École Polytechnique Fédérale de Lausanne

Fuzzing GraalVM native images

by Eva Vaughan

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Mathias Neugschwandtner
External Expert

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 16, 2024

Abstract

This thesis presents an approach for integrating fuzzing capabilities into the GraalVM Native Image compiler. We implement sanitizer coverage instrumentation directly in the compiler, enabling efficient fuzzing of GraalVM components and applications compiled to native code. Custom harnesses are developed to adapt existing fuzzing tools like Jazzer and Fuzzilli to work with GraalVM's architecture. A novel differential fuzzing technique is introduced to systematically compare the behavior of GraalJS against other JavaScript engines. The implemented instrumentation supports both TracePCGuard and Inline8Bit modes, with performance evaluated using the Renaissance benchmark suite. A fuzzing campaign targeting GraalJS identified 28 bugs, representing approximately 80% of reported issues during the evaluation period. The differential fuzzing approach uncovered compliance issues by comparing GraalJS behavior to other engines. While comparative analysis with existing Java fuzzing tools faced some challenges, preliminary results suggest comparable performance. This work demonstrates the potential for enhancing the robustness and correctness of GraalVM components through integrated fuzzing capabilities, and provides a foundation for further work in this area.

Contents

Chapter 1

Introduction

1.1 Setting

In the realm of software testing and security, fuzzing has emerged as a crucial technique for uncovering vulnerabilities and ensuring robustness. Most native compilers have built-in support for instrumentation tailored for fuzzing, either directly, as seen in Go [4], or by leveraging LLVM, as demonstrated in C and C++ [8] or Rust [6]. However, the GraalVM Java native image compiler, a significant player in the polyglot runtime ecosystem, lacks this inherent capability. This deficiency is particularly noteworthy given the diverse array of applications and languages that rely on this runtime, including Java applications and languages like JavaScript, Python, and Ruby through the Truffle framework. The absence of native fuzzing support for these targets presents a significant gap in the comprehensive security testing of software built on the GraalVM platform.

1.2 Main challenges

The primary challenge addressed in this work is the implementation of effective fuzzing techniques for the GraalVM ecosystem, with a particular focus on Graal.js, the JavaScript implementation for GraalVM. This endeavor is complicated by the unique architecture of GraalVM, which employs partial evaluation which differs significantly from traditional JavaScript engines. Consequently, the effectiveness of existing JavaScript fuzzing tools, which are often tailored to specific JIT compiler behaviors, may be limited when applied to Graal.js.

1.3 Related works

In the context of Java-based fuzzing, Code Intelligence proposed Jazzer [7], a tool for JVM bytecode instrumentation. Jazzer operates by instrumenting classes as they are loaded by the JVM's `ClassLoader`. Its instrumentation is designed to be compatible with LibFuzzer, a popular coverage-guided fuzzing engine. To bridge the gap between Java and the native LibFuzzer harness, Jazzer employs the Java Native Interface (JNI) to insert calls around compare operations when cmplog is enabled, and to initialize coverage counters.

Jazzer's capabilities extend beyond basic coverage guidance, incorporating LibFuzzer's cmplog feature and custom array hooks. This comprehensive approach has led to its adoption by Google's OSS-Fuzz project for large-scale fuzzing of Java libraries. The success of Jazzer in this domain underscores the importance and potential impact of effective fuzzing tools for managed language environments.

However, while Jazzer represents a significant advancement in Java fuzzing, it operates at the bytecode level and is primarily designed for traditional JVM environments. This leaves a gap in the context of GraalVM and its native image compilation, where the runtime characteristics and optimization opportunities differ from standard JVM implementations.

1.4 Motivation

The GraalVM ecosystem has become increasingly mature and used, with many applications and components relying on native images for improved performance and reduced startup times. However, the ability to effectively fuzz binaries built with Native Image has been limited, potentially leaving vulnerabilities undiscovered. While programs built using memory-safe languages like Java are generally less susceptible to traditional memory corruption vulnerabilities, they are not immune to bugs that can lead to unexpected program states, denial of service (DoS) attacks, or other security issues. Therefore, developing robust fuzzing techniques for Native Image binaries is crucial for ensuring the security and reliability of the GraalVM ecosystem.

Furthermore, several language interpreters within the GraalVM framework are implemented in Java and subsequently compiled to native code using Native Image. Bugs in these interpreters can have serious consequences, potentially causing the interpreter to fail and exit abruptly instead of raising a managed guest language exception. More subtly, execution that deviates from the interpreted language's specification can lead to unexpected behavior in user code, which could be exploited for malicious purposes. This is particularly critical for widely-used components like GraalJS, where ensuring compliance with the ECMAScript specification is paramount for maintaining compatibility and security across the JavaScript ecosystem.

Our approach addresses these challenges by providing a comprehensive fuzzing solution tailored to the unique characteristics of GraalVM and Native Image. By integrating fuzzing capabilities directly into the Native Image compiler, we enable more efficient and effective testing of a wide range of GraalVM components, including language interpreters and libraries. This is particularly important given that Jazzer, a previously popular Java fuzzing tool, has been deprecated and operates at the bytecode level, which is not ideal for Native Image binaries. Our method not only improves upon existing Java fuzzing techniques but also extends fuzzing capabilities to the native code generated by GraalVM, filling a critical gap in the security testing of this ecosystem.

Moreover, our differential fuzzing approach for JavaScript interpreters provides a powerful means of detecting subtle deviations from language specifications. By comparing the behavior of GraalJS against other established JavaScript engines, we can identify potential compliance issues that might otherwise go unnoticed. This not only enhances the robustness and correctness of GraalJS but also contributes to the overall security and reliability of the JavaScript ecosystem, as many applications may rely on consistent behavior across different JavaScript implementations.

In summary, our approach is necessary to address the unique challenges posed by the GraalVM ecosystem, particularly in the context of Native Image binaries and language interpreters. By providing effective fuzzing techniques for these components, we contribute to the security, reliability, and compliance of a wide range of applications built on GraalVM, thereby addressing a critical need in the evolving landscape of polyglot runtime environments.

1.5 Research questions

How much performance increase can native in-compiler instrumentation brings compared to bytecode instrumentation?

Jazzer inserts JNI calls in the JVM bytecode for instrumentation, which incurs a significant performance overhead, instrumenting in the native-image compiler allows us to directly insert native calls with less overhead. Previous work [11] suggest that it is the one of the main performance bottleneck for Java fuzzing. We will compare our implementation against Jazzer, without the implementation of [11] we will however see whether we achieve the same relative performance gain over Jazzer or better.

Does an IR-oriented JS fuzzer such as Fuzzilli perform as well on a partially evaluated JS engine such as Graal.js?

Fuzzilli authors motivates their use of an SSA intermediate representation because it will generate test cases that are closer to what the JIT compiler operates on [5], this is however not true for JS engines using heterodox methods such as partial evaluation. The JS engine never touches anything close to an SSA IR. Moreover, the lower levels of the compiler are fuzzed independently, which would render a strictly JIT-targeted fuzzer for the JS engine superfluous. In this regard we will measure how Fuzzilli perform on increasing code-coverage and compare the results with V8.

1.6 Results

Our fuzzing approach has proven effective in identifying numerous bugs across various components of GraalVM. We found:

- A total of 28 bugs were discovered through our fuzzing efforts targeting Graal.JS.
- These findings represent approximately 80% of all bugs reported to the GraalJS project during this period, underscoring the effectiveness of our approach.
- While primarily focused on GraalJS, our fuzzing technique also uncovered bugs in related components:
  - One bug was identified in the Truffle framework, highlighting the potential for our method to detect issues in the underlying infrastructure.
  - Another bug was found in the TRegex component, demonstrating the reach of our fuzzing approach beyond the core JavaScript engine.
- Our differential fuzzing strategy proved particularly valuable, leading to the discovery of 3 bugs that were only detectable when comparing GraalJS behavior against other JavaScript engines.

These results demonstrate the efficacy of our fuzzing approach in improving the robustness and correctness of GraalVM components, particularly GraalJS. The high proportion of bugs found through our method, compared to the total reported, indicates its significant contribution to the overall quality assurance process for GraalVM.

1.7 Core contributions

This thesis presents several key contributions to the field of fuzzing for GraalVM and its components. Our work has resulted in significant improvements to the Native Image compiler, enabling more efficient fuzzing through intrinsic support for coverage instrumentation. We have developed custom harnesses for both Jazzer and Fuzzilli, adapting these powerful fuzzing tools to work effectively with GraalVM's unique architecture.

