Automated Fuzz Driver Generation for Java via Search-Based Testing: A Preliminary Study

Leo Fernandes[1], **Juvenal Carvalho**[2], **André Catarino**[2]
José Campos[2,4], **Elvys Soares**[1], **Rui Abreu**[2,3]

[1]Instituto Federal de Educação, Ciência e Tecnologia de Alagoas (IFAL), Brazil

[2]Faculdade de Engenharia, Universidade do Porto, Portugal

[3]INESC-ID, Portugal

[4]LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

{leonardo.fernandes, elvys.soares}@ifal.edu.br, jovycarvalho@gmail.com

{up202408593, jcmc, rma}@fe.up.pt

Abstract. *Coverage-guided fuzzing has proven to be remarkably effective in uncovering security and reliability issues across large software systems. However, its adoption in practice remains hindered by the need for manually crafted fuzz drivers, whose construction is labor-intensive, error-prone, and often unrealistic for open-source software, where thousands of libraries expose complex and interdependent APIs. In this paper, we present* arrabida*, a novel approach to automating the generation of fuzz drivers for Java through encapsulated controlled execution contexts and orchestrated use of APIs, such as unit test cases.* arrabida *integrates search-based test generation (via EvoSuite) with static analysis, transforming the synthesized tests into Jazzer-compatible fuzz drivers, which are compilable and deployable in OSS-Fuzz. We evaluated* arrabida *on three real-world Java projects from OSS-Fuzz, by comparing generated drivers with their manually written counterparts in terms of iteration throughput, coverage, feature exploration, and crash detection. Our results show that* arrabida *drivers achieve comparable results to manually written drivers, exceeding coverage and number of iterations, while discovering unique program features and behaviors.* arrabida *triggered 11 unique crashes.*

1. Introduction

Coverage-guided fuzzing is highly effective in discovering security-critical bugs and has uncovered tens of thousands of vulnerabilities in widely deployed systems [Manès et al. 2021, Böhme et al. 2021, Zhu et al. 2022]. However, its effectiveness drops when applied to *software libraries*, which lack natural entry points and require a *fuzz driver* to map raw inputs to structured API invocations. The success of a fuzzing campaign therefore depends on the quality of this driver, which must encode valid API usage, meaningful call sequences, and proper initialization logic [Babić et al. 2019, Chen et al. 2023, Jeong et al. 2023, Zhang et al. 2021, Zhang et al. 2024]. Inadequate drivers leave large portions of the attack surface unexplored, limiting vulnerability discovery [Zhang et al. 2024, Jeong et al. 2023].

Despite their importance, fuzz drivers are still predominantly written *manually*, making them costly to develop and maintain. This process requires expertise in both the target codebase and the fuzzing engine, and becomes increasingly challenging as libraries evolve [Serebryany 2017, Jeong et al. 2023, Chen et al. 2023]. As a result, driver scarcity has emerged as a key bottleneck in scalable fuzzing. For instance, OSS-Fuzz reports average coverage around 30% across more than 1,300 projects [Team 2023], with many stagnation points attributed to inadequate drivers rather than limitations of input generation [Gao et al. 2023].

Existing approaches only partially address this problem. Code-mining techniques [Babić et al. 2019, Ispoglou et al. 2020] depend on client code and often cover limited API subsets, while LLM-based synthesis [Lyu et al. 2024] introduces nondeterminism and prompt sensitivity, limiting reproducibility in security-critical settings.

In this work, we present *arrabida*, an automated approach for generating fuzz drivers for Java libraries. *arrabida* combines static analysis with search-based unit test generation to synthesize semantically valid API interactions and transform them into *Jazzer*-compatible fuzzing harnesses suitable for OSS-Fuzz. Our goal is to reduce manual effort while preserving determinism and reproducibility.

We evaluate *arrabida* through a preliminary multi-case study on three OSS-Fuzz Java projects. We compare generated and manually written drivers along three dimensions: (i) *coverage and behavioral exploration*, (ii) *fault discovery* (unique crashes), and (iii) *execution efficiency*. This evaluation provides an initial assessment of the feasibility and practical utility of automated fuzz driver generation. Our contributions are:

1. **Automated driver generation approach.** *arrabida* leverages search-based test generation and static-analysis–guided transformations to synthesize structured fuzz drivers for Java libraries.
2. **Tool implementation.** We provide an end-to-end pipeline that takes arbitrary Java projects and produces ready-to-run *Jazzer* drivers compatible with OSS-Fuzz.
3. **Preliminary multi-case evaluation.** We empirically compare *arrabida*-generated and manually written drivers on three OSS-Fuzz projects, analyzing coverage, exploration behavior, execution efficiency, and unique crash discovery.

