# LOOL: Low-Overhead, Optimization-Log-Guided Compiler Fuzzing (Registered Report)

Florian Schwarcz
Johannes Kepler Universität
Linz, Austria
florian.schwarcz@jku.at

Felix Berlakovich
Universität der Bundeswehr
Munich, Germany
felix.berlakovich@unibw.de

Gergö Barany
Oracle Labs
Vienna, Austria
gergo.barany@oracle.com

Hanspeter Mössenböck
Johannes Kepler Universität
Linz, Austria
hanspeter.moessenboeck@jku.at

## Abstract

Compiler fuzzing with randomly generated input programs is a powerful technique for finding compiler crashes and miscompilation bugs. Existing fuzzers for compilers are often unguided and must be manually parameterized to cover different parts of the compiler under test.

In this work we present Lool, an approach for fuzzing a compiler with low overhead, guided by optimization log information produced by the compiler. The optimization log tracks program transformations performed by the compiler on the level of individual methods compiled. We argue that using the optimization log has less overhead than off-the-shelf code coverage tools. At the same time, the optimization log's per-method data gives more information than code coverage collected over a number of distinct compilations. The level of detail of the optimization log is also easy to tune for the use case of guiding a fuzzer.

We are integrating the Lool approach in an existing fuzzer for the GraalVM compiler. A genetic optimization algorithm uses optimization log information for tuning code generation parameters with the goal of covering optimizations that were previously rarely exercised. Initial experiments confirm that varying the generator's parameters is effective at finding new bugs. The genetic algorithm will automate the exploration of the parameter space to improve testing of currently insufficiently fuzzed parts of the compiler.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Just-in-time compilers**.

## Keywords

fuzzing, JIT compiler, genetic algorithm, GraalVM


*FUZZING '24, September 16, 2024, Vienna, Austria*

ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685533

**ACM Reference Format:**
Florian Schwarcz, Felix Berlakovich, Gergö Barany, and Hanspeter Mössenböck. 2024. LOOL: Low-Overhead, Optimization-Log-Guided Compiler Fuzzing (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3678722.3685533

## 1 Introduction

Bugs in compilers can have a significant impact on users, especially when the symptom is not a crash, but a silent miscompilation. Ascertaining that a suspected program bug is actually a miscompilation is a time-consuming process for software developers. Hand-written tests for the compiler, such as unit tests for specific components, typically cover cases the compiler engineer already envisions during development, but edge cases are easily overlooked.

Our work concerns testing of the GraalVM compiler, a compiler for Java and other languages (Section 3.3). Being written entirely in Java, the GraalVM compiler is not affected by memory safety issues plaguing compilers written in C or C++. However, the GraalVM compiler is not immune to implementation errors, occasionally leading to miscompilations or failed assertions, which lead to a crash. An effective way to compensate the blind spot of hand-written tests is *compiler fuzzing*. Compiler fuzzing aims to cover the missing test scenarios by feeding random inputs to the compiler. We developed a compiler fuzzer targeting the GraalVM compiler and, in line with other compiler fuzzing work [38], found several previously unknown bugs.

However, the ongoing effectiveness of a compiler fuzzer depends on the capabilities of the input code generator. The generator can trigger certain compiler optimizations only with very specific combinations of language features. A naive input generation approach uses a fixed probability distribution of language features to include. A problem with this approach, however, is the marginally small probability of certain language feature combinations occurring in a test program of limited size. If these combinations are necessary to trigger a specific optimization, the optimization remains untested with a high probability. Groce et al. propose a method called *swarm testing* that varies code generator configurations throughout a fuzzing campaign to generate more diverse test cases [14]. In the literature as well as in our own experiments,

this has proven to be effective. *Directed* swarm testing [2] is a refinement of this method that guides the configuration mutation by code coverage data collected during testing.

Code coverage is a popular feedback mechanism for steering a fuzzer's input generation [26]. "Code coverage" is an umbrella term for several different kinds of coverage (e.g., basic-block coverage, edge coverage, *n*-gram coverage). Given the large number of AFL descendants, AFL's edge coverage is presumably the most common coverage used. However, when fuzzing software written in Java, AFL's assembly-level coverage instrumentation is too fine-grained. Instead, a solution for Java is to instrument Java bytecode instead of assembly code, for example with JaCoCo[1]. JaCoCo incurs a non-trivial overhead and lacks the frequency information of AFL's edge-coverage instrumentation. A common shortcoming of all these coverage metrics is the lack of context.

Based on these observations, in this work we extend a fuzzer for the GraalVM compiler to leverage a new kind of coverage: We propose to use the GraalVM compiler's optimization log for coverage feedback. Our fuzzer uses the precise, domain-specific information from the optimization log to guide the input generation towards rare or entirely uncovered compiler optimizations. At the core of the input generation is a genetic algorithm that uses the coverage feedback to choose among different generation options. During our experiments so far, we have found 30 previously unknown bugs in the GraalVM compiler.

To summarize, this paper contributes the following:

- We present Looʟ, an optimization-log guided fuzzer that uses domain-specific knowledge to reach even rare compiler optimizations.
- We describe the relevant details of the GraalVM compiler fuzzer (Section 4) and the optimization-log guided, genetic input mutation (Section 5).
- We lay out a detailed evaluation plan for Phase 2 of the Fuzzing workshop (Section 6).