A major focus of our work has been on JavaScript fuzzing, where we have developed a novel approach to differential fuzzing. This method allows us to compare the behavior of GraalJS against other JavaScript engines, uncovering subtle inconsistencies and potential specification violations. Our differential fuzzing harness is designed to work seamlessly with Fuzzilli, requiring no modifications to the underlying engines. This approach has proven particularly effective, leading to the discovery of several bugs that would have been difficult to detect through traditional fuzzing methods alone.

In addition to our work on JavaScript, we have also developed a fuzzer for GraalWasm, expanding our fuzzing capabilities to WebAssembly. This fuzzer, built using LibAFL, provides a modular and reusable execution harness that can be adapted for fuzzing other language implementations within GraalVM.

Our contributions extend beyond tool development to include methodological innovations in how we approach fuzzing polyglot runtimes. By leveraging GraalVM's unique architecture and capabilities, we have created fuzzing techniques that are particularly well-suited to detecting issues in language implementations and cross-language interactions. These methods have not only improved the robustness of GraalVM components but also demonstrated the potential for applying similar approaches to other polyglot or multi-language runtime environments.

Chapter 2

Background

2.1 GraalVM

GraalVM is a virtual machine and compiler designed to execute programs written in multiple programming languages. It's built on top of the Java Virtual Machine (JVM) and uses a Just-In-Time (JIT) compiler called Graal. Unlike traditional Java compilers that are typically written in C++, the Graal compiler is implemented in Java itself. This approach, known as a metacircular compiler, potentially allows for easier maintenance and development of the compiler. GraalVM aims to provide performance improvements over standard JVMs for certain workloads, particularly long-running server applications. It achieves this through aggressive optimizations and inlining across method boundaries. However, the actual performance benefits can vary depending on the specific application and use case. GraalVM also differs from standard JVMs in its support for running non-Java languages, which could potentially simplify polyglot application development and deployment.

2.1.1 Native Image

Native Image is a technology within GraalVM that allows for ahead-of-time compilation of Java applications into standalone executables. This process involves analyzing the application and its dependencies to determine which classes and methods are required, then compiling this subset of the application into native machine code. The resulting executable typically has faster startup times and lower memory footprint compared to traditionally deployed Java applications. However, Native Image has limitations, particularly with dynamic features of Java such as reflection, which may require additional configuration. It's often used for command-line tools, microservices, and serverless functions where quick startup and low resource usage are priorities.

2.1.2 Truffle

Truffle is a framework within GraalVM for implementing programming language interpreters. It provides a set of tools and APIs that allow language developers to create efficient interpreters by expressing the semantics of their language in a form that can be optimized by the Graal compiler. Truffle-based interpreters can potentially achieve performance comparable to purpose-built compilers, while requiring less development effort. This framework enables GraalVM to support a wide range of programming languages beyond Java, including JavaScript, Python, Ruby, and R. Truffle also facilitates interoperability between these languages, allowing developers to use libraries and functions from one language within programs written in another.

Languages interpreted by interpreters built using Truffle are referred to as guest languages while the runtime running the Truffle interpreter is reffered to as the host runtime. The most frequent configuration is to use Native Image as the host runtime as it allows to avoid the intensive warmup phase required to efficiently run a truffle interpreter.

2.2 Fuzzing

Fuzzing is a software testing technique used to discover coding errors and security loopholes in software, operating systems, or networks. It involves inputting massive amounts of random data into a system in an attempt to make it trigger some signal, called an oracle, when fuzzing to find memory corruption bugs it would be when the program crashes. The basic idea is to automatically generate unexpected or random data as inputs to a program, then monitor for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Fuzzing can be particularly effective at finding security vulnerabilities that might be exploited by malicious actors. There are various types of fuzzing, including mutation-based fuzzing (which modifies existing input samples), generation-based fuzzing (which creates inputs from scratch based on input models), and evolutionary fuzzing (which uses genetic algorithms to generate inputs). While fuzzing can be a powerful tool for identifying bugs and vulnerabilities, it's typically used in conjunction with other testing methods for comprehensive software quality assurance.

2.2.1 Coverage Feedback

Coverage feedback is a key technique in modern fuzzing that significantly enhances the effectiveness of the fuzzing process. This approach uses runtime information about which parts of the program are executed (code coverage) to guide the fuzzer towards exploring new execution paths. By instrumenting the target program, the fuzzer can track which code paths are exercised by each input. This information is then used to prioritize inputs that trigger previously unexplored code, focusing the fuzzing effort on areas more likely to uncover new bugs or vulnerabilities.

The implementation of coverage feedback involves adding instrumentation code to the target program, which updates a coverage map during execution. Common approaches include tracking edge coverage (transitions between basic blocks), basic block coverage, or more complex path coverage. This technique has proven highly effective in practice, allowing fuzzers to explore deeper program states and discover bugs more efficiently than purely random input generation. However, it also introduces challenges such as instrumentation overhead and the need for careful design to balance coverage granularity with performance.

2.2.2 Dataflow Feedback

Dataflow feedback, exemplified by techniques like cmplog (used in AFL++) and RedQueen, extends beyond simple coverage tracking to provide insight into how input data influences program behavior. These approaches focus on capturing and analyzing comparison operations within the target program, allowing the fuzzer to understand and overcome complex branching conditions more effectively. By instrumenting comparison instructions, the fuzzer can identify which parts of the input are being compared and what values lead to different execution paths. Consider two code snippets: (1) `if (a[0]==0xc0 && a[1]==0xff && a[2]==0xee)` versus (2) `if (memcmp(a, "\xc0\xff\xee", 3) == 0)`. While both perform the same logical check, the first presents multiple nested branches that are easier for traditional coverage-guided fuzzers to navigate incrementally. The second, using a single `memcmp`, appears as one branch to a coverage-guided fuzzer, potentially hiding the internal comparison from view. Dataflow feedback techniques can expose the internals of such comparisons, allowing the fuzzer to more easily derive the correct input bytes (`\xc0\xff\xee`) to satisfy the condition, regardless of how it's implemented. This capability significantly enhances the fuzzer's ability to explore paths guarded by complex comparisons, such as magic byte sequences or checksum validations.

2.2.3 Differential fuzzing

Differential fuzzing is a technique used to find implementation bugs by suppling the input to another implementation of the same specification in parallel with to the target, and compare the output of the executions. If they differ, then one of the implementations is not compliant. However, specifications often leave some behaviours to be implementation-defined. The role of the differential fuzzer is to distinguish between different inputs that arise from a bug in one of the implementation from an actual defect.

2.3 Fuzzilli

Fuzzilli[5] is an open-source JavaScript engine fuzzer primarily designed to find vulnerabilities in JIT (Just-In-Time) compilers. Fuzzilli uses a code generation approach to create complex JavaScript programs that can potentially trigger bugs in JavaScript engines. It employs a custom intermediate language to represent JavaScript programs, which allows for more precise and effective mutations during the fuzzing process. Fuzzilli uses coverage-guided fuzzing techniques to explore different code paths within the target engine. It has been successful in uncovering numerous vulnerabilities in major JavaScript engines, including V8 (used in Chrome), JavaScriptCore (used in Safari), and SpiderMonkey (used in Firefox). The tool is highly configurable and can be adapted to fuzz different JavaScript engines. While primarily focused on JIT compiler bugs, Fuzzilli can also potentially uncover other types of vulnerabilities in JavaScript engines.

2.4 LibAFL

LibAFL is an open-source, modular fuzzing framework written in Rust [3]. It aims to provide a flexible and extensible toolkit for building custom fuzzers. Unlike monolithic fuzzing tools, LibAFL allows researchers and practitioners to compose fuzzing pipelines from reusable components, enabling rapid prototyping and experimentation with novel fuzzing techniques. The core design philosophy of LibAFL centers around modularity and composability. Key fuzzing primitives such as input generation, mutation, and scheduling are implemented as interchangeable modules. This allows users to easily swap out components or introduce new ones to tailor the fuzzing process to specific targets or research goals.

Chapter 3

Design

3.1 Instrumentation

To facilitate effective fuzzing of GraalVM native images, we implemented an instrumentation framework that adheres to the LLVM sanitizer coverage (SanCov) interface. This choice was motivated by the widespread adoption of SanCov among contemporary fuzzing tools, ensuring broad compatibility with off-the-shelf fuzzers. The SanCov interface, originally developed for LLVM-based projects, has become a de facto standard in the fuzzing community due to its efficiency and flexibility in providing code coverage information.

By adopting the SanCov interface, our instrumentation framework seamlessly integrates with popular fuzzing engines such as American Fuzzy Lop (AFL), Honggfuzz, and LibFuzzer. This compatibility extends to more specialized tools like Fuzzilli, which is tailored for fuzzing JavaScript engines. The uniformity of the interface allows us to leverage the strengths of various fuzzing techniques without the need for custom adaptations for each tool. Furthermore, this approach enables us to benefit from ongoing developments in the fuzzing ecosystem, as improvements to these tools can be readily applied to our GraalVM native image fuzzing efforts with minimal additional engineering overhead.