2. Motivational Example

To tailor a fuzzer workflow to software projects, practitioners write small snippets of code, the fuzz driver. This auxiliary code—also referred to as a fuzz harness—bridges the gap between the fuzzing engine and the system under test. This driver is responsible for translating arbitrary byte streams into valid objects, enforcing API preconditions, and filtering expected behavior from true crashes. A poorly constructed driver can suppress actual vulnerabilities, generate spurious crashes, or leave significant attack surface unexplored.

To illustrate these challenges, we consider the *JavaPoet* library,[1] a Java code-generation framework exposing more than 20 public classes and over 300 public methods. Such libraries provide numerous interdependent entry points that require well-formed identifiers,

[1]We rely on the OSS-Fuzz version of *JavaPoet*; although discontinued upstream, this version remains widely used and continuously fuzz-tested.

Listing 1. Manually-written driver for the *JavaPoet* project extracted from OSS-Fuzz [LCC 2024].

```
public class TypeSpecFuzzer {
  public static void fuzzerTestOneInput(FuzzedDataProvider data) {
    try{
        TypeSpec taco = TypeSpec.classBuilder(data.consumeString(10))
            .addMethod(MethodSpec.methodBuilder(data.consumeString(20))
                .addAnnotation(Override.class)
                .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
                .returns(String.class)
                .addCode(data.consumeRemainingAsString())
                .build())
            .build();
        taco.hashCode();
    }
    catch (java.lang.IllegalArgumentException e) {} // ignore expected exception
  }
}
```

valid invocation orders, and complex builder-pattern configurations. Even a conservative approximation of possible call sequences quickly leads to a combinatorial explosion, making manual driver construction brittle and error-prone.

Unlike unit tests, which encode fixed scenarios with assertions, a fuzz driver is a long-running, assertion-free harness. It must repeatedly translate arbitrary inputs into valid API objects, isolate state across iterations, and distinguish expected exceptions from crashes so that feedback-driven exploration remains meaningful. Failure in any of these dimensions degrades fuzzing effectiveness and can obscure security-relevant behavior.

Listing 1 shows a manually written fuzz driver for *JavaPoet*, used in OSS-Fuzz [SquareDev 2024]. The method `fuzzerTestOneInput` is repeatedly invoked by the fuzzing engine, each time receiving a fresh stream of bytes encapsulated in a `FuzzedDataProvider` instance.

This provider is responsible for transforming raw input bytes into structured values. For instance, `consumeString(10)` generates a string of up to 10 characters, while `consumeRemainingAsString()` consumes the remaining input. These bounds are crucial to control input size and avoid pathological cases that would hinder throughput.

The driver constructs a `TypeSpec` object using a builder pattern, where multiple API calls must follow a valid ordering. For example, method and annotation definitions must be added before invoking `build()`. Violating such constraints typically results in exceptions that are not indicative of real faults.

To address this, the driver explicitly suppresses expected exceptions such as `IllegalArgumentException`. This ensures that the fuzzer focuses on unexpected behaviors, which are more likely to correspond to genuine bugs or vulnerabilities.

Overall, this example highlights three key challenges in manual fuzz driver construction: (i) correctly mapping raw inputs into structured API calls, (ii) respecting API usage constraints and call ordering, and (iii) distinguishing expected from unexpected failures during execution.

These aspects illustrate why manually developed fuzz drivers are costly to write, fragile

to maintain, and a limiting factor in scalable vulnerability discovery. To reduce this burden, we introduce (in Section 3) our approach, *arrabida*, which systematically transforms automatically generated test cases into fuzz drivers.

3. *arrabida*

This section presents *arrabida*, an automated approach for generating fuzz drivers for Java libraries by transforming search-based unit tests into *Jazzer*-compatible harnesses.

3.1. Overview

arrabida is based on the observation that fuzz drivers and unit tests share similar execution structures, including object construction, ordered API invocations, and exception handling. The key difference lies in intent: unit tests validate expected behavior, whereas fuzz drivers aim to maximize behavioral exploration under adversarial inputs.

arrabida exploits this overlap by generating unit tests for a target class and systematically transforming them into fuzz drivers. Test generation is delegated to EvoSuite [Fraser and Arcuri 2011], which produces semantically valid API interactions. These tests are then rewritten into fuzzing harnesses that map raw inputs to structured calls, enforce API constraints, and suppress expected exceptions.

Unlike approaches based on client-code mining or reuse of existing tests [Babić et al. 2019, Jeong et al. 2023], *arrabida* generates execution scenarios from scratch, reducing manual effort while preserving determinism and reproducibility.