## 2 Motivation

Our goal in this work is to change the code generator's parameters in order to produce code that triggers rare optimizations more often. Like other fuzzers based on fixed probability distributions, GraalVM's fuzzing also has the problem that it does not exercise all parts of the compiler equally. Some optimizations are very frequent, while others are triggered very rarely. Figure 1 shows the counts for certain optimizations we recorded for one fuzzing campaign with 1000 randomly generated Java programs using the code generator's default configuration. The least frequent optimization in this data set occurs 11 times overall. In contrast, other optimizations apply hundreds of thousands or even millions of times on the same 1000 test programs. Our goal is to steer the GraalVM fuzzer towards exercising the rarer optimizations.

Code coverage would, in principle, tell us whether certain optimizations were exercised or not. For Java, code coverage can be either line coverage (e.g., JaCoCo) or an AFL-like coverage bitmap

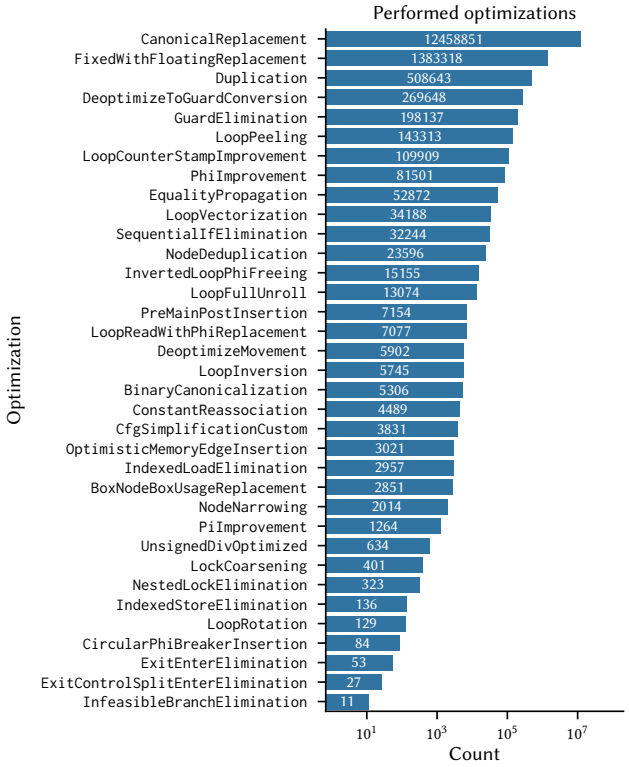


**Figure 1: Counts of selected optimizations during the compilation of 1000 fuzzed test cases (logarithmic axis).**

that additionally records execution frequencies as orders of magnitude (e.g., JQF [29], Jazzer [19][2] or kelinci [21]). However, code coverage suffers from two major drawbacks:

**Overhead** Collecting code-coverage information while fuzzing the GraalVM compiler comes with non-negligible overhead. For example, Nagy and Hicks report a 36% overhead for AFL [28], and we observed a 17% decline in test case throughput with JaCoCo code coverage. Similarly, the coverage instrumentation of JQF [29] and Kelinci [21] relies on bytecode instrumentation that adds runtime overhead.

**Context Insensitivity** JaCoCo's code coverage does not contain information about the *context*, such as the domain knowledge that a compiler compiles separate methods. As a result, conventional coverage tools merge the coverage information for the separate runs of the JIT compiler on these methods. Such merged information can tell us that on a given input program the compiler optimizations $X$ and $Y$ were exercised, but not whether optimizations $X$ and $Y$ happened during the same compilation of a single method.

In contrast to code coverage, our Looʟ approach to guided compiler fuzzing has low overhead, is precise, and can be made context sensitive according to an appropriate domain-specific notion of contexts. Consider the example loop in Figure 2a. This loop contains

[1] https://www.jacoco.org/jacoco

[2] According to the Github repository, Jazzer switched to JaCoCo to provide coverage

```
int sum = 0;
for (int i = 0; i < array.length; i++) {
    if (i == 0)
        log("summing non-empty array %s", array);
    sum += array[i];
}
```

**(a) Example loop not amenable to loop vectorization due to control flow and a method call in the loop.**

```
int sum = 0;
if (array.length > 0) {
    log("summing non-empty array %s", array);
    sum += array[0];
    for (int i = 1; i < array.length; i++) {
        sum += array[i];
    }
}
```

**(b) Example loop after loop peeling. The remaining loop is amenable to loop vectorization.**

**Figure 2: Combination of compiler optimizations.**

control flow and a method call, making the common optimization of *loop vectorization* inapplicable. However, applying *loop peeling* to separate the first loop iteration from the rest produces the loop in Figure 2b, which can be vectorized. Such information about pairs of optimizations can be interesting, since many intricate compiler bugs involve interactions between multiple compilation phases.

Code coverage could tell us that the separate optimizations have taken place as the compiler was compiling a given program, but it cannot tell us that they took place within the context of the compilation of a single method. In contrast, our coverage based on the optimization log can be configured to record the fact that the optimizations happened in the same method or even on the same loop. Thus, a context-sensitive coverage metric that is aware of the context of separate compilations, and that does not merge information from separate contexts, can guide the fuzzer in the direction of such interesting optimization pairs.

# 3 Background

## 3.1 Genetic Algorithms

Our proposed solution is based on a genetic algorithm [17]. In general, a genetic algorithm operates on a set of *individuals*. The set of individuals is called the *population* and is iteratively mutated and evaluated. The evaluation of each individual in a population yields a *fitness* score to judge the individual's ability to reach some predefined goal. Based on this score, the genetic algorithm creates a new *generation* (the next population) whose individuals are mutated descendants of selected individuals of the previous generation. The process of mutating individuals or combining individuals to form an individual of the new generation is sometimes called *reproduction*. A high fitness score raises the chance of the genetic algorithm choosing an individual for reproduction. This selection process ensures that desirable individuals can pass their properties to more children while less desirable individuals might not reproduce at all.