We implemented the our instrumentation inside the native image compiler to be able to call directly into the fuzzer runtime library without going through JNI, it allows us to efficiently insert cmplog hooks as well as counters initialization. Moreover, as we need the trace pc guard interface for fuzzilli, we even more needed an efficient way to call into the compiler runtime as this requires to insert callbacks at every edge.

Our implementation also allows us to parially instrument some Java packages this is especially useful in Graal as the whole compiler and VM are written in Java, and can potentially be instrumented which would create an extra counter overhead of code for which we are actually not interested in increasing the coverage.

An additional benefit of the instrumentation in native image is that the compilation happens in a "closed world", every code that will be rachable at runtime is known at compile time, as opposed to the JVM which allows to load additional code at runtime, therefore we do not need to rely on dynamically allocating more counters, which makes an apporach such as jazzer work fine with libfuzzer but would break simpler fuzzer runtimes, such as fuzzilli which does not support multiple counters regions or resetting the counter map during the execution of the target.

Even though it would have been possible to use the inline 8 bit counters from Jazzer and adapt Fuzzilli to support it, however the goal is to continue to use Fuzzilli to fuzz GraalJS, which would imply an additional maintainance burden rather than having it buil-in the compiler, which might also be useful for other fuzzers in the future.

3.1.1 Input-to-State Correspondence

To enhance the effectiveness of our fuzzing approach, we incorporated dataflow feedback mechanisms inspired by techniques like RedQueen [10]. These methods aim to provide deeper insights into how input data influences program behavior, particularly in overcoming complex branching conditions.

Implementation Strategy

Our implementation adds cmplog instrumentation around conditional nodes in the intermediate representation (IR) of the program. Specifically, we insert hooks before `if` nodes in the control flow graph. This approach allows us to capture comparison operations that directly influence program flow.

While there is a theoretical risk that compiler optimizations might reorder these instrumentation points, our testing did not reveal this to be a significant issue in practice. The robustness of this approach is supported by the design of both AFL++'s cmplog and the original RedQueen technique, where the precise position of the instrumentation hook does not impact the algorithm's effectiveness.

Input-to-State Correspondence

The primary goal of cmplog is to establish a correspondence between input bytes and internal program state, particularly the operands of comparison operations. This correspondence is crucial for understanding how specific input patterns influence program behavior and for guiding the fuzzer towards exploring new execution paths.

The process of establishing this correspondence can be conceptualized as "coloring" both the input and the internal state exposed by the cmplog hooks. By tracking which input bytes contribute to which comparison operations, the fuzzer can more intelligently mutate inputs to trigger new behaviors. This approach is particularly effective in overcoming complex conditions such as magic byte sequences or checksum validations, which might otherwise pose significant challenges to coverage-guided fuzzers.

Memory compares

In GraalVM, memory comparison operations, particularly for strings and arrays, are typically implemented using assembly intrinsics for performance reasons. These intrinsics pose a challenge for instrumentation due to the compiler's architecture, which does not provide a straightforward mechanism for intercepting or modifying their behavior.

A naive approach might be to simply disable these intrinsics. However, this leads to unintended consequences. When intrinsics are disabled, the compiler falls back to generating standard comparison loops. These loops are then subject to various optimizations, including vectorization. Vectorization is an optimization technique that allows multiple data elements to be processed simultaneously using SIMD (Single Instruction, Multiple Data) instructions. While this can significantly improve performance, it creates a new problem for our instrumentation efforts: the sanitizer coverage (SanCov) interface does not provide hooks for SIMD operations.

To address this issue, we employ a more sophisticated strategy using invocation plugins. These plugins allow us to override the default implementations of methods like String.equals and Array.equals with our own custom versions. In our custom implementations, we append both the necessary instrumentation code and the original intrinsic comparison logic to the program graph. This approach ensures that we maintain the performance benefits of intrinsics while gaining the ability to instrument these critical comparison operations.

An additional advantage of our approach becomes apparent when handling length information for comparisons. Unlike in C, where the length of a string is not always readily available, Java's string and array objects inherently carry length information. We leverage this characteristic to provide more precise feedback to the fuzzer. By including accurate length data in our instrumentation, we enable the fuzzer to make more informed decisions about input mutations, potentially leading to more efficient exploration of the program's state space.

This comprehensive approach to instrumenting memory comparisons allows us to overcome the limitations imposed by both intrinsics and vectorization, while also taking advantage of language-specific features to enhance the quality of feedback provided to the fuzzer.

3.2 Fuzzilli Harness

To integrate GraalVM's JavaScript implementation with Fuzzilli, we designed a custom harness on top of the JavaScript launcher.

3.2.1 JavaScript Launcher

The JavaScript launcher serves as the entry point for executing JavaScript code within GraalVM. It is responsible for initializing the JavaScript context and configuring the runtime environment. Our custom launcher incorporates several key features:

3.2.2 REPRL Implementation

Fuzzilli employs a Read-Eval-Print-Reset-Loop (REPRL) model to efficiently execute and evaluate multiple test cases within a single process. This approach significantly reduces the overhead associated with process creation and initialization. Our REPRL implementation for GraalVM adheres to the following workflow:

1. Read: Accept JavaScript code input from Fuzzilli.
2. Evaluate: Execute the provided code within the GraalVM JavaScript context.
3. Print: Capture and return the execution results, including any errors or exceptions.
4. Reset: Clear the JavaScript context and reset code coverage counters.
5. Loop: Return to step 1 for the next test case.

This implementation ensures that each test case is executed in isolation while maintaining the performance benefits of a persistent process.

3.2.3 Fuzzilli-Specific Configurations

To fully integrate with Fuzzilli, we implemented several specific configurations and features:

- Coverage Instrumentation: We integrated Fuzzilli's coverage tracking mechanism into GraalVM, allowing the fuzzer to guide its exploration based on code coverage feedback.

- Custom Builtin Functions: We added Fuzzilli-specific builtin functions to the JavaScript environment, enabling controlled crash scenarios for testing purposes.
- Communication Protocol: Our harness implements Fuzzilli's expected communication protocol, facilitating seamless interaction between the fuzzer and the target JavaScript engine.

3.3 Differential fuzzing of Graal.js

To evaluate the effectiveness of fuzzing Graal.js and identify potential compliance issues or bugs, we designed a differential fuzzing setup comparing Graal.js against other major JavaScript engines. The key components of this design are:

Test case generation We utilize Fuzzilli as our primary test case generator due to its effectiveness in creating complex JavaScript programs that can exercise various parts of JavaScript engines. Fuzzilli's intermediate representation allows for targeted mutations that are more likely to trigger interesting behaviors in JIT compilers and interpreters.

We modify the Fuzzilli generator to serialize the last unused expression in each generated scope to JSON. This approach leverages Fuzzilli's design, where it maintains a list of pending expressions depending on previous values to be used for assignments. By default, at the end of a block, Fuzzilli emits these expressions as statements (i.e., not using their results). We target these expressions because:

- They potentially depend on the most behavior from previous parts of the program
- They incorporate the most previous expressions, therefore being the only one potentially covering the whole dataflow graph
- If the result was not being used, the JS engine might have optimized them out

Coverage Feedback In our differential fuzzing setup, we exclusively utilize coverage feedback from Graal.js to guide Fuzzilli's test case generation. This focused approach aligns with our primary objective of identifying bugs and compliance issues specifically within Graal.js. By concentrating on Graal.js's coverage, we maximize our exploration of its unique code paths and behaviors, increasing the likelihood of uncovering Graal.js-specific bugs or inconsistencies.

To ensure consistent and reproducible results across different JavaScript engines, we implement mechanisms to control non-deterministic aspects of the environment. This includes setting a fixed seed for pseudo-random number generators (PRNGs) and mocking date and time functions to return constant values. By standardizing these environmental factors, we eliminate potential sources of spurious differences between engine outputs that are not related to actual implementation discrepancies. This approach allows us to focus on genuine behavioral differences arising from the engines' implementations rather than external, variable factors.

Engine selection For our differential fuzzing targets, we selected V8 (used in Chrome) to compare against Graal.js during the differential fuzzing, because it it a wall established and popular JavaScript engine. We also used JavaScriptCore (used in Safari) and SpiderMonkey (used in Firefox) to further filter the buggy testcases.