3.2. Overall Architecture of *arrabida*

arrabida requires three inputs: (i) the project source code for static analysis, (ii) a target class or method to define the attack surface, and (iii) classpath dependencies to ensure correct test generation and driver compilation, keeping adoption overhead low while enabling reproducible and semantically valid driver synthesis. *arrabida* performs the following steps:

1. Parses the project and performs static analysis (3.2.1),
2. Invokes EvoSuite to generate unit tests (3.2.2),
3. Filters and selects tests suitable for transformation (3.2.3),
4. Rewrites the selected tests into a *Jazzer* fuzz driver (3.2.4), and
5. Applies driver-level optimizations for scalability and signal quality (3.2.5).

The following subsubsections describe these steps.

3.2.1. Static Analysis of Source Code

The first pipeline stage inspects the target project's source code to collect information needed for safe and effective driver construction:

- **API signatures.** We extract method signatures (names, parameter and return types) for the public methods of the target class. These define the API surface that unit tests and drivers are allowed to exercise.

- **Exception information.** We record declared exceptions for the target class and its public methods and derive a consistent handling order (e.g., more specific exceptions before generic ones). This catalog is later used to distinguish expected error conditions from potential vulnerabilities.
- **Type dependencies.** We track parameter and field types referenced by the target class so that subsequent stages can construct valid argument objects from fuzzer inputs.

This static analysis step constrains the subsequent generation and transformation stages to rely only on the public, documented interface of the project, which is crucial for producing deployable drivers.

3.2.2. Test Case Generation with EvoSuite

Following the static analysis phase, *arrabida* employs EvoSuite [Fraser and Arcuri 2021, Fraser and Arcuri 2011, Fraser and Arcuri 2014a] to automatically generate unit test cases for the designated target class. EvoSuite is a state-of-the-art tool in the Search-Based Software Testing community [Fraser and Arcuri 2014b, Fraser and Arcuri 2016, Campos et al. 2019, Panichella et al. 2020, Jahangirova and Terragni 2023], capable of synthesizing tests by leveraging evolutionary algorithms to maximize coverage. Its output consists of compilable Java test classes that exercise the APIs of the SUT through automatically constructed input values, method invocations, and exception handling logic.

The rationale for integrating EvoSuite into our pipeline is twofold. First, it capitalizes on more than a decade of research in automated test generation, ensuring that the produced tests are both syntactically valid and semantically meaningful with respect to the target class. Second, EvoSuite's focus on coverage-oriented test synthesis provides a natural bridge to fuzz driver construction, as fuzzing similarly aims to explore diverse execution paths and corner cases.

During this phase, EvoSuite is executed with the project's classpath, including the target class and its dependencies. This focused approach ensures that the generated tests are highly relevant to the desired fuzz target. The tests generated include not only direct method calls but also sequences of API invocations that initialize objects, propagate parameters, and capture exception-handling behavior. Although these tests are not yet fuzz drivers, they embody the essential ingredients of driver design: valid entry points, structured argument handling, and resilience against common runtime exceptions. EvoSuite's strength in generating diverse tests that achieve high code coverage provides an excellent foundation. The resulting suite of unit tests forms the initial candidate pool from which *arrabida* subsequently selects the most relevant cases for transformation. This selection process, described in Section 3.2.3, ensures that only tests contributing meaningful API coverage and context are retained for fuzz driver synthesis.

3.2.3. Selection of Relevant Test Cases

The unit tests generated by EvoSuite serve as a valuable foundation, but not all of them are equally suitable for transformation into fuzz drivers. Automated test generation often produces a mixture of useful, redundant, and semantically irrelevant cases. For example,

tests may repeatedly exercise trivial methods, encode constant values that are unlikely to generalize under fuzzing, or generate redundant exception checks that do not contribute to exploring new execution paths.

To address this challenge, *arrabida* applies a static analysis to the generated test suite to identify and retain only the most relevant test cases. The selection process considers a four-step filtering pipeline:

- **Target reachability.** Tests that do not exercise the target class (or method, if specified) in a dry run are discarded.
- **Public-API compliance.** Tests that access non-public methods or rely on test-specific utilities are removed, as fuzz drivers should interact only through the public API surface.
- **Transformability.** Tests whose behavior is dominated by hard-coded literals or degenerate control flow (e.g., always taking the same path regardless of input) are excluded, since they cannot be meaningfully parameterized by fuzz inputs.
- **Expected-exception cataloging.** For the remaining tests, we record which declared exceptions are observed in practice. These exceptions are later treated as expected and suppressed to reduce spurious crash reports.

The result is a refined set of unit tests that represent meaningful, public, and parameterizable API usage scenarios.