## 3.2 JIT Compiler Fuzzing

As most software is processed by a compiler at some point, and compiler bugs can have severe consequences, compiler fuzzing is of particular importance [25, 37, 38]. Compiler fuzzers are faced with a number of specific challenges. First, constructing valid inputs (i.e., inputs that are not rejected at an early stage) typically requires specialized input generators to prevent the majority of inputs being rejected by the parser. Second, miscompilations, i.e., incorrect compilations that crash neither the compiler nor the compiled program, but lead to erroneous programs, are hard to detect. Generic fuzzers typically use sanitizers to detect non-crashing, but faulty program states in their subjects. Sanitizers cannot, however, detect miscompilations. Third, like fuzzers in general, compiler fuzzers try to exercise previously unexplored states in the state space of the compiler. To that end, compiler fuzzers must be able to *differentiate* such compiler states and *steer input generation* to reach new states.

JIT compilers face a specific fourth challenge: When a JIT compiler runs as part of a virtual machine (VM), even valid inputs might run solely in the VM's interpreter, thus never reaching the actual fuzzing target.

Thus, JIT compiler fuzzers must solve the following problems:

- A How to generate inputs that are both syntactically correct and do not exit early due to a semantic error?
- B How to generate inputs that are actually JIT compiled?
- C How to detect miscompilations?
- D How to differentiate compiler states?
- E How to adjust the input generation to reach new states?

## 3.3 GraalVM

We are studying compiler fuzzing in the context of GraalVM, a family of technologies for high-performance execution of programs written in various programming languages[3].

*3.3.1 The GraalVM compiler.* GraalVM features both just-in-time (JIT) and ahead-of-time (AOT) compilation of programs written in Java and other languages such as Scala or Kotlin that compile to Java virtual machine (JVM) bytecode. GraalVM also includes the Truffle language implementation framework [36]. Truffle allows users to implement, and obtain a JIT compiler for, domain-specific languages or other programming languages. The GraalVM project maintains Truffle implementations of several common languages including Python, JavaScript, and Ruby. The common GraalVM compiler compiles all languages running on GraalVM (i.e., Java and Truffle languages) to machine code. Thus, bugs in the compiler affect *all* GraalVM use cases and the compiler's reliability is of utmost importance to the GraalVM project.

*3.3.2 GraalVM optimization log.* The GraalVM compiler can generate an optimization log during compilation. This log includes entries for every optimizing code transformation the compiler performs. Full log entries include the compiler phase performing the optimization, the name of the optimization performed, and the source code location of the program position where the optimization was performed. An example optimization log entry is shown in Figure 3. The log data is used by various debug tooling and can be exported as a text file in JSON format.

[3] https://www.graalvm.org

```
"phaseName": "UseTrappingNullChecksPhase",
"optimizations": [ {
    "optimizationName": "UseTrappingNullChecks",
    "eventName": "NullCheckInsertion",
    "position": {
      "HashCodeTest.hashCodeSnippet01(Object)": 1
    }
} ]
```

**Figure 3: Example GraalVM compiler optimization log entry, describing the elimination of an explicit `null` check.**

An abridged form of the optimization log only increments a counter for each optimization performed. These counters have very low overhead compared to the full log or code coverage information. Our current work uses only the optimization counters as feedback to guide our fuzzer. Future extensions will use a fuller log, which will allow us to use context-specific information, such as whether two different optimizations happened in the same method.

## 4 GraalVM Compiler Fuzzing

In this work we focus on fuzzing JIT compilations of Java code using the GraalVM compiler. To that end, we built a new fuzzer that specifically targets the GraalVM compiler. Since our fuzzer differs in certain aspects from existing Java fuzzers, we describe the relevant details in the following sections.

### 4.1 Input Program Generation

Our fuzzer generates Java source code based on the "liveness-driven" random code generation approach [3]. In this approach, all values computed by the program are eventually used by later statements in the program. This avoids unused computations being trivially eliminated by the compiler, thus increasing the density of "interesting" code per test case. Our generator ensures syntactically and semantically correct inputs by design, solving Problem A.

Our generator supports a large subset of Java, including:

- class structures: multiple classes with inheritance, multiple methods per class with overriding of inherited methods, `enum` classes, both `static` and non-`static` fields in classes
- statements: assignments, `if` statements, `while` and `for` loops including `break` and `continue`, `synchronized` blocks
- expressions: use of method parameters, class fields, local variables, array elements (inside `for` loops only); arithmetic, comparison, logic, and conditional expressions
- method calls: calls to methods defined in the same generated program, to a helper library, or to a small list of explicitly allowed pure methods from the Java standard library

The likelihood of generating each of these features and their exact form (e.g., the number of classes and methods generated or maximum statement depth) can be configured with parameters. We call a set of such parameters a *parameter vector*.

Our generator also combines these features into certain patterns, such as map-style `for` loops (i.e., loops that read an input array, compute on each element, and write corresponding elements of a result array). Such patterns enable specific optimizations [24].