Execution harness We developed a custom execution harness that can:

- Take a generated JavaScript test case as input
- Execute it on Graal.js, and V8 in parallel
- Capture and normalize the outputs and any runtime errors
- Compare the results across engines

Interpreter Consensus As a post-processing step, we implement an interpreter consensus mechanism to further validate potential bugs or compliance issues. When a discrepancy is detected between Graal.js and V8, we execute the test case on additional JavaScript engines, specifically JavaScriptCore and SpiderMonkey. This multi-engine comparison serves as a form of "interpreter voting," where the behavior exhibited by the majority of engines is considered the expected behavior. This approach helps distinguish between genuine compliance issues in Graal.js and cases where its behavior, while different from V8, aligns with other major implementations. This consensus mechanism reduces false positives and focuses our analysis on discrepancies that are more likely to represent actual bugs or significant deviations from common JavaScript engine behavior.

This differential fuzzing design allows us to systematically compare Graal.js against well-established JavaScript engines, helping to identify potential compliance issues, bugs, or areas where Graal.js's behavior diverges from other implementations. By leveraging Fuzzilli's generation capabilities and implementing a robust comparison framework, we can efficiently explore the JavaScript language space and focus on areas where Graal.js's partial evaluation approach may lead to unique behaviors or optimizations.

3.4 Jazzer/LibFuzzer Harness

Our implementation of the sanitizer coverage (SanCov) interface in GraalVM Native Image enabled us to create a function fuzzing interface comparable to Jazzer. This interface facilitates the fuzzing of libraries for which Jazzer fuzzing harnesses have already been developed, thereby leveraging existing work in the Java fuzzing ecosystem.

The harness operates by detecting the class that would typically serve as the main entry point for the application. However, instead of searching for a traditional "main" method, it looks for the Jazzer entry function. This function is then invoked from the LLVMFuzzerTestOneInput entry point, which is the standard entry point for LibFuzzer-compatible fuzzers. This approach allows for seamless integration with the LibFuzzer framework while maintaining compatibility with existing Jazzer harnesses.

To support the full range of LibFuzzer features, we implemented both 8-bit counter instrumentation and PC table generation. The 8-bit counters provide a compact representation of code coverage, allowing for efficient tracking of execution paths. The PC table, on the other hand, enables more detailed analysis of program behavior by mapping program counter values to specific locations in the source code.

A key component of our harness is the inclusion of the FuzzedDataProvider. This utility class, originally part of the Jazzer toolkit, facilitates the generation of typed test inputs from raw byte arrays. By incorporating this functionality, we ensure that our harness can handle complex input structures in a manner consistent with existing Jazzer harnesses.

One of the most significant advantages of our approach is the ability to directly utilize Open Source Security (OSS-Fuzz) harnesses for Java libraries. OSS-Fuzz is a continuous fuzzing service for open source software, and many Java projects have already developed Jazzer-compatible harnesses for this platform. By maintaining compatibility with these harnesses, we enable immediate fuzzing of a wide range of Java libraries without the need for extensive rewriting of existing fuzzing setups.

The process of fuzzing a Java library using our harness involves several steps. First, the harness initializes the Native Image JVM or attaches to a previously created VM, depending on the fuzzing configuration. It then loads the target class and locates the Jazzer entry function. For each fuzzing iteration, the harness generates input data using the FuzzedDataProvider, invokes the entry function with this data, and monitors for any crashes or unexpected behavior.

Our implementation also accounts for the nuances of the Java runtime environment. For instance, we ensure proper handling of exceptions, memory management, and thread safety, which are critical considerations when fuzzing Java applications. Additionally, we've implemented mechanisms to reset the application state between fuzzing iterations, preventing cross-iteration pollution that could lead to false positives or negatives.

By combining the power of Native Image compilation with the flexibility of the Jazzer interface, our harness provides a robust and efficient platform for fuzzing Java libraries. It bridges the gap between the native code fuzzing capabilities of LibFuzzer and the Java-specific features of Jazzer, offering a comprehensive solution for identifying vulnerabilities and bugs in Java applications compiled with GraalVM Native Image.

3.5 Wasm Fuzzing

We also implemented a simple fuzzer for GraalWASM, the WASM VM built on top of Truffle. We built it using LibAFL, a modular re-implementation of AFL++, it enables us to quickly assemble a fuzzer with a custom exection engine, input generator and mutators while retainning important parts such as the corpus queue and the coverage guidance.

The executor employs a multi-level execution strategy to efficiently fuzz WebAssembly modules within the GraalVM environment. It begins by forking the process, creating an isolated environment for each fuzzing session. Within this forked process, a Graal isolate is established, serving as the fundamental unit for executing the Truffle engine. The executor then enters a loop, creating a new WebAssembly context for each iteration. This context provides the necessary runtime environment for executing the WebAssembly module under test. For a set number of exections, the executor repeats the process by creating a new WebAssembly context, allowing for multiple executions within the same Graal isolate. Once the iteration limit is reached, the entire process is destroyed, and the cycle begins anew with a fresh fork. This approach balances the need for isolation between major fuzzing sessions with the efficiency of reusing the Graal isolate for multiple WebAssembly executions, optimizing the fuzzing process for both thoroughness and performance. In parallel, the parent exectuor observe the child for unwanted behaviour such as timeout and other signals, if it happens the parent destory the child executor and astarts over. The parent and child executor communicate in a similar way to what Fuzzilli does.

This execution model allows us to achieve functionality similar to AFL's fork server, but with finer-grained control over the execution environment. By creating a new WebAssembly context for each iteration within the same Graal isolate, we strike a balance between isolation and efficiency. This approach offers several advantages over a traditional fork server:

- **Reduced Overhead:** By reusing the Graal isolate across multiple executions, we minimize the time spent initializing the runtime environment.
- **Finer-grained Isolation:** Each WebAssembly context provides a clean slate for module execution, ensuring that state from previous iterations does not interfere with subsequent tests.
- **Improved Performance:** We avoid spawning processes for each executions as well as forking.

- **Modularity:** LibAFL provides a modular re-implementation of AFL++, allowing for flexible customization of fuzzing components.
- **Reusability:** The execution harness developed using LibAFL can be readily adapted for fuzzing other language implementations within GraalVM.
- **Direct Integration:** Our harness directly calls into the language shared library, eliminating the need for process spawning and reducing overhead.

3.5.1 Testcase Generation

We utilized the Bytecodealliance wasm-smith [13] tool to generate syntactically valid WebAssembly modules. This ensures that our initial corpus consists of well-formed Wasm binaries.

3.5.2 Testcase Muatation

We initially only used wasm-mutate [12] to mutate the modules, however this proved unsuccesful as it did not perform enough mutations, to counterbalance the very conservative approach to mutation we also decided to use it in conjuction with the default unrestrircted byte-level mutator of AFL. We balance them using a round-robin approach.

To avoid ending up with a fully invalid corpus, inputs that fail to increase coverage or result in a syntax error are automatically discarded, ensuring that the fuzzer focuses on productive paths.

3.5.3 Integration with other languages

Our Wasm fuzzer design takes advantage of GraalVM Native Image to directly call the interpreter, built as a shared library, instead of having to deal with processes. All the language interpreter are built in a similar fashion, therefore our executor could be reused to fuzz other languages.

Chapter 4

Implementation

4.1 Instrumentation

The instrumentation framework developed for this project underwent several iterations to achieve optimal performance and compatibility with existing fuzzing tools. Initially, we leveraged the profiling infrastructure of Profile-Guided Optimization (PGO) within GraalVM, adapting it to utilize 32-bit counters compatible with the PC guard mechanism employed by sanitizer coverage whereas it initially used 64-bit counters. As the project progressed, we refactored this initial implementation into a common instrumentation framework shared with PGO. This modular design allowed for straightforward implementation of additional instrumentation types, such as the 8-bit counters required for LibFuzzer compatibility. The framework inserts counters or guards at every edge of the control flow graph during the native image build process.

4.1.1 Native Implementation

We made a deliberate decision to implement the instrumentation within the native image-specific components of the compiler. This approach enables direct invocation of external library functions, as mandated by sanitizer coverage, without the overhead and complexity introduced by Java Native Interface (JNI) calls. Consequently, this design choice yields improved performance and simplified integration with existing fuzzing tools.

4.1.2 Dataflow Feedback

To achieve parity with the dataflow feedback mechanisms employed by state-of-the-art fuzzers like AFL and LibFuzzer, our compiler phase also instruments memory and array comparison operations.