3.2.4. Transformation into Fuzz Driver

arrabida then transforms the selected tests into a single *Jazzer*-compatible fuzz driver. This involves several systematic rewrites:

- **Decoupling from EvoSuite.** Removes EvoSuite scaffolding (*e.g.*, runners, base classes, utilities), limiting dependencies to the SUT.
- **Unit-test artifact removal.** Assertions, `fail()` calls, and test harness constructs are stripped, as fuzzing is concerned with crashes and unexpected behavior, not explicit oracles.
- **Fuzzer entrypoint synthesis.** We generate the canonical `fuzzerTestOneInput(FuzzedDataProvider data)` entrypoint and refactor selected test bodies into internal helper methods invoked from this entrypoint.
- **Literal-to-input rewriting.** Constants in the original tests are replaced with calls to `FuzzedDataProvider` methods (e.g., `consumeString`, `consumeInt`, `consumeBytes`), turning each test into a parameterized fragment driven by fuzz input. For enums or small sets of alternatives, we map fuzzer data into choices over admissible values.
- **Exception triage.** Declared, expected exceptions are caught and ignored; unexpected throwables are allowed to surface as potential security-relevant crashes.

Compared to the manually written driver shown in Listing 1, the generated driver (Listing 2) differs in both structure and generality. While Listing 1 encodes a single, hand-crafted interaction sequence, Listing 2 aggregates multiple interaction patterns derived from distinct EvoSuite test cases. Each pattern is parameterized by fuzz input, enabling

Listing 2. Driver generated by *arrabida* for *JavaPoet*.

```java
public class TypeSpec_EvosuiteFuzzer {
    private static String generateStackTraceKey(Throwable e) {
        // Each exception generates a key based on StackTrace.
        // .. so, we do not report the exact same exception.
    }
    public static void test13(FuzzedDataProvider data) throws Throwable {
        Object[] objectArray0 = new Object[3];
        TypeSpec.Builder typeSpec_Builder0 =
            TypeSpec.anonymousClassBuilder(data.consumeString(50), objectArray0);
        CodeBlock codeBlock0 =
            CodeBlock.of(data.consumeString(100), objectArray0);
        typeSpec_Builder0.addStaticBlock(codeBlock0);
        TypeSpec typeSpec0 = typeSpec_Builder0.build();
        try {
            typeSpec0.hashCode();
        } catch (IllegalStateException e) { throw e; }
    }
    public static void test80(FuzzedDataProvider data) throws Throwable {
        Class<TypeParameterElement> class0 = TypeParameterElement.class;
        ClassName className0 = ClassName.get(class0);
        TypeSpec.Builder typeSpec_Builder0 = TypeSpec.classBuilder(className0);
        typeSpec_Builder0.superclass((Type) class0, data.consumeBoolean());
        TypeSpec typeSpec0 = typeSpec_Builder0.build();
        typeSpec0.hashCode();
    }
    // ... several other tests generated by EvoSuite and selected by arrabida

    public static void fuzzerTestOneInput(FuzzedDataProvider data) throws Throwable {
        try {
            test13(data);
        } catch (IllegalArgumentException e) {
        } catch (Throwable e) { raisedExceptions.put(generateStackTraceKey(e), e); }

        try {
            test80(data);
        } catch (IllegalArgumentException e) {
        } catch (Throwable e) { raisedExceptions.put(generateStackTraceKey(e), e); }
        //... calls for other tests
    }
}
```

systematic exploration of variations that would be impractical to encode manually. Moreover, whereas the manual driver relies on developer intuition to ensure valid API usage, the generated driver inherits structurally valid call sequences from EvoSuite, which are then generalized to accept arbitrary inputs. This shift from manual design to automated generalization is the key mechanism enabling broader behavioral exploration.

The resulting driver bridges fuzzer-generated byte streams to structured, semantically valid API invocations, suitable for integration into scalable fuzzing workflows, although further large-scale evaluation is required to fully assess its effectiveness in such settings.

3.2.5. Driver Optimization

To improve efficiency, *arrabida* applies optimizations such as disciplined input consumption, execution of multiple interaction patterns per iteration, and crash de-duplication based on exception signatures. These techniques increase exploration while reducing redundant failures and improving signal quality.

- **Disciplined input consumption.** We centralize access to the `FuzzedDataProvider`, slice inputs deterministically among helper methods, and enforce bounds to avoid premature exhaustion or inconsistent states.
- **Iteration-resilient execution.** Within one fuzz iteration, multiple helper methods are executed using the same input. Unexpected exceptions are recorded but do not immediately abort the iteration, preventing single "blocking" paths from starving exploration. At the end of the iteration, recorded unexpected exceptions are surfaced to the fuzzer as crashes.
- **Crash de-duplication.** The driver maintains a campaign-wide set of crash signatures (based on exception type and project-local stack frames). Only new signatures are reported, reducing duplicate findings and focusing triage effort on distinct failure modes.

These optimizations are geared toward security-oriented fuzzing at scale, where noisy drivers can quickly overwhelm analysts with redundant crashes or stall coverage growth.

