hours and was repeated 5 times.

6.4.1 Seed Preservation

To evaluate the seed preservation strategy of OptFuzz, we additionally implemented five seed preservation strategies including *countpath*, *n2*, *n4*, *n8* and *looppath*. AFL [30] not only evaluates whether a test case triggers a new edge but also takes into account the number of times an edge has been executed when assessing whether a test case is interesting. We have incorporated a similar mechanism, named *countpath*, into OptFuzz. Furthermore, we have introduced *n2*,*n4*, and *n8* paths [40], representing paths comprising 2/4/8 consecutive edges, respectively, and utilize these paths for seed preservation instead of OptPaths. *looppath* denotes optimization paths that do not ignore nested loops. Fuzzilli indicates that only edge coverage is used for seed preservation.

The experimental results detailed in Table 7 demonstrate that OptPath-based seed preservation strategy can uncover more bugs. Additionally, Figure 4 depicts the quantity of seeds preserved by the various seed preservation strategies as well as the number of paths explored. It is evident that both *looppath* and *n8* suffer from the path explosion problem. *looppath* kept 10 million new paths in 72 hours. Path explosion makes the number of seeds grow rapidly, causing *looppath* and *n8* to waste most of their testing resources on unproductive seeds, thereby diminishing their bug-finding capabilities. Despite *n4* having a lower number of seeds than *n8*, it suffers from the same problem.

Path explosion does not occur in *n2*. Fuzzilli's edge coverage is approximated using basic block coverage. *n2* is actually a path consisting of two consecutive basic blocks, which is actually an edge. *n2* and Fuzzilli employ a similar seed preservation method, thereby yielding comparable bug discovery capabilities. *n2* has fewer seeds than Fuzzilli because we limit *n2* to optimization passes rather than the whole JavaScript

Table 7: The number of bugs found for different designs.

Seed Preservation			Seed Scheduling			Threshold			Switching Strategy		
Fuzzer	Bug	*p-value*	Fuzzer	Bug	*p-value*	Fuzzer	Bug	*p-value*	Fuzzer	Bug	*p-value*
countpath	1	**0.007**	OptFuzz_RandomSelect	1	**0.032**	OptFuzz_10	0	**0.009**	OptFuzz_RandomSwitch	2	1.000
n2	2	**0.030**	OptFuzz_PowerSchedule	1	**0.032**	OptFuzz_50	1	**0.027**	OptFuzz_AIMD	1	0.256
n4	0	**0.002**	OptFuzz_RarenessSchedule	3	1.000	OptFuzz_100	2	0.134	OptFuzz_NoSwitch	0	0.067
n8	0	**0.002**	Trimming			OptFuzz_200	3	0.106	OptFuzz_Wind	2	0.349
looppath	0	**0.002**	OptFuzz_oneseed	1	**0.027**	OptFuzz_300	4	1.000			
Fuzzilli	2	**0.016**	OptFuzz_twoseed	0	**0.009**	OptFuzz_500	0	**0.009**			
OptFuzz	4	1.000	OptFuzz_nseed	3	1.000	OptFuzz_800	2	**0.027**			

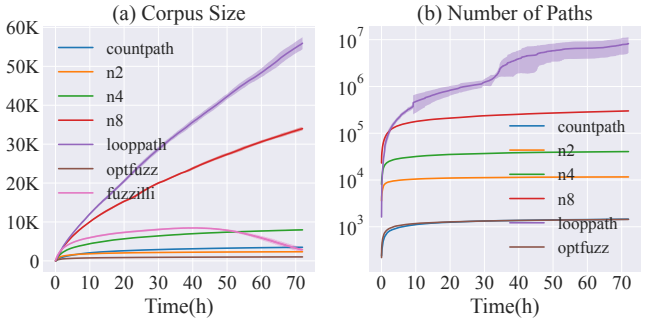

Figure 4: (a)The number of seeds preserved by different seed preservation strategies. (b)The number of paths explored by different seed preservation strategies.