While these fuzzers typically accomplish this by intercepting calls to libc functions such as `memcmp` and `strcmp`, the Java ecosystem necessitates a different approach. In our implementation, we focus on instrumenting the `String.equals` and `Array.equals` methods, which serve as the Java equivalents to the aforementioned libc functions. These methods are commonly implemented in GraalVM as hand-written intrinsics for performance reasons. Simply disabling these intrinsics proved suboptimal, as it led to vectorized comparisons that eluded instrumentation. To address this challenge, we developed custom intrinsics that not only maintain the original functionality but also invoke the appropriate AFL memory comparison instrumentation hooks (`__cmplog_rtn_hook_n` and `__cmplog_rtn_hook_strn`). This modular approach to intrinsic replacement facilitates future extensions, such as the incorporation of analogous LibFuzzer hooks, with minimal additional effort. This comprehensive instrumentation strategy ensures that our framework captures both control flow and data-dependent behavior, providing fuzzing tools with rich feedback to guide their exploration of the target program's state space.

4.1.3 Floating points handling

A key consideration in implementing the cmplog instrumentation was handling floating point comparisons correctly. On x86 architectures, floating point values are typically passed in XMM registers rather than general purpose registers. However, the cmplog hooks expect integer arguments. To address this, we ensure correct register allocation and parameter passin by casting the floating point values to their integer representations before passing them to the cmplog hook. This approach aligns with AFL++'s implementation of cmplog instrumentation, which performs a similar conversion. While LLVM's SanitizerCoverage does not perform this float-to-int conversion, we chose to follow AFL++'s model as it is our primary target for cmplog integration. This design decision ensures compatibility with AFL++ while still allowing integration with other fuzzers that use the SanitizerCoverage interface.

4.2 Differential Fuzzing

The differential fuzzer harness was implemented as an autonomous intermediary component that mediates between the fuzzer and the two JavaScript engines under test. This harness serves as a bidirectional proxy, implementing both the client and server sides of the Fuzzilli protocol to facilitate seamless communication between the fuzzer and the fuzz targets (See Fig 4.1). Upon initialization, the harness executes a series of preparatory steps: it launches the fuzz targets (Graal.js and V8), establishes the necessary communication channels, and enters a wait state until both engines signal their readiness. Only after confirming the operational status of both targets does the harness notify the fuzzer of its preparedness to commence the fuzzing process. During active fuzzing, the harness receives JavaScript inputs from the fuzzer and simultaneously dispatches these inputs to both Graal.js and V8 for execution. This architecture enables the harness to efficiently manage the parallel execution of test cases and collect results from both engines, facilitating real-time differential analysis of their behaviors.

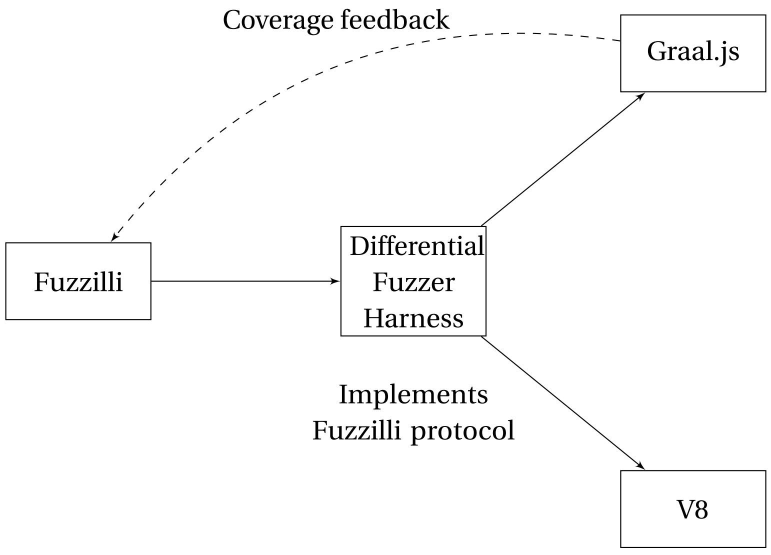


Figure 4.1: Differential Fuzzer Harness Architecture with Coverage Feedback

4.2.1 Comparing the outputs

We compare the output line by line, whenever we encounter a line that does not match its counterpart, we fallback to the slower comparaison mode, in which both line are scanned for floating point values using a regular expression. If we match the same number of floating points in both lines we compare them pairwise with a relative comparison for a set ϵ. We only consider both result to be the same if all floating points and text similar enough.

$$abs(\frac{a-b}{b}) < \epsilon$$

4.3 WASM fuzzer harness

To create an efficient execution environment for fuzzing the WebAssembly interpreter, we leveraged the native image polyglot API provided by GraalVM. This approach allowed us to minimize the overhead typically associated with forking processes for each test case execution. The implementation required extending the existing native image polyglot API, which previously only supported string inputs. As WebAssembly bytecode can contain invalid bytes sequences in the charset of the string,

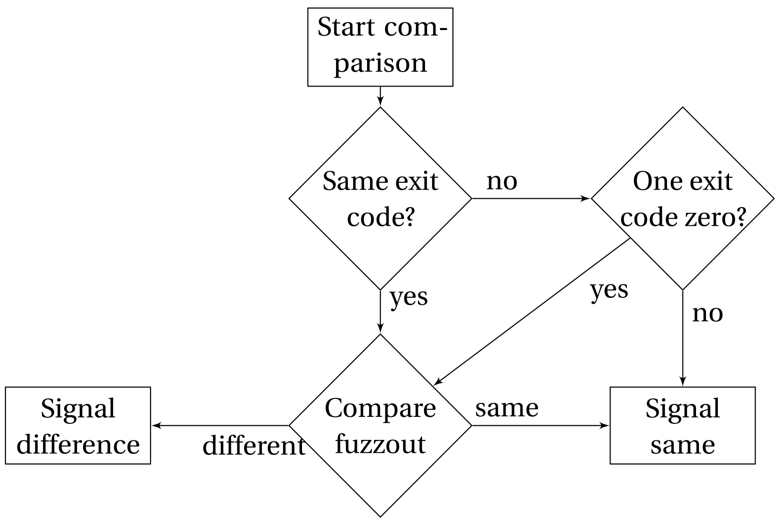


Figure 4.2: Differential Fuzzer Output Comparison Flowchart

the standard Java String representation was insufficient. Therefore, we introduced a new entry point in the API to accommodate arbitrary byte sequences as input.

This extension enables the direct passing of WebAssembly bytecode to the interpreter without the need for intermediate encoding or decoding steps. By avoiding the creation of new processes for each test case, and by enabling direct bytecode input, our harness achieves improved performance in the fuzzing loop.

Chapter 5

Evaluation

5.1 Performance benchmark

We measured the performance of our implementation using the Renaissance benchmarking suite [9], the suite was run on a . Our implementation supports both the TracePCGuard and Inline8Bit instrumentation modes, with the latter demonstrating notably superior performance (see Fig. 5.1). This disparity can be attributed to the fundamental differences in their mechanisms:

- **Inline8Bit**: This mode directly increments 8-bit counters inline at each instrumentation point. By avoiding function calls and operating directly on counters, it minimizes execution overhead.
- **TracePCGuard**: In contrast, this mode inserts calls to an external function at each instrumentation point. While these calls provide flexibility, they introduce additional overhead due to function call mechanics and potential cache misses.

It is important to note that in typical LLVM-based implementations, the performance gap between these modes is often mitigated through Link Time Optimization (LTO). LTO allows the compiler to inline the guard functions, effectively reducing the TracePCGuard overhead to levels comparable with Inline8Bit.

In the context of GraalVM Native Image, we currently lack an equivalent to LLVM's LTO phase. This absence amplifies the performance difference between the two modes in our implementation. However, this also presents an opportunity for future optimization. Implementing an LTO-like phase for Native Image could potentially equalize the performance of these instrumentation modes, offering users the flexibility of TracePCGuard without significant performance penalties.

For optimal performance in the current implementation, we recommend using the Inline8Bit mode when compatible with the chosen fuzzing tool. For tools requiring TracePCGuard, users should be aware of the potential performance impact. Future work on Native Image optimization could further refine these trade-offs, potentially offering the best of both worlds in terms of flexibility and performance.