Listing 2 illustrates a fuzz driver automatically generated by *arrabida*. Each method (e.g., `test13`, `test80`) corresponds to a unit test originally synthesized by EvoSuite and later transformed into a fuzzable execution fragment. Unlike the manually written driver in Listing 1 , which encodes a single API usage scenario, the generated driver aggregates multiple independent interaction patterns into a single harness. These patterns are invoked sequentially within `fuzzerTestOneInput`, sharing the same input source (`FuzzedDataProvider`).

The parameter `data` is used to replace constant values from the original test cases with dynamically generated inputs. For example, calls such as `data.consumeString(n)` allow the driver to explore different API configurations across fuzzing iterations, while preserving the structural constraints inferred from the original tests. Additionally, exceptions are handled in two tiers: expected exceptions (e.g., `IllegalArgumentException`) are suppressed, while unexpected exceptions are recorded and reported after execution. This design prevents premature termination of an iteration while still surfacing distinct failure modes.

4. Preliminary Multi-Case Evaluation

Our goal is to assess, through a preliminary multi-case evaluation, whether fuzz drivers generated by *arrabida* provide practical value in security-focused fuzzing workflows against manually written drivers across three real-world Java libraries from OSS-Fuzz.

In this paper, we investigate the following research questions:

RQ1: Coverage. Do fuzz drivers generated by *arrabida* achieve coverage comparable to or higher than their manually written counterparts?

RQ2: Exploration Efficiency. How do *arrabida*-generated drivers compare in execution rate, iteration count, and feature exploration?

RQ3: Failure Discovery. Are automatically generated drivers capable of revealing meaningful failures (unique crashes) relative to the manual drivers?

4.1. Subjects

We evaluate *arrabida* on three Java libraries actively fuzzed through OSS-Fuzz. We intentionally limited the number of subjects to three in order to conduct a controlled and

Table 1. Details of the set of subjects used in our preliminary evaluation.

Project	Domain	Version	Target Class	LoC	# Public Methods
brotli-java	Brotli (de)compression	v1.1.0	BrotliInputStream	91	6
cbor-java	CBOR en/decoding for compact binary data	v0.8	CborDecoder	206	18
httpcomponents-core	HTTP components in Java	v5.4-alpha1	HttpRequester	413	24

in-depth preliminary evaluation. This design choice prioritizes internal validity and detailed analysis over broad generalization, which we explicitly leave for future work. Each subject satisfies the following criteria:

- It is a real-world, security-relevant Java project used by OSS-Fuzz.
- It contains at least one manually written fuzz driver serving as a baseline.
- It exposes a non-trivial public API surface suitable for fuzzing.
- Its build configuration is compatible with our local OSS-Fuzz environment.

This sampling strategy aims not at statistical generalization, but at evaluating whether *arrabida* performs robustly across qualitatively different fuzzing targets. Table 1 lists the set of projects and briefly summarizes their metadata.

4.2. Procedure

For each subject, we perform the following steps:

1. **Project preparation.** Clone the project revision used by OSS-Fuzz and build it locally using standard configuration.
2. **Baseline identification.** Extract the manually written fuzz driver and determine the targeted API class.
3. **Driver generation with *arrabida*.** Run *arrabida* on the target class. EvoSuite executes for three minutes [Shamshiri et al. 2015] without mocks, ensuring that the generated tests—and thus the synthesized driver—depend only on the project's public API surface.
4. **Fuzzing campaigns.** Using a local OSS-Fuzz instance, run *Jazzer* on both drivers (baseline and *arrabida*) for one hour each, with identical configurations. Fuzzing continues after the first crash to allow enumeration of distinct failures.

This procedure ensures comparability across drivers and subjects, while controlling for environmental factors known to influence fuzzing outcomes.

4.3. Experimental Metrics

To characterize the behavior of each driver, we collect the following metrics from *Jazzer* logs:

- **Coverage:** number of unique control-flow edges reached during fuzzing.
- **Iterations:** total number of fuzzing iterations completed under the time budget.
- **Features:** distinct coverage features (or exploration signals) exercised.
- **Execution rate:** iterations per second, reflecting input-processing efficiency.

- **Unique crashes:** failures deduplicated via a fingerprint ⟨exception class, first three stack frames⟩, preventing inflation by repeated manifestations of the same fault.

These metrics capture complementary dimensions of fuzzing efficiency, effectiveness, and fault-detection capability. We emphasize *unique crashes*, as they correspond to distinct failure modes relevant for vulnerability analysis.

4.4. Experimental Setup

Experiments ran on an Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, 8 cores, 32 GB of RAM and Ubuntu 24.04.2 LTS. We used OpenJDK Corretto v11.0.26.4.1 and relied on EvoSuite v1.2.0, to generate the tests and therefore transform into drivers, and on *Jazzer* v0.19.0 to perform the fuzzing. The manually-written drivers were extracted from OSS-Fuzz commit bd2ee62.