engine. Consequently, if a test case executes to new edges in the code that is not related to optimization (e.g., interpreter), Fuzzilli can preserve it as a seed, but n2 cannot. Moreover, the number of seeds in Fuzzilli first increases and then gradually decreases due to the inclusion of *seed cleanup* in Fuzzilli's corpus. Fuzzilli's corpus performs a seed cleanup every 30 minutes and removes the seeds that have been selected more than 25 times. If the corpus has less than 1000 seeds, no cleanup is performed. The number of seeds for *countpath* is slightly higher than that of OptFuzz due to the consideration of multiple executions of an OptPath in *countpath* as distinct seeds. In fact, *countpath* comes to the similar number of OptPaths as OptFuzz in Figure 4(b). *countpath* does not perform well in bug discovery, probably because *countpath* does not fit well with our scheduling strategy. Our scheduling strategy prioritizes the paths that are currently less tested. If *countpath* itself considers the number of executions per path, the paths chosen by the scheduling strategy may not be the paths that are currently less tested.

In conjunction with the aforementioned analysis, the seed preservation strategy based on OptPaths can avoid the path explosion problem. Among the various strategies, the OptPaths yield the fewest number of seeds but can find the most number of bugs, which demonstrates the high quality of seeds preserved based on the OptPaths.

6.4.2 Seed Scheduling

To evaluate the seed scheduling strategy of OptFuzz, we additionally implemented two scheduling strategies, *RandomSelect* and *PowerSchedule*. *RandomSelect* denotes the random selection of seeds for mutation. *PowerSchedule* denotes the implementation of seed power scheduling based on OptPaths. *PowerSchedule* determines the number of test cases mutated from each seed. If an OptPath is tested less, the corresponding seed can be mutated to produce more test cases.

The results in Table 7 show that our scheduling strategy outperforms *RandomSelect* as well as *PowerSchedule*. In contrast to *RandomSelect*, our scheduling strategy focuses on testing the OptPaths that are currently tested less often and tests them in a more targeted manner. Power schedule is a commonly used method for fuzz testing in seed scheduling. However this method does not perform well in OptFuzz. *PowerSchedule* uses random selection to decide the order in which the seeds are selected, which can avoid the local optimum problem, but it also suffers from the drawbacks of *RandomSelect*, which spreads the test targets too thinly when the number of seeds is large, and does not adequately test the OptPaths that actually trigger the optimization.

The OptPath proposed in this paper may contain some branches that exit early without triggering the optimization. In subsection 4.4, we point that the impact of these extraneous paths can be alleviated using a rareness-based scheduling approach. To evaluate the performance of *RarenessSchedule* strategy, we randomly selecte a number of optimization passes from different optimization levels (DFG, FTL, B3) of JavaScriptCore, and manually analyze these optimizations by marking the code locations that actually trigger the optimization. If an OptPath executes to the marked code location, it is considered that the OptPath really triggers the optimization. Table 8 shows how OptPaths are tested in a 72-hour fuzz testing. The second column of Table 8 records the number of OptPaths that OptFuzz explored in different optimization passes, with the data in parentheses indicating the percentage of OptPaths that actually trigger the optimization. For example, OptFuzz explores a total of 156 OptPaths in *StrengthReduction*, of which 28% actually trigger the optimization. The third column of Table 8 records the number of times OptFuzz selects OptPaths, and the data in parentheses indicates the percentage of OptPaths that trigger the optimization among the OptPaths selected. For example, OptFuzz selects 19,015 OptPaths from *StrengthReduction* over 72 hours, and ultimately, the percentage of selected OptPaths that trigger the optimization remains stable at 42%.

Optimization	*OptPath*	$OptPath_{selected}$	↑
StrengthReduction	156 (28%)	19015 (42%)	↑ 14%
TypeCheckHoisting	21 (48%)	3389 (92%)	↑ 44%
ConstantFolding	234 (45%)	24569 (57%)	↑ 12%
StoreBarrierInsertion	116 (28%)	15562 (41%)	↑ 13%
IntegerCheckCom	18 (50%)	2882 (84%)	↑ 34%
IntegerRangeAnalysis	37 (30%)	5358 (63%)	↑ 33%
B3FoldPathConstants	26 (12%)	4043 (32%)	↑ 20%
B3LowerMacros	12 (50%)	2666 (96%)	↑ 46%

Table 8: Percentage of OptPaths that actually trigger the optimization and the scheduling of OptPaths by *RarenessSchedule* in different optimizations.