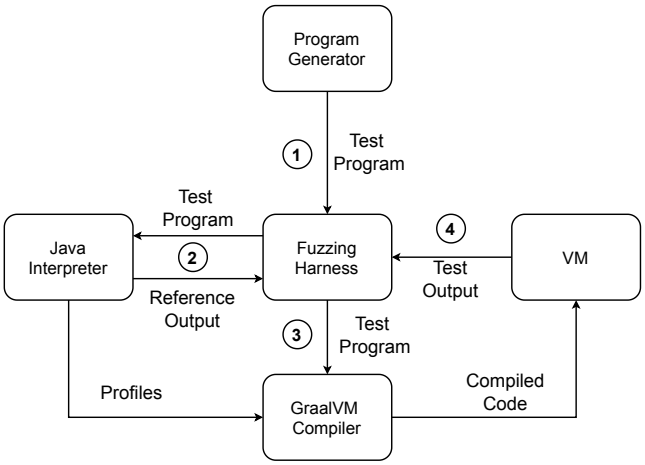


**Figure 4: Overview of the GraalVM fuzzing framework.**

The generated programs include a `main` method that calls other generated methods and prints their results. The inputs to these calls are embedded in the code as constants or are pseudo-randomly generated by the helper library, with a fixed seed per program. The helper library included with each program also contains methods for printing and hashing of data for which Java's default `toString` and `hashCode` methods don't provide deterministic behavior across runs. Thus we generate fully self-contained, stand-alone programs that print the same deterministic data on each run. The generator itself uses a pseudo-random number generator and embeds the seed as a comment in the program for easy reproducibility.

### 4.2 Test Execution

Our fuzzer tests the generated programs using a test harness that compiles the Java source code to bytecode with `javac` and loads the bytecode classes. Fig. 4 gives an overview of the test execution. First, the harness receives the generated program from the input generator ① (see Section 4.1). The harness executes the program's `main` method in the Java bytecode interpreter and records its output as the reference output ②. After interpreting the test program, the harness compiles methods in the program with the GraalVM compiler ③. The harness compiles all methods starting from `main` up to a certain call graph depth and installs the compiled code as the default. The explicit compilation of methods with existing profiles avoids the need for JIT-compilation triggering snippets. As all later executions of the test program execute the *compiled* methods, this compilation step solves Problem B, i.e., inputs not reaching the JIT compiler. Running the interpreter prior to the compilation, but within the same VM instance, has two advantages:

(1) The interpreted run warms up the VM's profiles, which guide certain compiler decisions. Compilation of methods with warmed up profiles resembles real-world compilations more faithfully. We found certain profile-dependent bugs only thanks to the warmed up profiles.
(2) The test program has to run in the interpreter only once. In contrast, many Java fuzzers interpret a test program twice:

```
public static void main(String[] args) {
 double var12 = 95.5965959289008;
 double var13;
 for (int i2 = 527935578; i2 <= 527935876; i2 = i2 + 3) {
  var13 = i2;
  var12 = var12 - Math.log10(Long.sum(1706025860L,
                                      (long) var13));
 }
 GraalDirectives.deoptimize();
}
```

**Figure 5: Example bug after test case reduction. The GraalVM compiler crashed while compiling this method.**

once in interpreter-only mode and once with both the interpreter and the JIT compiler enabled. Thus, the test program runs entirely in the interpreter the first time and at least partially in the second run, until the hot methods are JIT-compiled [1, 23, 35]. Such an approach is more time-consuming due to the (initially) slower execution of interpreted code. More importantly, with such an approach only certain parts of the test program actually reach the JIT compiler.

Finally, the harness executes the `main` method again, this time using the newly GraalVM-compiled methods, and compares the output to the reference output ④.

The harness detects crashes of the GraalVM compiler and mismatches in the outputs of the reference and compiled runs, and reports these as errors. The differential testing against the interpreter solves Problem C, i.e., detection of miscompilations.

This fuzzing framework has been in regular use within the GraalVM project for three years. It has been effective at finding both miscompilations and bugs that crash the compiler with failed assertions or unexpected exceptions being thrown.

We use off-the-shelf source code reducers such as Perses [31] to reduce programs that provoke errors to minimal versions for easier debugging. Figure 5 shows an example of a reduced, crashing compiler bug found by our GraalVM fuzzer.

## 5 Design and Implementation of LOOL

In Section 4 we described how our GraalVM fuzzer solves three of the five challenges described in Section 3.2 (Problems A to C). For the remaining challenges (Problems D and E), we propose a new, domain-specific approach. Lool uses the compiler's optimization log (see Section 3.3.2) to select among suitable parameter vectors. The selection process is driven by a genetic algorithm that tries to breed more desirable parameter vectors over time. Fig. 6 shows an overview of the interaction between the genetic algorithm, the input generator and the optimization log.

### 5.1 Overview

The idea of using an evolutionary algorithm in (compiler) fuzzing is not new. In fact, producing new inputs with an evolutionary algorithm is at the core of the popular fuzzer AFL and its descendants [39]. Some fuzzers even use genetic algorithms, a subbranch

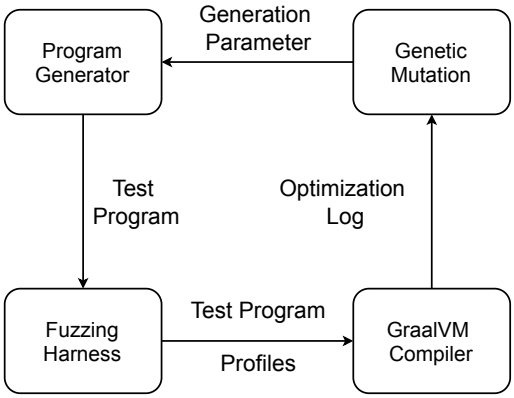


**Figure 6: Genetic mutation of input programs based on optimization-log coverage.**

of evolutionary algorithms that includes crossover, to generate inputs [18, 32]. However, these fuzzers typically apply the evolutionary algorithm *directly* to the inputs (e.g., the generated programs).