5. Analysis and Discussion of Results

We analyze the results of our preliminary multi-case evaluation with respect to the three research questions.

5.1. RQ1: Coverage

Table 2 reports edge coverage for manually written and *arrabida*-generated drivers, measured over the entire project. Across all three subjects, *arrabida* consistently achieves higher coverage. This improvement is primarily due to the ability of *arrabida* to systematically generate and combine multiple API usage scenarios derived from test cases. In contrast, manual drivers tend to focus on specific execution paths, limiting their coverage of the overall API surface.

The most pronounced gain is observed in **httpcomponents-core**, a complex networking library with a large and heterogeneous API. Here, the manual driver exercises a narrow, realistic usage pattern, whereas the *arrabida*-generated driver invokes a broader set of public methods and constructs richer fixtures. As a result, additional branches in request construction and error-handling logic are explored, expanding the effective attack surface exercised by fuzzing.

For **brotli-java** and **cbor-java**, the coverage improvements are more modest but remain positive. In both cases, *arrabida* explores additional internal paths and API methods while preserving the core behaviors already reached by the manual drivers.

Answer to RQ1

arrabida-generated drivers achieve higher coverage across all subjects, including a substantial gain on the HTTP protocol stack (httpcomponents-core), thus exposing a larger effective attack surface to fuzzing.

5.2. RQ2: Exploration Efficiency

Table 2 also reports execution efficiency metrics, including iterations, execution rate, and number of explored features. The results reveal a trade-off between raw throughput and behavioral exploration.

For **brotli-java**, the *arrabida*-generated driver substantially outperforms the manual driver in both iteration count and execution rate, while also achieving higher coverage and

Table 2. Comparison of manually written vs. *arrabida*-generated drivers.

Project	Coverage		Iterations		Exec. Rate		Features	
	Manual	*arrabida*	Manual	*arrabida*	Manual	*arrabida*	Manual	*arrabida*
brotli-java	617	**711**	2,228,172	**46,908,495**	618	**13,026**	2,971	**3,245**
cbor-java	502	**606**	**1,100,963**	897,672	**934**	249	2,756	**3,218**
httpcomponents-core	581	**1,691**	**65,826,456**	2,022,335	**18,280**	561	700	**4,548**

more features. In contrast, for **cbor-java** and **httpcomponents-core**, the manual drivers execute more iterations per second, but *arrabida* explores a larger number of distinct features. This behavior can be explained by structural differences between the drivers. *arrabida*-generated drivers construct richer object graphs and execute multiple API interaction patterns within a single fuzzing iteration. As a result, each iteration explores a larger portion of the program state space, albeit at a higher computational cost. In contrast, manually written drivers are typically optimized for specific and well-understood usage scenarios. This leads to faster execution and higher iteration throughput, but limits the diversity of reachable program states. Consequently, although manual drivers may achieve higher execution rates, they often explore fewer distinct behaviors.

These results suggest that execution rate alone is not a sufficient proxy for fuzzing effectiveness. Instead, the ability to reach diverse program states plays a more critical role in improving coverage and fault discovery.

Answer to RQ2

Although *arrabida*-generated drivers are not uniformly faster, they consistently achieve richer behavioral exploration, indicating that higher throughput does not necessarily translate into better fuzzing effectiveness.

5.3. RQ3: Failure Discovery

Table 3 summarizes the distinct crashes observed for each subject. Crashes are considered unique if they differ in exception type or in the first three stack frames. Overall, *arrabida*-generated drivers expose more distinct failure modes than manually written drivers, with partial overlap between the two. This difference can be attributed to the way *arrabida* combines multiple API interaction patterns into a single driver. By generalizing several EvoSuite-generated test cases, the resulting driver is able to exercise a broader range of input configurations and call sequences than manually written drivers, which are typically designed around a limited set of expected use cases. This increased diversity in exercised behaviors raises the likelihood of reaching corner-case states, which in turn contributes to the discovery of additional failure modes. In contrast, manually written drivers may overlook such states due to their more focused and deterministic structure.

Each approach reveals crashes missed by the other.

For **brotli-java**, only the *arrabida*-generated driver triggers failures, including `IllegalStateException` and `NullPointerException`, suggesting exploration of decompression states not exercised by the manual harness.

For **cbor-java**, both drivers uncover severe resource-exhaustion failures (`StackOverflowError` and `OutOfMemoryError`), which are critical denial-of-service risks in parsers. The manual driver additionally triggers numerous

Table 3. Unique crashes per subject and driver.