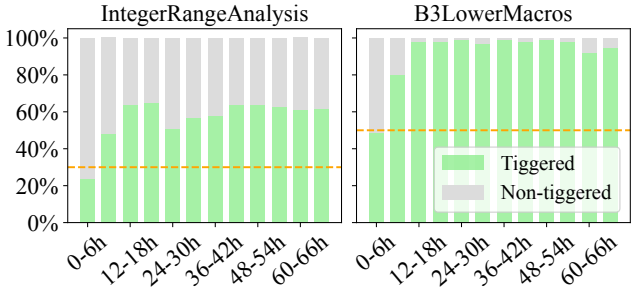

Figure 5: The percentage of OptPaths that actually trigger the optimization over time among the selected OptPaths.

Among the eight optimization passes, only 35% of the OptPaths can really trigger the optimization. The percentage increases to 53% with *RarenessSchedule* strategy and in some of the optimization passes, the percentage of OptPaths that can trigger the optimization is more than 90%. The experimental results demonstrate that even though the OptPaths contain some branches that do not trigger the optimization or exit early, the utilization of *RarenessSchedule* mitigates the impact of these extraneous paths, enabling more computational resources to be allocated for testing the OptPaths that actually trigger the optimization.

Figure 5 illustrates the percentage of OptPaths that actually trigger the optimization over time among the **selected OptPaths** (the third column in Table 8). The orange horizontal line indicates the baseline percentage of OptPaths that actually trigger the optimization (the second column in Table 8). As shown in Figure 5, the proportion of selected OptPaths that really trigger the optimization is not constant during the scheduling process. At the beginning of the fuzz testing, the number of OptPaths is small, and the OptPaths that have been tested a small number of times are not necessarily the ones that really trigger the optimization, resulting in a relatively low proportion of OptPaths that actually trigger the optimization in the selected OptPaths. As the test proceeds, the branches that exit early are tested more often, *RarenessSchedule* is able to select more OptPaths that really trigger the optimization, leading to a gradual rise in the percentage of OptPaths that actually trigger the optimization. Eventually, the percentage stabilize at 63% in *IntegerRangeAnalysis* and 96% in *B3LowerMacros*.

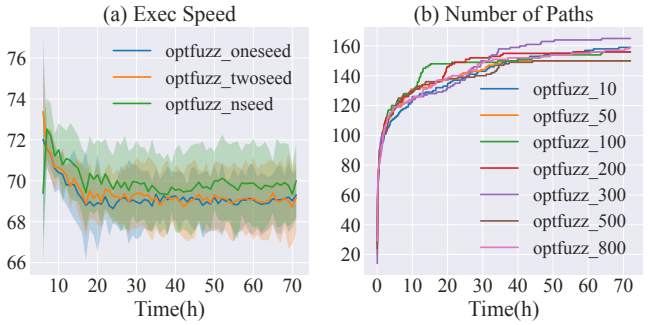

Figure 6: (a)Execution speed of fuzz testing with different trimming strategies. (b)Number of OptPaths in StrengthReduction explored by OptFuzz at different thresholds.

6.4.3 Trimming

To evaluate the trimming strategy of OptFuzz, we compare three trimming strategies including *nseed*, *twoseed* and *oneseed*. *nseed* is the trimming strategy used in this paper. *nseed* produces distinct seeds for the newly executed OptPaths in each optimization pass. *twoseed* generates a maximum of two seeds for an interesting test case. One seed preserves new edges, while the other preserves all new OptPaths, without distinguishing between specific optimization passes. Conversely, *oneseed* only generates a single seed for a test case, which preserves both new edges and new OptPaths.