In our case, mutating the input program directly would, with a high probability, destroy the liveness and semantic-correctness properties of the input. Instead, our genetic algorithm mutates parameter vectors (see Section 4.1). That is, a population consists of different parameter vectors, each of which we use to generate a certain number of input programs. A parameter vector is desirable if it triggers new optimizations or bugs. The approach of mutating the input generator's parameter space instead of the inputs (i.e., test programs) resembles the idea of Zest [30]. Zest aims to bridge the gap between bit-level mutations operating on bit sequences and high-level structural mutations on syntactically valid inputs.

The genetic algorithm starts with a set of baseline parameter vectors, fuzzes with each vector and builds a new generation of parameter vectors based on the fuzzed compiler's feedback. This process repeats for a set number of generations if specified, otherwise it runs indefinitely.

### 5.2 Shrinking the Search Space

Even with a relatively small number of tunable probabilities in a parameter vector, the search space is large. A large search space increases the time until the genetic algorithm discovers more interesting parameter vectors and thus slows down the fuzzing effort. Our code generator has over 120 parameters, so we restrict the parameter vectors to a subset of the possible parameters. The reason for this choice is that some parameters have a significant influence on the generated code, while changing others is less noticeable.

We experimented with subsets of the parameters in the full parameter vector to analyze how the generator reacts to changes of each parameter. By matching the resulting optimization logs with sets of parameters, we calculated a rough correlation between parameters and optimizations. We consider all parameters that show a non-negligible absolute correlation coefficient ($|r| > 0.3$) to at least *some* optimizations to be good candidates for mutation.

More specifically, during our preliminary experiments, we restricted the genetic algorithm to consider five of the ten possible

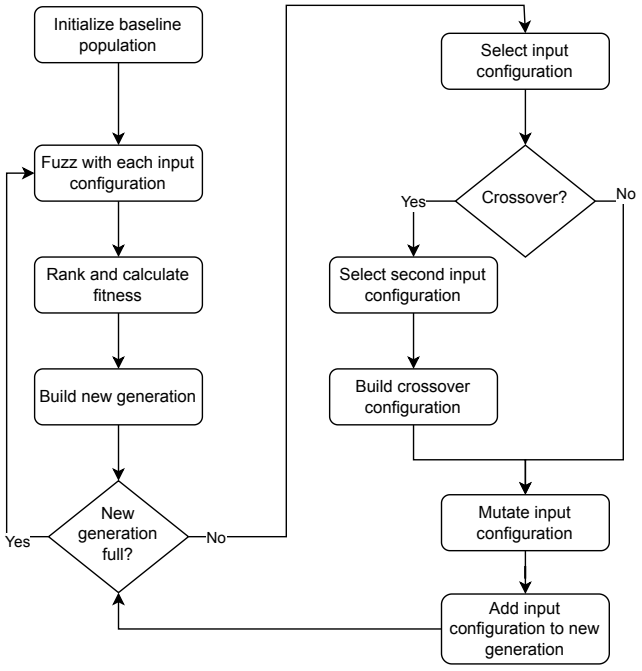


**Figure 7: Workflow of our genetic algorithm loop.**

statement types: `if` statements; fold-style `for` loops; map-style `for` loops; `while` loops; and `synchronized` blocks. Similarly, we limited the optimization of expression probabilities to six of ten: local variable initialization; local variable usage; parameter usage; binary (arithmetic or logic) expression; call of a generated method; and ternary (conditional) expression.

## 5.3 Fitness

After fuzzing with each parameter vector in a generation, the genetic algorithm assesses the fitness of each vector. Our desired outcome is a population of diverse parameter vectors, each of which either triggers different optimizations with a greater likelihood than its siblings, or even triggers bugs. As we want to score parameter vectors relative to each other, we do not compute their fitness scores in isolation, but relative to the entire generation. For this we compare the parameter vectors' fitness along multiple dimensions. Specifically, we are interested in the number of *different rare optimizations* the parameter vector triggered and in the *number of bugs* discovered.

In preliminary experiments we identified seven GraalVM compiler optimizations that were reported fewer than 100 times in a batch of 1000 test cases. We used the counts of only these seven rare optimizations in the optimization log for ranking individuals by fitness. More frequent optimizations did not contribute to the fitness score. Additionally, we used the counts of bugs and nonterminating compilations that had occurred. As bugs and timeouts are infrequent, usually only a few parameter vectors have a count greater than zero in this category. In these cases the algorithm considered only individuals with a strictly positive count, so as not to reward vectors for doing nothing.

For each of these nine dimensions, we rewarded the ten best individuals with a score from 10 to 1, multiplied by a weight. The weight was configured manually and depended on the importance of stressing a specific optimization, e.g., because its code was relatively new and less tested. Considering that the primary goal of the fuzzer is bug finding, the weight for the bug and timeout dimensions was generally higher than for triggering an optimization.

To avoid overfitting on bugs that are easy to trigger, we scanned the error messages for specific substrings to identify known bugs and exclude them from counting. Improving this deduplication, e.g. with more specific error codes, is the subject of ongoing research. Overfitting on the seven optimizations poses a similar threat, so we plan to dynamically select the N rarest optimizations of a generation for ranking in Phase 2 of this work.

## 5.4 Building Generations

As shown in Fig. 7, the genetic algorithm mutates and crossbreeds parameter vectors to form a new generation. During mutation the genetic algorithm adjusts the parameters in a parameter vector. Most of these parameters are independent of each other. For example, parameters such as `InitInStaticBlock` (probability that a variable is initialized in a static initializer block) describe probabilities that guide binary decisions in the generation process. We randomly decrease or increase independent probabilities by a small amount while staying in the valid range, typically between 0.05 and 0.95. Similarly, for discrete parameters, such as `StatementDepth` (the maximum expression depth of a statement), we decrease or increase their value by a small discrete amount while staying within the defined bounds.