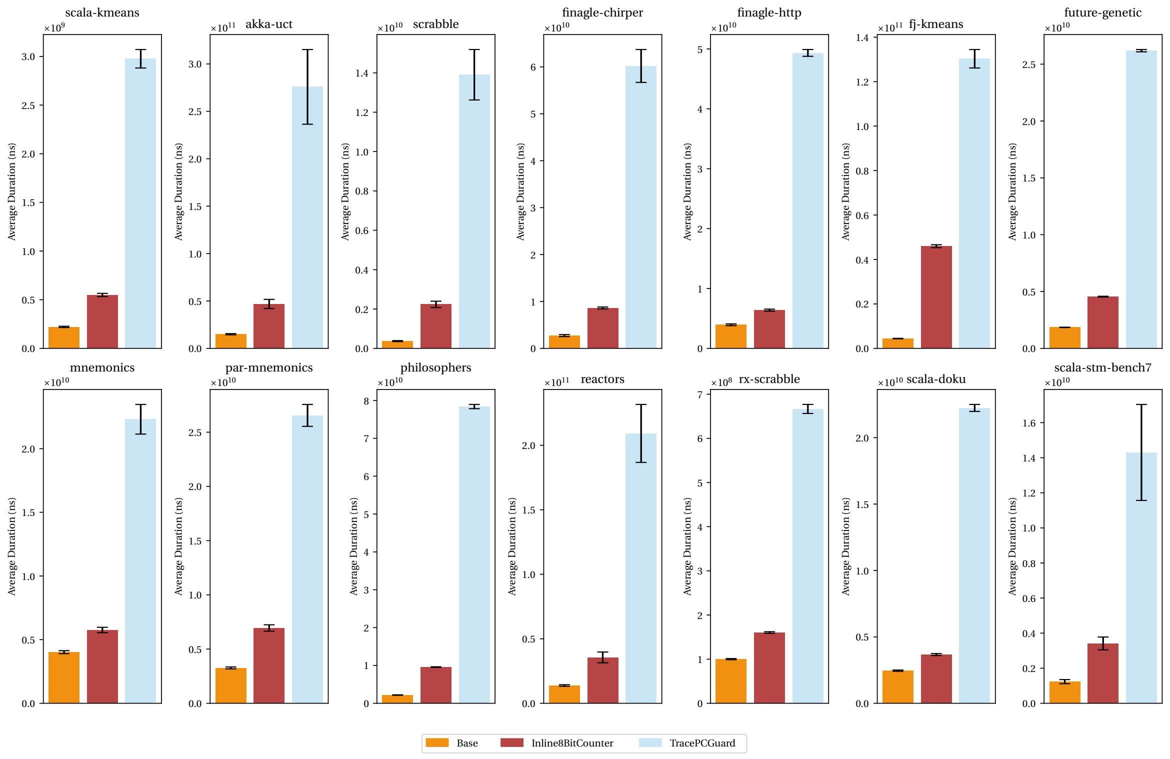


Figure 5.1: Renaissance benchmark, without instrumentation, Inline8Bit and PCGuard, 50 iterations

5.1.1 Effectiveness of Cmplog Instrumentation

Our evaluation of the cmplog instrumentation revealed a significant improvement in the fuzzer's ability to discover program constants. With cmplog enabled, AFL consistently identified target constants in under 10 seconds. In contrast, when limited to only coverage instrumentation, the fuzzer was unable to discover these constants within the allotted time frame.

5.2 Comparison with Jazzer

An attempt was made to conduct a comparative analysis between our implementation and Jazzer, a widely-used Java fuzzing tool. The evaluation aimed to assess the relative performance and effectiveness of our approach against Jazzer across different runtime environments.

5.2.1 Experimental Setup

The comparison utilized targets from the OSS-Fuzz project, a comprehensive collection of open-source software subjected to continuous fuzzing. Three configurations were evaluated:

1. Jazzer executing on the Java Virtual Machine (OpenJDK 22.0.2)
2. Jazzer running on GraalVM
3. Jazzer operating on a native image compiled with GraalVM

To establish a baseline for comparison, the evaluation focused solely on code coverage metrics as an initial performance indicator. The instrumentation was limited to specific prefixes to mitigate the potential performance overhead incurred by GraalVM's comprehensive engine instrumentation. Each configuration underwent continuous fuzzing for a duration of 24 hours.

5.2.2 Challenges and Limitations

Despite the rigorous setup, the comparative analysis encountered significant obstacles that impeded a comprehensive evaluation. Many of the most relevant examples from the OSS-Fuzz project required Java features, such as reflection or dynamic class loading, which necessitated additional configuration for native image compilation. This requirement introduced complexity and potential inconsistencies in the comparison. Of the two targets that were successfully executed for the full duration, one (org.apache.collections) failed to produce any coverage data when fuzzed with Jazzer, for reasons that remain unclear at the time of writing. The other target (xz-java) exhibited rapid coverage saturation, plateauing after less than an hour of fuzzing across all configurations.

5.2.3 Preliminary Observations

While the experimental results were insufficient to draw definitive conclusions, preliminary observations suggested that the performance of our implementation was comparable to that of Jazzer in terms of fuzzing throughput. However, the lack of consistent, long-term coverage growth across the evaluated targets prevents a more detailed comparative analysis of the approaches' effectiveness in exploring the target programs' state spaces.

5.3 JavaScript Engine Fuzzing

To evaluate the effectiveness of our fuzzing approach, we conducted an extensive campaign targeting GraalJS. For comparison, we also performed fuzzing on the V8 JavaScript engine to establish a baseline. The non-differential fuzzing experiments were executed five times, each lasting over 24 hours.

5.3.1 Experimental Setup

The experiments were conducted on an Oracle Cloud virtual machine with the following specifications:

- Compute: 50 OCPU (100 vCPU)
- Machine Type: VM.Standard.E3.Flex
- Processor: AMD EPYC 7742
- RAM: 256 GB

This configuration was chosen to closely align with the experimental setup used by Groß et al. in their evaluation of Fuzzilli [5].

5.3.2 Results

Our analysis revealed that the coverage increase pattern for GraalJS closely resembles that of V8, as illustrated in Figure 5.2. Similarly, the relative contributions of different mutation operators to coverage growth showed comparable trends between the two engines, as depicted in Figure 5.3.

These results suggest that our fuzzing approach for GraalJS achieves performance comparable to established techniques used for V8, indicating its effectiveness in exploring the JavaScript engine's state space and uncovering potential vulnerabilities.

Figure 5.4 presents a treemap visualization of the code coverage achieved in GraalVM after a 36-hour fuzzing session. This visualization offers valuable insights into the effectiveness of our fuzzing approach and highlights areas for potential improvement. The JavaScript engine, represented by the

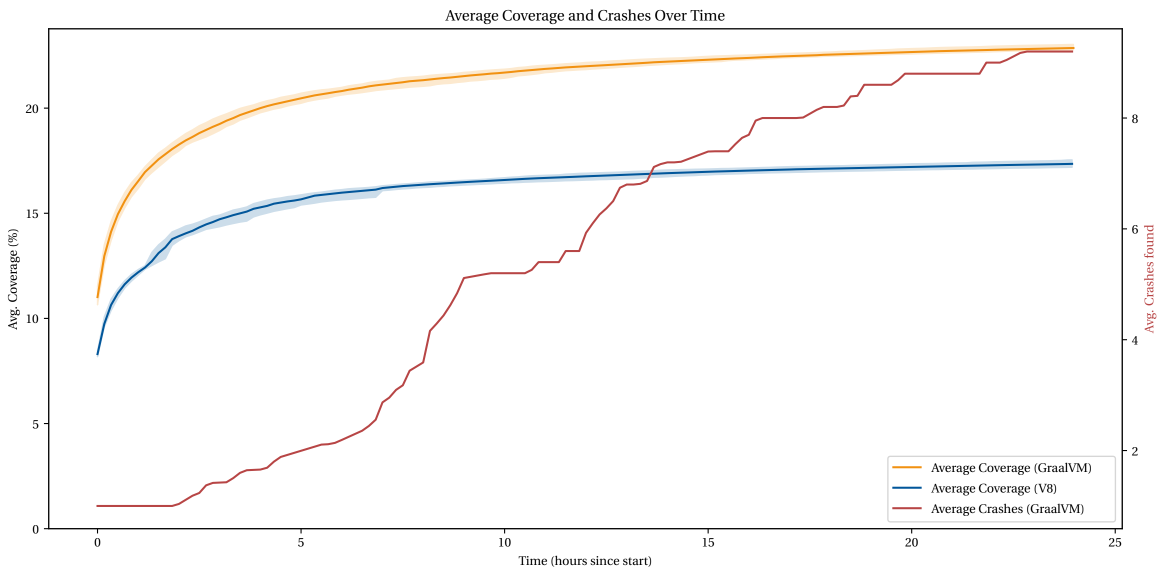
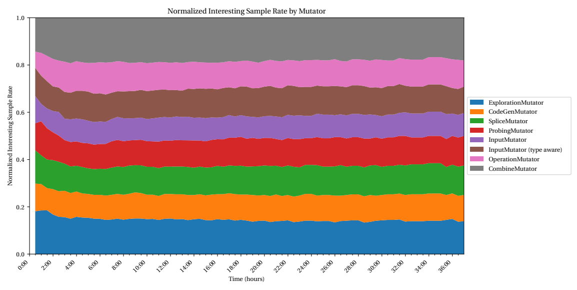