The results in Table 7 show that *nseed* can find more bugs. *nseed* outperforms *twoseed* and *oneseed* for two main reasons. (1) The trimming with edges and OptPaths in different optimization passes as the trimming standard ensures that each seed is as small as possible, which results in a faster execution of the program. Figure 6(a) shows the execution speed results of OptFuzz running on JavaScriptCore for 72 hours with different trimming standards. The solid lines represent mean and the shades around lines are confidence intervals for five runs. (2) Another reason is that a test case generates seeds for different optimization passes separately, the rareness scheduling strategy will select the OptPaths **in each optimization pass** that are currently being tested less for testing, which not only ensures that different OptPaths in an optimization pass are tested more evenly, but also ensures that different optimization passes are tested more evenly.

6.4.4 Threshold

This section evaluates the impact of different thresholds. The threshold in OptFuzz determines when to start selecting seeds from *Path Corpus*. If the threshold is too low, the fuzz testing does not explore the code sufficiently. The edge coverage feedback should be advantageous upfront. Conversely, a threshold set too high delays or negates the role of path feedback, thereby impacting the exploration of OptPaths.

In our experiments, we employ seven thresholds, with the results indicating that threshold 300 leads to the discovery

Trial	Bug1_exec	Bug2_bind	Bug3_in	Bug4_super
Trial 1	✓	-	✓	-
Trial 2	-	✓	-	-
Trial 3	-	✓	-	-
Trial 4	-	-	-	-
Trial 5	-	-	✓	✓

Table 9: Distribution of bugs found by OptFuzz across fuzzing trials when threshold is 300. ✓ means a bug was found during a fuzzing trial.

of more bugs in Table 7. Table 9 shows the distribution of bugs across fuzzing trials when threshold is 300. Upon further analysis of the bugs identified, we found that OptFuzz_300 is able to find *Bug1* that cannot be found with any other threshold, and the bug is related to *StrengthReduction* optimization pass. Figure 6(b) illustrates the exploration of OptPaths in the StrengthReduction optimization by OptFuzz for different thresholds. The results show that after 30h, Optfuzz_300 explored a higher number of OptPaths compared to the other thresholds, which is probably why OptFuzz_300 found *Bug1*.

6.5 Code Coverage

To answer Q4, this section compares code coverage with Fuzzilli and FuzzJIT. We ran Fuzzilli, FuzzJIT and OptFuzz on JavaScriptCore, SpiderMonkey and Hermes. Each Trial lasted 72 hours and was repeated five times. We use *llvm-cov* [27] to count the coverage results. Table 10 shows the overall code coverage as well as the JIT compiler code coverage obtained by Fuzzilli, FuzzJIT and OptFuzz for testing JavaScriptCore, SpiderMonkey and Hermes. The results in the table are the average of five runs. The corresponding standard deviations are shown in Table 16.

Table 10: Final mean code coverage results

Engine	Fuzzilli Line / Edge	FuzzJIT Line / Edge	OptFuzz Line / Edge
JSC	**55.5% / 40.3%**	41.5% / 26.3%	55.3% / 38.8%
JSC JIT	79.9% / 63.0%	62.1% / 47.8%	**80.5% / 63.7%**
SM	**48.7% / 32.4%**	47.1% / 30.7%	45.0% / 29.2%
SM JIT	64.9% / 45.7%	**65.5% / 46.3%**	59.3% / 41.6%
Hermes	**60.7% / 39.7%**	– / –	55.7% / 34.3%
Hermes Scalar	**71.3% / 60.7%**	– / –	71.1% / 60.5%

Fuzzilli outperforms FuzzJIT and OptFuzz in terms of overall JavaScript engine code coverage. However, different fuzzing methods show varying performance on different JIT compiler targets. OptFuzz achieves the highest code coverage in JavaScriptCore's JIT compiler, while in Hermes' Scalar optimization, the coverage was slightly lower than that of Fuzzilli. Additionally, in SpiderMonkey's JIT compiler, OptFuzz demonstrates the lowest edge coverage. Despite this, OptFuzz found more bugs in the same amount of time. This indicates that focusing on the testing of optimization paths and using OptPaths for guidance is effective in discovering bugs in JavaScript JIT compilers.