Statement and expression distributions form probability distributions, where each statement or expression type has some probability. We mutate distributions by selecting an entry which "steals" from another entry's probability by increasing itself and decreasing the other. This ensures that the total sum of probabilities in a distribution stays constant. As we only include a subset of the distribution in the parameter vectors (see Section 5.2), we equally distribute the difference to 1 among the excluded probabilities. With a low probability, currently 5%, we perform an extreme mutation where the property steals from *every* other probability, biasing the distribution towards the chosen property. Table 1 shows an abbreviated example of a parameter vector before and after mutation.

With a configurable probability, we perform a crossover of two parameter vectors to create a vector for the next generation. Independent probabilities and discrete parameters are each chosen randomly from either parent, but distributions are chosen as a whole to prevent violation of the constraints.

# 6 Evaluation

This section summarizes the preliminary experiments we have run so far, as well as our hypotheses for future detailed evaluation of the Lool approach.

## 6.1 Impact of Code Generation Parameters

*Preliminary experiments.* We performed initial experiments with varying the code generator's options to explore different distributions of generated code constructs. Besides manual tuning and

**Table 1: Simplified genetic representation of an individual before and after mutation. For example, Probabilities shifted from `While` to `MapFor` and from `Param` to `Local`. Note that the table does not show all the probabilities, which is why the sum of probability distributions does not sum up to 1.**

| Parameter | Before | After |
|---|---|---|
| Statement distribution | | |
| `If` | 0.0637 | 0.0637 |
| `FoldFor` | 0.0726 | 0.0726 |
| `MapFor` | **0.0767** | **0.1267** |
| `While` | **0.1761** | **0.1261** |
| Expression distribution | | |
| `Init` | 0.0703 | 0.0703 |
| `Local` | **0.0891** | **0.1291** |
| `Param` | **0.1206** | **0.0806** |
| `Binary` | 0.1059 | 0.1059 |
| Independent probabilities | | |
| `InitInStaticBlock` | **0.2548** | **0.2367** |
| `InheritClass` | **0.5162** | **0.5001** |
| `Cloneable` | 0.2074 | 0.2074 |
| Discrete parameters | | |
| `StatementDepth` | **2** | **3** |

random modification of options, we systematically explored edge cases by biasing statement and expression distributions heavily towards one element of the distribution (Section 5.2).

Inspection of the optimization logs generated by the compiler confirmed that some parameters correlate with particular optimizations. This was, of course, expected at a high level: For example, loop optimizations can only apply on code containing certain loop shapes. Disabling generation of loops will therefore disable loop optimizations. Our goal is to automatically learn more complex relationships between sets of parameters and optimizations, so that we can guide the fuzzer towards less frequently seen optimizations.

These initial experiments over a time frame of about three months identified 30 new crashing bugs in the GraalVM compiler. We believe that most of these bugs would very likely not have been found during routine fuzzing with unmodified parameters, as daily fuzzing of GraalVM generates new bug reports at a much lower rate. About half of these new bugs were found by various parameter tuning approaches, while the other half was then found by prototype versions of the genetic algorithm.

*Hypotheses.* We hypothesize that further, directed, variation of the code generation parameters will discover many new compiler bugs compared to the code generator's default configuration.

## 6.2 Overhead of Coverage Information

*Preliminary experiments.* Our goal in using the optimization log, and specifically only the counters recorded by the optimization log machinery, is efficient collection and subsequent processing of only *relevant* coverage data. As our preliminary measurements show, recording of line coverage information of the GraalVM compiler with JaCoCo adds significant overhead to the execution time of our fuzzing runs, even without analyzing the recorded coverage information. This overhead reduces the number of test cases we can execute in a given amount of time.

*Hypotheses.* We hypothesize that full line coverage information of the compiler contains too much irrelevant information with regards to our goal of guiding the fuzzer towards rarely executed optimizations. While it would be possible to extract only the relevant information that is comparable to the optimization log counters (coverage of exactly those lines that call the optimization log machinery), this would involve recording and then parsing masses of irrelevant data. We intend to show in more detail that an approach based on the optimization log can increase the number of rare optimizations executed and find bugs with low overhead.

We also hypothesize that optimization log data is useful for increasing overall *code* coverage of the compiler. Our intuition for this hypothesis is as follows: Rare optimizations are triggered by particular code shapes, and their implementation checks particular preconditions on those code shapes. As the guided fuzzer learns to generate such code shapes, it will also learn to generate similar code shapes that "get close to", but do not actually trigger, the rare optimizations in question. Thus we suspect that a fuzzer that is good at triggering rare optimizations should also be good at triggering other code paths in the compiler phases implementing those optimizations. Overall code coverage of these compiler phases should therefore increase. Coverage in phases implementing more common optimizations should not suffer, since common code paths will still be executed by many generated tests.

## 6.3 Context-Sensitive Coverage Information

*Problem statement.* While coverage metrics such as line coverage or method coverage record much irrelevant information, they also do *not* record information that may be relevant in the particular application domain. Recall from Section 4 that our fuzzed test programs contain multiple methods, which are all compiled in a run of the test. As noted in Section 2, it can be useful to keep the coverage information for the compilations of individual methods separate from each other. This would allow us to determine whether certain combinations of optimizations applied within the same method, which would be useful for uncovering bugs only found through the interactions of multiple complex optimizations.