Figure 5.2: Average coverage increase on GraalJS and V8 with minimum and maximum over 5 runs



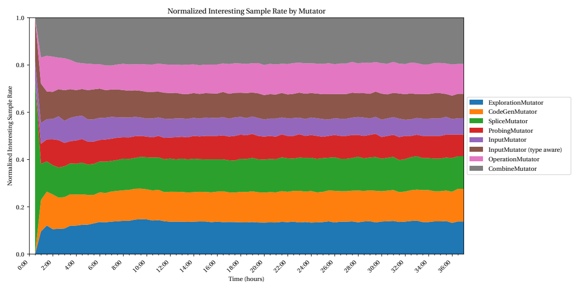


Figure 5.3: Fuzzilli mutators participaton for GraalJS and V8

purple region on the left, demonstrates relatively uniform coverage across its components. However, notable exceptions are evident, particularly in the areas related to development tools such as debuggers, the Debug Adapter Protocol (DAP), and the Chrome Inspector interface. Additionally, we observe a significant gap in coverage for the JavaScript Temporal API, indicated by the conspicuous "hole" in the corresponding region. This observation suggests that future fuzzing efforts should explicitly enable and target the Temporal API to ensure comprehensive coverage, as the API is still experimental it was disabled by default. Furthermore, the visualization reveals that eventhough we targeted the JS engines, the compiler and the Truffle engine were well exercised as well.

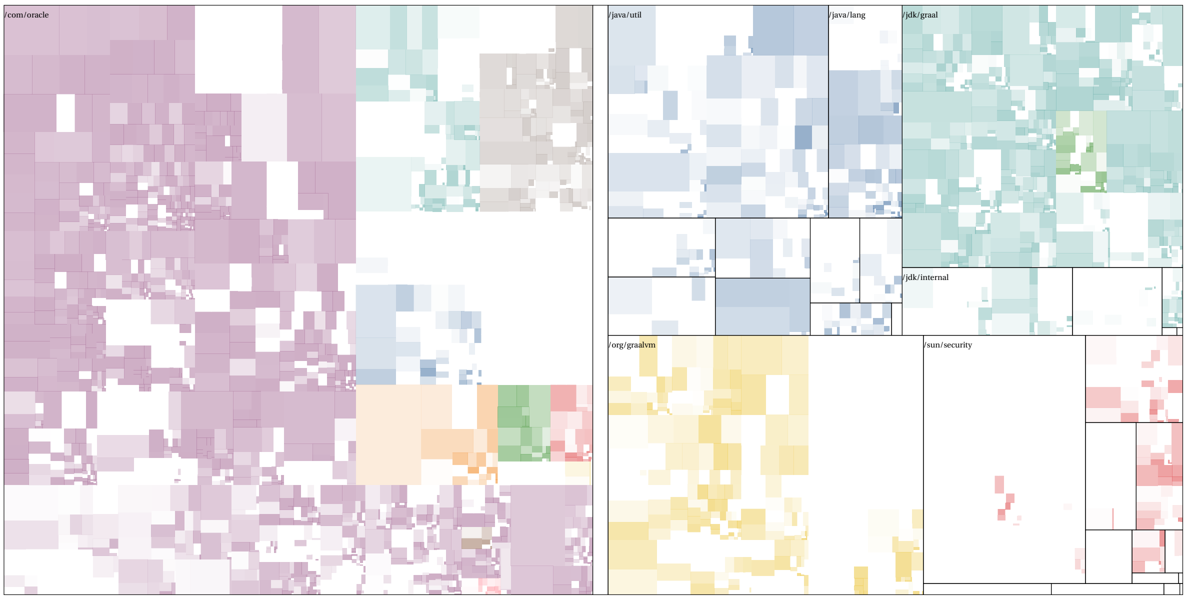


Figure 5.4: Code coverage of the final corpus after a 36h run

5.3.3 Differential

The differential fuzzer found 3 bugs in the GraalJS engine, all of which have since then been fixed. An interesting example, shows how GraalJS used to over-approximate inverse hyperbolic functions (See Fig. 5.5).

`Math.tan(1073741824)`

V8: -0.784696613319200**5**
GraalJS: -0.784696613319200**4**
$\rightarrow abs(\frac{a-b}{b}) < \epsilon$

`Math.acosh(127 ** 127)`

V8: 615.904907160801
GraalJS: Infinity
$\rightarrow abs(\frac{a-b}{b}) \not< \epsilon$

Figure 5.5: On the left an example where both implementation differ however it is in an acceptable margin, on the right an actual bug where GraalJS over-apporximated

However even the best comparator could still lead to some false positives, for instance the example in Fig. 5.6, all popular JS engines insert a fast path, depite being not conformant to the specification. This erroneous behaviour has been acknowledged by V8 developers and will not be fixed as it is considered "web-reality". [1]

Our evaluation primarily focused on maximizing coverage for GraalJS, which may have limited the breadth of our differential analysis. A more comprehensive evaluation could be achieved by utilizing V8 as the primary target for coverage-guided fuzzing. This approach would likely yield a more balanced exploration of the JavaScript language space, potentially uncovering additional discrepancies across implementations.

```
const v1 = [1];
function F2(){}
F2.defineProperty = 0;
const v10 = new Proxy(v1, F2);
console.log(v10.sort());
```

Figure 5.6: A snippet where all JS engines except GraalJS insert a fast path

Furthermore, our current methodology prioritizes GraalJS's coverage information to guide the fuzzing process. While this approach is effective for identifying issues specific to GraalJS, it may inadvertently bias the test case generation towards GraalJS's implementation details. To address this limitation, future work could incorporate coverage feedback from multiple engines simultaneously, allowing for a more holistic exploration of the JavaScript specification.

It is worth noting that although our primary objective was to identify bugs in GraalJS, the differential fuzzing framework we developed is inherently capable of detecting issues in any of the compared engines. By adjusting the coverage guidance and analysis priorities, this approach could be readily adapted to target other JavaScript implementations, potentially revealing bugs or specification deviations in engines like V8, SpiderMonkey, or JavaScriptCore.

Chapter 6

Related Work

6.1 Jazzer

Jazzer [7] is an open-source coverage-guided fuzzing tool designed specifically for Java applications. Developed by Code Intelligence, Jazzer has gained significant traction in the field of Java security testing, notably being adopted by Google's OSS-Fuzz project for continuous fuzzing of popular Java libraries.

The key features and characteristics of Jazzer include:

- Java Native Interface (JNI) instrumentation: Jazzer inserts JNI for cmplog data feedback.
- Integration with existing fuzzing engines: Jazzer acts as a bridge between Java applications and established fuzzing engines like libFuzzer.
- Compatibility with sanitizers: It supports various sanitizers to detect issues beyond crashes, such as memory leaks or undefined behavior.

Jazzer's architecture consists of two main components:

1. A driver that initializes the JVM and manages the fuzzing process.
2. An agent that performs runtime instrumentation of Java bytecode.

The instrumentation process involves modifying the bytecode to insert counter increments, which are then accessed by the fuzzing engine through raw memory access.

While Jazzer has proven effective for fuzzing Java applications, it faces certain limitations and challenges:

- Performance overhead: The use of JNI calls for instrumentation can introduce significant runtime overhead.
- Maintenance challenges: The open-source edition of Jazzer has faced maintenance issues, potentially impacting its long-term viability and community support.
- Limited compiler integration: As an external tool, Jazzer lacks deep integration with Java compilers, which could potentially offer more efficient instrumentation techniques.

In contrast, our approach implements instrumentation directly within the Native Image compiler, offering several potential advantages:

- Improved performance: By integrating at the compiler level, we can potentially reduce the overhead associated with JNI calls.
- Enhanced flexibility: Direct compiler integration allows for more fine-grained control over the instrumentation process.
- Specialized instrumentation: We can implement efficient `TracePCGuard` functionality, which is crucial for integration with fuzzers like Fuzzilli.

Morevoer GraalVM components such as Truffle interpreters are almost exclusively built using native image, and in order to guarentee the effectiveness of our fuzz testing we need to run the targets on an environment as close as possible to the environment it will actually be used in, which also justify our choice to implement the instrumentation directly in the native image compiler.

While Jazzer has set an important precedent in Java fuzzing, our work aims to build upon its foundation by leveraging the unique capabilities of GraalVM and Native Image to provide more efficient and effective fuzzing for Java applications and polyglot runtimes.

6.2 SunnyMilkFuzzer

SunnyMilkFuzzer [11] is a recent optimization of fuzzing for Java Virtual Machine (JVM) languages, primarily addressing performance bottlenecks in Jazzer, the current industry-leading Java fuzzing tool. A key innovation of SunnyMilkFuzzer is its intrinsic support for edge coverage probes:

- The JVM is modified to provide low-level support for coverage instrumentation, significantly reducing the overhead associated with coverage tracking.
- This is achieved by implementing a rewriting rule in the JIT compiler to detect edge coverage probes and rewrite them to optimal native code during the native code generation phase.