7 Related Work

Fuzzing for JavaScript Engines. JavaScript engines have syntactic checks, which would prevent malformed test case from execution. In order to pass these checks, jsfunfuzz generates JavaScript test cases based on manually implemented syntax rules. Mozilla's LangFuzz [20] uses a mutation-based approach, where test cases are first parsed into a syntax tree, and replace nodes in the syntax tree to complete the mutation operation. Skyfire [41] and Superion [42] are both grammer-based fuzzing methods. Skyfire focuses on seed generation by replacing leaf nodes on syntax tree, while Superion focuses on mutation and trimming based on edge coverage. Similar to Superion, Nautilus [2] also performs trimming and mutation on the abstract syntax tree. The difference is that Superion requires a corpus to be prepared beforehand, while Nautilus can generate seeds based on the grammar and mutate on the abstract syntax tree of the seeds.

The following works focus on passing semantic checks. CodeAlchemist [15] first disassembles PoC and regression files into code fragments, and then reconstructs test cases from code fragments using information collected by analyzer. Similarly, Montage [23] mutates test cases based on regression test files fragments. Montage trains a neural network language model (NNML) to predict the next fragment given some sequence of fragments, and utilizes the model for mutation. DIE [32] is also mutation-based fuzzing method. DIE proposes a new mutation strategy that randomly preserves structural and type information in the seeds. PolyGlot [5] takes a further step to other languages. PolyGlot generates test cases in different languages for different test objectives based on their new intermediate representation. SoFi [16] takes another perspective. They make use of the reflection mechanism of the JavaScript language and come up with a mutation strategy based on it.

In recent years, the focus of fuzz testing has shifted to the JIT compiler. Fuzzilli [14] devises numerous generation templates and mutation strategies to trigger JIT, and proposes a set of bytecode-like intermediate representation languages FuzzIL. JIT-Picking [3] compares the result of interpretation and JIT compilation in the process of fuzzing to detect non-crash bugs in the JavaScript JIT compilers. FuzzJIT [43] is also a work that detects JIT compiler bugs through differential testing. The authors designed a template for triggering a JIT by embedding test cases into a *for* loop.

Seed Scheduling. Previous work in seed scheduling has prioritized seeds based on edge coverage [4] [47], call graph [24] and inter-procedural control flow graph (CFG) [38], as well as more security-sensitive metrics like execution time [33], memory accesses [7] [45] [44]. AFLFast [4] prioritizes the edges that are currently tested less often. Ecofuzz [47] proposed a variant of the Adversarial Multi-Armed Bandit model and used it for modeling the scheduling problem. Cerebro [24] proposes a multi-objective optimization(MOO) model together

with a nondominated sorting based algorithm to quickly calculate the Pareto Frontier which decides the prioritized seed. K-Scheduler [38] uses a graph centrality algorithm to evaluate the probability of a seed being able to trigger edges that are currently unvisited based on the inter-procedural CFG. SlowFuzz [33] uses resource-usage-guided evolutionary search techniques to find inputs that trigger worst-case algorithmic behavior. Mem-fuzz [7] uses memory access instrumentation to guide evolutionary fuzzing. It prioritizes the input that accesses previously unseen memory addresses. Memlock [45] is a memory usage guided fuzzing technique for exposing uncontrolled memory consumption. It generates and prioritizes the excessive memory consumption inputs. TortoiseFuzz [44] proposes *coverage counting*, an approach for input prioritization with metrics that evaluates edges in terms of the relevance of memory corruption vulnerabilities.