The full GraalVM optimization log currently records more information than we would need to search for pairs of optimizations within a method being compiled. The simpler optimization log counters merge information from separate compilations, so they are not context-sensitive.

*Hypotheses.* The internal GraalVM API for recording optimization log information is clean, so as domain experts it will be simple to implement recording of just the context-sensitive set of optimization events per compiled method. We hypothesize that this modification will allow us to guide the fuzzer towards exercising previously untested pairs of optimizations while keeping a low overhead due to not recording any irrelevant data.

## 6.4 Future Evaluation Plan

We will finalize the implementation of our genetic algorithm for guiding the GraalVM compiler fuzzer based on feedback from the optimization log. Given the above observations, we are planning to perform the following full experiments for Phase 2 of this work.

*6.4.1 Experiment Evaluation.* We will evaluate four different configurations, each running 10 times for 24 hours:

**Default Configuration** hardcoded parameters in the code generator, as currently used in day-to-day GraalVM fuzzing.

**Context-Insensitive Looʟ** a variant of Looʟ that uses optimization log counters, with the objective of significantly increasing the number of previously rare optimizations.

**Context-Sensitive Looʟ** a Looʟ variant considering per-method optimization pairs, with the objective of exercising as many distinct pairs of optimizations as possible.

**AFL** a configuration where we replace optimization-log coverage with AFL code coverage and replace the input mutation with AFL. Specifically, we will use bytecode-instrumentation like in JQF [29] to achieve AFL code coverage including frequencies. We will let AFL mutate the parameter space by using a proxy binary as AFL's test subject as implemented in Kelinci [21] and JQF [29].

During each run, the Looʟ configurations try to breed as many generations as possible. Each generation consists of 16 parameter vectors and the fuzzer creates a new generation when each vector has generated 40 tests or after one hour, whichever happens first.

We will evaluate the different configurations according to the following criteria:

- number of rare optimizations applied
- number of interesting optimization pairs applied
- number of unique bugs found
- amount of code coverage, measured as line coverage

We will additionally compare the overheads of collecting and parsing the two types of optimization log coverage data with the overhead of collecting method and line coverage using JaCoCo and AFL coverage using JQF's coverage instrumentation. The success criterion is significantly reduced execution time of compiler tests with optimization log data collection vs. standard code coverage tools.

## 7 Related Work

*Swarm testing.* Looʟ follows the idea of swarm testing [14]. The insight behind swarm testing is that inputs including more features are not necessarily beneficial and in some cases even detrimental to the testing effort [13]. Features, in our case Java language constructs, can either trigger, suppress or be irrelevant for a certain bug. Thus, Looʟ tries to achieve a large diversity of input features, which includes also the *omission* of features in some inputs. Alipour et al. extended swarm testing to be *directed* [2]. Like Looʟ, directed swarm testing incorporates statistical information about triggers and suppressors from previous runs into the generation of inputs. However, Looʟ chooses the targets for its directed swarm testing approach automatically based on the optimization log coverage.

*Alternative coverage metrics.* The majority of fuzzers today uses some variant of code coverage (e.g., branch coverage, block coverage or *n*-gram coverage) for the coverage-feedback mechanism. Recent work suggests, however, that code coverage alone is insufficient to judge a fuzzer's bug-finding performance [5]. In particular, code coverage is often not *sensitive* enough to detect bugs in already covered code [33]. Improved versions of code coverage, such as context-sensitive branch coverage aim to increase the sensitivity but suffer from coverage-state explosion [8]. To mitigate the shortcomings of code coverage, researchers have suggested alternative coverage metrics [11, 12, 16, 22, 27]. In contrast, Looʟ leverages a more abstract and domain-specific coverage metric and is, thus, less prone to state explosion.

*JIT compiler fuzzing.* Fuzzili is a coverage-guided fuzzer for JavaScript VMs that builds and mutates JavaScript input programs based on a custom intermediate language [15]. The generated programs are always syntactically correct, avoid semantic errors, and contain JIT compilation triggering fragments. Fuzzili does not, however, detect miscompilations. JITPicker extends Fuzzili by creating JavaScript functions that expose the interpreter state at different points, such as variable values [4]. With this state exposure, JITPicker can perform differential testing against the JavaScript interpreter and detect miscompilations. FuzzJIT follows the same idea of differential testing against the interpreter [34]. To exercise more potentially interesting parts of the JavaScript JIT, FuzzJIT uses specialized input templates containing, for example, array expressions. JavaTailor follows a similar approach by weaving historical test programs into its seeds[40].

Classfuzz mutates Java class files based on JVM code coverage [10]. Classming follows a similar approach, but it guides the class file mutation based on the executed instructions in the class file [9]. JITFuzz is a coverage-guided fuzzer for the JVM [35]. In contrast to Looʟ, JITFuzz uses Java class files as inputs and mutates them based on the coverage feedback. Similar to Looʟ, the mutations are specifically tailored to exercise JVM JIT compiler optimizations. Artemis generates input programs with Java* Fuzzer [1] and transplants code snippets from JIT compiler test suites into the program [23]. In addition, Artemis tests combinations of compiled and interpreted methods by triggering only compilation of randomly selected methods in the input program. Following the spirit of swarm testing, the authors of JOpFuzz investigate triggering test case features and required JVM options based on real bug reports [20]. In an initial step, JOpFuzz tries to automatically infer the relationship between code features and JIT compiler optimizations. During the fuzzing campaign, JOpFuzz not only mutates Java input programs, but also JIT compiler options passed to the JVM.