Project	Crash	Manual	*arrabida*
brotli-java	⟨IllegalStateException, C1⟩	0	1
	⟨NullPointerException, D1⟩	0	1
cbor-java	⟨StackOverflowError, E1⟩	1	1
	⟨StackOverflowError, E2⟩	1	1
	⟨OutOfMemoryError, F1⟩	1	1
	⟨OutOfMemoryError, F2⟩	1	0
	⟨IllegalArgumentException, G1⟩	3,707	0
	⟨IllegalArgumentException, G2⟩	2,760	0
	⟨IllegalArgumentException, G3⟩	3,529	0
httpcomponents-core	⟨FuzzerSecurityIssueMedium, A2⟩	1	4
	⟨NullPointerException, D3⟩	0	1
	⟨NullPointerException, D4⟩	0	1
	⟨NullPointerException, D5⟩	0	3
	⟨URISyntaxException, L1⟩	0	1
	⟨URISyntaxException, L2⟩	0	22
Distinct crashes (3 subjects)		8 (Manual)	11 (*arrabida*)

`IllegalArgumentException`-based crashes, which are suppressed by *arrabida* as expected exceptions.

For **httpcomponents-core**, both drivers reproduce a known `FuzzerSecurityIssueMedium` vulnerability. Beyond this, the *arrabida*-generated driver exposes additional `NullPointerException` and `URISyntaxException` failures, indicating exploration of alternative request and URI construction paths.

Answer to RQ3

Across the three subjects, *arrabida*-generated drivers expose more distinct failure modes (11 vs. 8), including crashes not reached by the manual drivers in brotli-java and httpcomponents-core, while still reproducing key high-impact issues. This indicates that automated drivers can contribute meaningfully to vulnerability discovery, complementing existing manually crafted harnesses.

6. Threats to Validity

Our findings rely on three Java/JVM libraries—*brotli-java*, *cbor-java*, and *httpcomponents-core*—which limits the generalizability of the results. Differences in API structure, semantic constraints, or input-processing behavior across these subjects may influence the relative performance of manually written versus automatically generated drivers. Moreover, both EvoSuite and *Jazzer* were executed using fixed seeds and default configurations, with a one-hour fuzzing budget; alternative settings, longer campaigns, or different fuzzing guidance may lead to different outcomes. We attempted to mitigate these issues by fixing all parameters, documenting the full experimental setup, and providing all scripts and artifacts to support independent replication and sensitivity analysis.

Our analysis is based on widely adopted fuzzing metrics (coverage, iterations, execution rate, feature points, unique crashes), which approximate—but do not fully capture—security-relevant behavior such as exploitability or fault reachability. Some crashes may still be false positives, pending developer triage, and manually written drivers

in OSS-Fuzz may already reflect historical hardening, potentially skewing comparisons. Thus, the results should be interpreted as characterizing the *current* behavior of *arrabida* on these three subjects rather than establishing broader conclusions about Java fuzzing or alternative engines. Despite these limitations, the study provides a reproducible and transparent basis for evaluating feasibility and guiding future, larger-scale assessments.

7. Related Work

Automated fuzz driver generation has been explored through different sources of program knowledge. A first class of approaches infers API usage from *client code* or execution traces, as in Fudge [Babić et al. 2019], FuzzGen [Ispoglou et al. 2020], APICraft [Zhang et al. 2021], and WINNIE [Jung et al. 2021]. While effective, these techniques depend on the availability and quality of client artifacts and may inherit usage-specific biases. A second line of work leverages *existing test artifacts*. UTopia [Jeong et al. 2023] transforms unit tests into fuzz drivers but requires mature test suites, while Hopper [Chen et al. 2023] learns API behavior through interpretative fuzzing at the cost of additional complexity. More recently, LLM-based approaches such as PromptFuzz [Lyu et al. 2024] generate drivers via prompt-based synthesis. These methods offer flexibility but raise challenges related to non-determinism, prompt sensitivity, and reproducibility. In contrast, *arrabida* generates fuzz drivers from scratch using search-based test generation combined with static analysis, emphasizing reproducibility and independence from external artifacts.

8. Conclusions and Future Work

We introduced *arrabida*, a fully automated approach for synthesizing fuzz drivers for Java libraries by transforming search-based unit tests into *Jazzer*-compatible harnesses. Our evaluation on three OSS-Fuzz subjects shows that automatically generated drivers can achieve competitive coverage, explore substantially more behavioral features, and uncover failures not exposed by manually written drivers. These findings indicate that systematic, test-derived driver synthesis is a viable path toward scaling fuzzing in the Java ecosystem. However, given the narrow scope of this preliminary study, a stronger empirical foundation is needed before broader claims can be made.