- The implementation replaces the original probe, which involved a native method call, with a direct memory increment instruction, eliminating the overhead of method invocation.

However, it's important to note a significant limitation of the current implementation:

- The intrinsic probe support is implemented only for the C1 compiler, which is the first-tier, quick compiler in the JVM's multi-tiered compilation system.
- The C1 compiler, while faster to compile, produces less optimized code compared to the C2 compiler, which is the second-tier, highly optimizing compiler.
- This limitation means that SunnyMilkFuzzer does not fully leverage the potential performance benefits that could be achieved if the intrinsic probe support were implemented for the C2 compiler as well.

Despite this limitation, SunnyMilkFuzzer still demonstrates substantial performance improvements over Jazzer, with an average throughput increase of 138% and peak improvements of up to 441% across a range of Java programs from Google's OSS-Fuzz project. These results suggest that even with the C1 compiler limitation, the intrinsic probe support provides significant benefits. The authors acknowledge this limitation and suggest that implementing intrinsic probe support for the C2 compiler could potentially yield even greater performance improvements. This represents an important area for future work, as it could further enhance the efficiency of Java-based fuzzing tools.

Our approach of doing it in the native image compiler allows us to target the highest-tier compilation level, however it is to be noted that GraalVM Native Image is still often somewhat slower than running the compiler in JIT mode, due to the lack of profiling information available to the compiler.

6.3 JIT-Picking

JIT-Picking [2] is a differential fuzzing technique designed to find bugs in JavaScript engines by comparing the behavior of code across different optimization levels within the same engine. This approach targets the just-in-time (JIT) compilation process, which is a common source of vulnerabilities in JavaScript engines.

The key insight of JIT-Picking is to exploit the fact that modern JavaScript engines typically employ multiple tiers of optimization. By executing the same code snippet at different optimization levels and comparing the results, JIT-Picking can potentially identify inconsistencies that may indicate bugs in the JIT compiler or optimization passes.

While JIT-Picking shares some similarities with the differential fuzzing approach used in this work, there are several important distinctions:

- JIT-Picking focuses on intra-engine comparisons across optimization tiers, whereas our approach compares different JavaScript engine implementations (e.g., GraalJS vs. V8).
- The optimizations in GraalJS, are achieved by leveraging Truffle and Graal, therefore we are mostly looking for implementation bug in the engine, because we can fuzz the compiler separately at a lower level. This makes the JIT-Picking approach less relevant for GraalJS.
- JIT-Picking may not be as effective at identifying bugs in built-in functions, as these are often implemented uniformly across optimization levels within a single engine.

Despite these differences, JIT-Picking demonstrates the value of differential analysis in identifying JavaScript engine bugs, particularly those related to JIT compilation. Our work extends this concept to cross-engine comparisons, which can potentially uncover a broader range of implementation discrepancies and specification compliance issues.

Chapter 7

Conclusion

This thesis has presented a comprehensive approach to fuzzing the GraalVM ecosystem, with a particular focus on integrating fuzzing capabilities directly into the Native Image compiler. The key contributions of this work include:

- Implementation of sanitizer coverage instrumentation within the Native Image compiler, enabling efficient fuzzing of GraalVM components.
- Development of custom harnesses for Jazzer and Fuzzilli, adapting powerful fuzzing tools to work effectively with GraalVM architecture.
- Creation of a novel differential fuzzing approach for JavaScript engines, allowing for systematic comparison of GraalJS against other major implementations.
- Implementation of a WebAssembly fuzzer using LibAFL, demonstrating the extensibility of our approach to other language implementations within GraalVM.

Our evaluation demonstrated the effectiveness of these approaches, particularly in fuzzing GraalJS. The fuzzing campaign identified 28 bugs, representing approximately 80% of all bugs reported to the GraalJS project during the evaluation period. This high proportion of discovered bugs underscores the value of our fuzzing methodology in improving the robustness and correctness of GraalVM components.

The performance analysis of our instrumentation implementation revealed that the Inline8Bit mode offers superior performance compared to TracePCGuard, due to the current lack of Link Time Optimization in Native Image. This finding highlights an area for potential future optimization in the GraalVM compilation process.

While our comparative analysis with Jazzer faced challenges due to compatibility issues with some OSS-Fuzz targets, preliminary observations suggested comparable performance between our implementation and Jazzer in terms of fuzzing throughput. However, further research is needed to draw definitive conclusions about relative effectiveness in exploring target programs' state spaces.

The differential fuzzing approach proved particularly valuable, leading to the discovery of three bugs that were only detectable when comparing GraalJS behavior against other JavaScript engines. This demonstrates the power of cross-implementation comparison in identifying subtle specification deviations and potential compliance issues.

Future work could focus on several promising directions:

Firstly, implementing Link Time Optimization for Native Image could potentially equalize the performance of different instrumentation modes. This optimization would address the current performance disparity between Inline8Bit and TracePCGuard modes, potentially offering improved efficiency across all instrumentation types.

Secondly, extending the differential fuzzing approach to incorporate coverage feedback from multiple engines simultaneously could allow for a more holistic exploration of language specifications. This multi-engine feedback mechanism could provide deeper insights into cross-implementation discrepancies and enhance the overall effectiveness of the fuzzing process.

Lastly, adapting the fuzzing methodology to target other language implementations within the GraalVM ecosystem, such as Python or Ruby, would broaden the scope and impact of this work. Such extensions could contribute to improved robustness and correctness across the entire GraalVM platform.

In conclusion, this work has helped advance the state of fuzzing for GraalVM and its components, providing a robust foundation for continued improvement in the security and reliability of the Graal ecosystem. The methodologies and tools developed here such as differential fuzzing could also have applications beyond GraalJS and help find bugs and inconsistencies in other JavaScript implementations.

Bibliography

[1] *Array.Prototype.Sort Fast Track for Length == 1 Bypasses, Has, Get, Set, Delete [42201594] - Chromium.* URL: `https://issues.chromium.org/issues/42201594` (visited on 08/15/2024).

[2] 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.* CCS '22. New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 351–364. ISBN: 978-1-4503-9450-5. DOI: `10.1145/3548606.3560624`. URL: `https://dl.acm.org/doi/10.1145/3548606.3560624` (visited on 08/07/2024).

[3] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS).* Ccs '22. Los Angeles, U.S.A.: ACM, Nov. 2022.

[4] *Go Fuzzing - The Go Programming Language.* URL: `https://go.dev/doc/security/fuzz/` (visited on 08/16/2024).

[5] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. "FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities". In: *Proceedings 2023 Network and Distributed System Security Symposium.* Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society, 2023. ISBN: 978-1-891562-83-9. DOI: `10.14722/ndss.2023.24290`. URL: `https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f290_paper.pdf` (visited on 08/06/2024).

[6] *Introduction - Rust Fuzz Book.* URL: `https://rust-fuzz.github.io/book/introduction.html` (visited on 08/16/2024).

[7] *Java Fuzzing with Jazzer [Complete Guide].* URL: `https://www.code-intelligence.com/blog/java-fuzzing-with-jazzer` (visited on 08/01/2024).

[8] *libFuzzer – a Library for Coverage-Guided Fuzz Testing. — LLVM 20.0.0git Documentation.* URL: `https://llvm.org/docs/LibFuzzer.html` (visited on 08/16/2024).

[9] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. “Renaissance: Benchmarking Suite for Parallel Applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 8, 2019, pp. 31–47. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314637. URL: https://doi.org/10.1145/3314221.3314637 (visited on 08/15/2024).

[10] *REDQUEEN: Fuzzing with Input-to-State Correspondence*. NDSS Symposium. URL: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/ (visited on 08/15/2024).

[11] Junyang Shao. “SUNNYMILKFUZZER - AN OPTIMIZED FUZZER FOR JVM-BASED LANGUAGE”. thesis. Purdue University Graduate School, July 27, 2023. DOI: 10.25394/PGS.23750652.v1. URL: https://hammer.purdue.edu/articles/thesis/SUNNYMILKFUZZER_-_AN_OPTIMIZED_FUZZER_FOR_JVM-BASED_LANGUAGE/23750652/1 (visited on 07/01/2024).

[12] *Wasm_mutate - Rust*. URL: https://docs.rs/wasm-mutate/latest/wasm_mutate/ (visited on 08/15/2024).

[13] *Wasm_smith - Rust*. URL: https://docs.rs/wasm-smith/latest/wasm_smith/ (visited on 08/15/2024).