Finally, some JIT compiler fuzzers focus on bug categories different from crashes and miscompilations. Confuzzion, for example, tries to find Java type confusion vulnerabilities [6]. Brennan et al. use a fuzzer to detect timing side-channels in JIT-compiled Java programs [7].

*Compiler optimization logs.* Our approach to guided fuzzing should apply to any mature compiler that saves information about the optimizations it performed.

Several production compilers produce some form of optimization log, such as LLVM's Remarks[4] or the Intel Compiler's Optimization Reports[5]. In contrast to GraalVM's optimization log, these compilers also include negative entries in the log for optimizations that were attempted but did not succeed or were not considered profitable. Our current work hypothesizes that logging only successful optimization information is already useful for guiding a fuzzer.

Other compilers, such as GCC, do not produce such structured logs. As an alternative to an explicit structured log, debug dumps of the program state before and after compiler phases could be compared to infer if an optimization has taken place. This would also provide a form of optimization coverage information suitable for use as feedback to a fuzzer. However, if a structured log proves to be helpful for fuzzing, it should be easy for compiler developers to retrofit the required logging in their compilers.

## 8 Conclusions

We have presented Looʟ, a low overhead optimization log guided approach to compiler fuzzing. Looʟ uses feedback from the compiler to guide test case generation. The domain-specific information from the compiler's optimization log can be collected and used with lower overhead than general code coverage information. At the same time, the optimization log is easy to use for recording context-sensitive information, such as optimizations performed during the compilation of a single method.

Our preliminary experiments with the implementation of Looʟ within the GraalVM fuzzing infrastructure have shown promising results. We are planning a thorough evaluation of the Looʟ approach, with the goal of demonstrating that this approach is both efficient in execution time and effective at covering the compiler's code and finding new bugs.

## Acknowledgments

We would like to thank the anonymous reviewers for their feedback that has helped us improve an earlier version of this paper.

This research project was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research Group at Oracle Labs. Oracle, Java, GraalVM, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. We also thank all researchers at the Johannes Kepler University's Institute for System Software for their support of and feedback on our work.

## References

[1] [n. d.]. Java* Fuzzer for Android*. https://github.com/android-art-intel/Fuzzer

[2] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating Focused Random Tests Using Directed Swarm Testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken Germany, 70–81. https://doi.org/10.1145/2931037.2931056

[3] Gergö Barany. 2017. Liveness-Driven Random Program Generation. In *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10855)*, Fabio Fioravanti and John P. Gallagher (Eds.). Springer, 112–127. https://doi.org/10.1007/978-3-319-94460-9_7

[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Los Angeles CA USA, 351–364. https://doi.org/10.1145/3548606.3560624

[5] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1621–1633. https://doi.org/10.1145/3510003.3510230

[6] William Bonnaventure, Ahmed Khanfir, Alexandre Bartel, Mike Papadakis, and Yves Le Traon. 2021. Confuzzion: A Java Virtual Machine Fuzzer for Type Confusion Vulnerabilities. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 586–597. https://doi.org/10.1109/QRS54544.2021.00069

[7] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1011–1023. https://ieeexplore.ieee.org/document/9284028

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046

[9] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. *ACM SIGPLAN Notices* 51, 6 (Aug. 2016), 85–99. https://doi.org/10.1145/2980983.2908095

[11] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696. https://doi.org/10.1109/SP.2018.00040

[13] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. 2013. Help, Help, i'm Being Suppressed! The Significance of Suppressors in Software Testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Pasadena, CA, USA, 390–399. https://doi.org/10.1109/ISSRE.2013.6698892

[14] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, Minneapolis MN USA, 78–88. https://doi.org/10.1145/2338965.2336763

[15] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2023.24290

[16] Adrian Herrera, Mathias Payer, and Antony L Hosking. 2022. Registered Report: Towards a Data-Flow-Guided Fuzzer. In *International Fuzzing Workshop 2022*. 11.

[17] John H. Holland. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[18] Katherine Hough and Jonathan Bell. 2024. Crossover in Parametric Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3597503.3639160

[19] Code Intelligence. [n. d.]. *Jazzer*. https://github.com/CodeIntelligenceTesting/jazzer

[20] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 43–55. https://doi.org/10.1109/ICSE48619.2023.00016

[21] Rody Kersten. [n. d.]. *Kelinci*. https://github.com/isstac/kelinci

[22] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing Guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[23] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3600006.3613140

[24] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–25. https://doi.org/10.1145/3428264

[25] Haoyang Ma. 2023. A Survey of Modern Compiler Fuzzing. https://doi.org/10.48550/arXiv.2306.06884 arXiv:2306.06884 [cs]

[26] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47,

[4] https://llvm.org/docs/Remarks.html

[5] https://www.intel.com/content/www/us/en/developer/articles/technical/getting-the-most-out-of-your-compiler-with-new-optimization-reports.html

11 (Nov. 2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[27] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with Data Dependency Information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 286–302. https://doi.org/10.1109/EuroSP53844.2022.00026

[28] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[29] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 398–401. https://doi.org/10.1145/3293882.3339002

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 329–340. https://doi.org/10.1145/3293882.3330576

[31] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

[32] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

[33] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. 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)*. 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[34] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[35] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 56–68. https://doi.org/10.1109/ICSE48619.2023.00017

[36] Thomas Wuerthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. 187–204. https://doi.org/10.1145/2509578.2509581

[37] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *ACM SIGPLAN Notices* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[39] Zalewski. 2016. AFL Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt

[40] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1133–1144. https://doi.org/10.1145/3510003.3510059

Received 2024-06-21; accepted 2024-07-22