8 Conclusion

By conducting an analysis on the JavaScript JIT compilers, this paper summarizes three observations: 1) Entering the optimization pass does not mean triggering the optimization; 2) Edge coverage-guided fuzzing will overlook test cases that trigger new optimization paths; 3) Different optimization paths are tested unevenly. This paper introduces a novel fuzzing technique that leverages optimization trunk paths to guide fuzz testing. Specifically, this technique preserves a test case that triggers a new optimization trunk path as a seed and prioritizes the testing of optimization trunk paths that are infrequently tested. Through extensive testing, our proposed method found 36 new bugs in JavaScriptCore, V8, SpiderMonkey and Hermes. 26 of these bugs have already been confirmed or fixed with 3 CVEs assigned. While our current focus lies on JavaScript JIT compilers, we recognize broader applicability of our method across other programming languages such as Java, Lua, among others. We will adapt our technique to these alternative languages in the future work.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 62272442, 61902374, U1736208, and the Innovation Funding of ICT, CAS under Grant No.E161040.

References

[1] Apple. Javascriptcore, the built-in javascript engine for webkit. https://trac.webkit.org/wiki/JavaScriptCore, 2023.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[3] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[5] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 642–658. IEEE, 2021.

[6] Clang. Sanitizercoverage. https://clang.llvm.org/docs/SanitizerCoverage.html, 2023.

[7] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using memory accesses to guide fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.

[8] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[9] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2440–2451, 2022.

[10] FaceBook. Hermes, a javascript engine optimized for fast start-up of react native apps. https://github.com/facebook/hermes, 2023.

[11] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[12] Google. Continuous fuzzing of open source software. https://opensource.google/projects/oss-fuzz, 2023.

[13] Google. Open source javascript and webassembly engine for chrome and node.js. https://v8.dev/, 2023.

[14] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *Network and Distributed Systems Security (NDSS) Symposium*, 2023.

[15] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[16] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2229–2242, 2021.

[17] Hermes. Handle dead phis in simplifyphiinst. https://github.com/facebook/hermes/commit/d8c07dfa, 2023.

[18] Hermes. Move simplifyswitchinst to simplifycfg. https://github.com/facebook/hermes/commit/3462558f, 2023.

[19] Hermes. Only visit reachable blocks in instsimplify. https://github.com/facebook/hermes/commit/de0b8336, 2023.

[20] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[21] Honggfuzz. Honggfuzz found bugs. https://github.com/google/honggfuzz#trophies, 2018.

[22] Chromium Issue. V8 issue 40064863. https://issues.chromium.org/issues/40064863, 2023.

[23] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630, 2020.

[24] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[25] LLVM. Honggfuzz found bugs. https://github.com/google/honggfuzz#trophies, 2023.

[26] LLVM. Llvm loop terminology. https://llvm.org/docs/LoopTerminology.html, 2023.

[27] LLVM. llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html, 2024.

[28] Mozilla. Spidermonkey, a mozilla's javascript and webassembly engine. https://spidermonkey.dev/, 2023.

[29] M.Rash. A collection of vulnerabilities discovered by the afl fuzzer. https://github.com/mrash/afl-cve, 2017.

[30] M.Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2019.

[31] OSSFuzz. Oss-fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017.

[32] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[33] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2155–2168, 2017.

[34] Filip Pizlo. Speculation in javascriptcore. https://webkit.org/blog/10308/speculation-in-javascriptcore/, 2020.

[35] Reactjs. https://react.dev/, 2023.

[36] Jesse Ruderman. Introducing jsfunfuzz. https://bugs.chromium.org/p/project-zero/issues/list, 2007.

[37] Saelo. https://bugs.chromium.org/p/project-zero/issues/detail?id=1789, 2019.

[38] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2194–2211. IEEE, 2022.

[39] Syzkaller. Syzkaller found bugs - linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[40] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[41] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[43] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. Fuzzjit: Oracle-enhanced fuzzing for javascript engine jit compiler. In *USENIX Security Symposium. USENIX*, 2023.

[44] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.

[45] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[46] Wikipedia. Opimizing compiler. https://en.wikipedia.org/wiki/Optimizing_compiler, 2023.

[47] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324, 2020.

A Appendix

A.1 p-value

This section shows the results of Mann-Whitney U Test and the standard deviations for code coverage in Table 10.