Future work includes expanding the subject pool, exploring multiple EvoSuite seeds to measure variability, evaluating robustness under longer fuzzing campaigns, and incorporating security-focused metrics such as *time-to-first-crash*. We also plan to investigate hybrid strategies that combine search-based generation with lightweight semantic inference to strengthen both coverage and failure diversity.

Data Availability

To support reproducibility, we provide all scripts, drivers, and processed data in a publicly accessible replication package: `https://osf.io/wjkx5/overview?view_only=b8193c6be5304ca898cc0e3d16f905af`.

Acknowledgments

Leo Fernandes conducted this work during a sabbatical as a postdoctoral researcher at the FEUP, University of Porto, and gratefully acknowledges the institution's support and academic environment. José Campos was supported by the LASIGE Research Unit, ref. UID/00408/2025, DOI

10.54499/UID/00408/2025. Juvenal Carvalho holds a scholarship funded by Fundação para a Ciência e Tecnologia (FCT), ref. SFRH/BD/151433/2021. This work was also supported by the SALVE project (DOI 10.54499/2024.14936.PEX) and HPC (2025.08039.CPCA).

References

Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L., and Wang, W. (2019). FUDGE: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, pages 975–985, Tallinn, Estonia. ACM.

Böhme, M., Cadar, C., and Roychoudhury, A. (2021). Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86.

Campos, J., Panichella, A., and Fraser, G. (2019). EvoSuite at the SBST 2019 Tool Competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32.

Chen, P., Xie, Y., Lyu, Y., Wang, Y., and Chen, H. (2023). Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1600–1614, New York, NY, USA. Association for Computing Machinery.

Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 416–419, New York, NY, USA. Association for Computing Machinery.

Fraser, G. and Arcuri, A. (2014a). A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2).

Fraser, G. and Arcuri, A. (2014b). Evosuite at the second unit testing tool competition. In Vos, T. E., Lakhotia, K., and Bauersfeld, S., editors, *Future Internet Testing*, pages 95–100, Cham. Springer International Publishing.

Fraser, G. and Arcuri, A. (2016). EvoSuite at the SBST 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, page 33–36, New York, NY, USA. Association for Computing Machinery.

Fraser, G. and Arcuri, A. (2021). Evosuite. `https://github.com/EvoSuite/evosuite`. Accessed: 28 Aug 2025.

Gao, W., Pham, V.-T., Liu, D., Chang, O., Murray, T., and Rubinstein, B. I. (2023). Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, page 47–55, New York, NY, USA. Association for Computing Machinery.

Ispoglou, K. K., Austin, D., Mohan, V., and Payer, M. (2020). Fuzzgen: automatic fuzzer generation. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA. USENIX Association.

Jahangirova, G. and Terragni, V. (2023). Sbft tool competition 2023 - java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 61–64.

Jeong, B., Jang, J., Yi, H., Moon, J., Kim, J., Jeon, I., Kim, T., Shim, W., and Hwang, Y. H. (2023). Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2676–2692.

Jung, J., Tong, S., Hu, H., Lim, J., Jin, Y., and Kim, T. (2021). Winnie : Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.

LCC, G. (2024). Typespecfuzzer. `https://raw.githubusercontent.com/google/oss-fuzz/refs/heads/master/projects/javapoet/TypeSpecFuzzer.java`. Accessed: 28 Aug 2025.

Lyu, Y., Xie, Y., Chen, P., and Chen, H. (2024). Prompt fuzzing for fuzz driver generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3793–3807, New York, NY, USA. Association for Computing Machinery.

Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2021). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331.

Panichella, A., Campos, J., and Fraser, G. (2020). EvoSuite at the SBST 2020 Tool Competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 549–552, New York, NY, USA. Association for Computing Machinery.

Serebryany, K. (2017). OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC. USENIX Association.

Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., McMinn, P., and Arcuri, A. (2015). Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 201–211.

SquareDev (2024). Javapoet. `https://github.com/square/javapoet`. Accessed: 28 Aug 2025.

Team, G. O. S. S. (2023). Ai-powered fuzzing: Breaking the bug hunting barrier. Google Security Blog. Accessed: 24 August 2025.

Zhang, C., Lin, X., Li, Y., Xue, Y., Xie, J., Chen, H., Ying, X., Wang, J., and Liu, Y. (2021). APICraft: Fuzz driver generation for closed-source sdk libraries. In *Proceedings of the 30th USENIX Security Symposium*, pages 2811–2828. USENIX Association.

Zhang, C., Zheng, Y., Bai, M., Li, Y., Ma, W., Xie, X., Li, Y., Sun, L., and Liu, Y. (2024). How effective are they? exploring large language model based fuzz driver generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1223–1235, New York, NY, USA. Association for Computing Machinery.

Zhu, X., Wen, S., Camtepe, S., and Xiang, Y. (2022). Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s).