Table 11: Mann-Whitney U Test results evaluated based on the result presented in Figure 3. A value less than 0.05 indicates the result is statistically different between OptFuzz and the selected fuzzer.

	Superion	DIE	Fuzzilli	FuzzJIT	OptFuzz
ArgumentsElimination	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	1.000
ObjectAllocationSinking	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	1.000
RangeAnalysis	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	1.000
ScalarReplacement	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	1.000

A.2 Case Study

CVE-2023-38542 is a UAF vulnerability related to function *simplifySwitchInst* and *simplifyPhiInst* in optimization pass *simplifyInst* of Hermes as shown in Figure 7. *simplifySwitchInst* converts *switch* to *branch* based on the input of *switch* and removes the corresponding child node of the phi nodes in the unreachable basic block. *simplifyPhi* is designed for *phi* with only one child node, replacing the phi node with the child node. In order to trigger UAF, Hermes needs to trigger the optimization in *simplifySwitchInst* and then trigger the optimization on line 23 of *simplifyPhi* twice to produce an intermediate representation of $x = phi(x)$ and an intermediate representation that uses x. Finally, the optimization at line 23 in *simplifyPhi* is triggered once more to eliminate x. At this point, a reference to x still exists in the code, resulting in Use-After-Free.

In this paper, we propose a seed preservation strategy based on OptPaths, which is able to select the test cases that trigger *simplifySwitchInst* and *simplifyPhi* and keep the OptPaths intact. The conditions in line 8 and 20 in Figure 7 are more difficult to satisfy, and our scheduling strategy tests more of the OptPaths for which such conditions are difficult to satisfy. In this way, our approach generates the test case that trigger both *simplifySwitchInst* and *simplifyPhi*, leading to the

Table 12: Mann-Whitney U Test results evaluated based on the result presented in Figure 6(a).

	optfuzz_oneseed	optfuzz_twoseed	optfuzz_nseed
optfuzz_oneseed	1.000	**< 0.001**	**< 0.001**
optfuzz_twoseed	**< 0.001**	1.000	**< 0.001**
optfuzz_nseed	**< 0.001**	**< 0.001**	1.000

Table 13: Mann-Whitney U Test results evaluated based on the result presented in Figure 6(b).

threshold	10	50	100	200	300	500	800
p-value	**< 0.001**	**< 0.001**	**< 0.001**	**< 0.001**	1.000	**< 0.001**	**< 0.001**

Table 14: Mann-Whitney U Test results evaluated based on the result presented in Figure 8(b).

	RandomSwitch	AIMD	NoSwitch	Wind
RandomSwitch	1.000	**< 0.001**	**< 0.001**	**< 0.001**
AIMD	**< 0.001**	1.000	**< 0.001**	**< 0.001**
NoSwitch	**< 0.001**	**< 0.001**	1.000	**< 0.001**
Wind	**< 0.001**	**< 0.001**	**< 0.001**	1.000

Table 15: Mann-Whitney U Test results evaluated based on the result presented in Figure 8(a).

	RandomSwitch	AIMD	NoSwitch	Wind
RandomSwitch	1.000	**0.035**	0.788	**< 0.001**
AIMD	**0.035**	1.000	0.082	**< 0.001**
NoSwitch	0.788	0.082	1.000	**< 0.001**
Wind	**< 0.001**	**< 0.001**	**< 0.001**	1.000

Table 16: Standard deviations for code coverage in Table 10.

Engine	Fuzzilli Line / Edge	FuzzJIT Line / Edge	OptFuzz Line / Edge
JSC	0.25 / 0.38	1.20 / 0.75	0.10 / 0.63
JSC JIT	0.33 / 0.33	0.55 / 0.40	0.10 / 0.14
SM	0.39 / 0.32	0.04 / 0.07	0.29 / 0.24
SM JIT	0.46 / 0.29	0.11 / 0.12	0.25 / 0.18
Hermes	0.22 / 0.12	– / –	0.15 / 0.12
Hermes Scalar	0.00 / 0.10	– / –	0.00 / 0.06